

Python

11 maggio 2025

Indice

I	Linguaggio	13
1	Introduzione	15
1.1	Introduzione	15
1.1.1	Caratteristiche del linguaggio	15
1.1.2	Setup	16
1.2	Esecuzione	16
1.3	Ottenere aiuto	18
1.4	Gestione sistema	18
1.4.1	Aggiornamento di sistema periodico	18
1.4.2	Formati pacchetto	19
1.4.3	Pacchetti	19
1.4.4	Virtual environments	19
1.5	ipython	20
1.5.1	Configurazione	20
1.5.2	Magic commands utili	21
1.5.3	Comandi di shell	21
2	Dati	23
2.1	Introduzione	23
2.2	Tipi nativi	24
2.3	Classificazione dei tipi di base	25
2.4	Numeri	26
2.5	Date e ore	27
2.6	Sequenze: stringhe, liste e tuple	28
2.6.1	Operatore di slice (selezione da sequenza) e indici	28
2.6.2	Stringhe	30
2.6.3	Liste	34
2.6.3.1	Definizione	34
2.6.3.2	Manipolazione e modifica	34
2.6.3.3	Metodi utili per le liste	35
2.6.3.4	Liste nested	35
2.6.3.5	List comprehensions	36
2.6.4	Tuple	38
2.6.5	Sequence unpacking	38
2.7	Classi mapping e set	39
2.7.1	Dict	39
2.7.1.1	Metodi utili	40
2.7.1.2	Dict comprehension	40

2.7.2	Sets	40
2.7.2.1	Operatori/metodi utili	41
2.7.2.2	Set comprehension	42
2.8	Altri tipi utili	42
2.8.1	<code>namedtuple</code>	42
2.8.2	<code>Counter</code>	42
2.9	Type annotation	43
2.9.1	Sintassi	43
2.9.2	Checking	44
2.9.3	Tipi utilizzabili per variabili	44
2.9.4	Creazione di alias	45
2.9.5	Annotazione di funzioni	46
2.9.6	Annotazione di metodi in classi	46
3	Controllo del flusso	49
3.1	Costrutti condizionali: <code>if</code> e <code>while</code>	49
3.1.1	<code>if</code>	49
3.1.2	<code>match</code>	50
3.1.3	<code>while</code>	50
3.1.4	<code>break</code> , <code>continue</code> ed <code>else</code>	50
3.2	Condizioni e test logici	50
3.2.1	Test verità	50
3.2.2	Operatori booleani	51
3.2.3	Comparazioni	51
3.2.4	Comparazioni concatenate	52
3.2.5	Check appartenenza: <code>in</code> e <code>not in</code>	52
3.2.6	Comparare sequenze e altri tipi	52
3.3	Looping su oggetti: <code>for</code>	53
3.3.1	Looping in sequenze (stringhe, liste, tuple)	53
3.3.2	Looping nei dict	54
3.3.3	Looping sui set	54
3.3.4	L'utilizzo di <code>range</code>	54
4	Funzioni	57
4.1	Definizione	57
4.1.1	Argomenti	58
4.1.2	Stringa di documentazione	58
4.2	Chiamata	59
4.2.1	Valutazione dei valori di default	59
4.2.2	Valore ritornato	60
4.2.3	Liste/tuple/dict come parametri di chiamata	60
4.3	Regole di scope	61
4.3.1	Namespace	61
4.3.2	Attributi	61
4.3.3	Scoping	62
4.4	<code>lambda</code> e funzioni anonime	63
4.5	Programmazione funzionale	63
4.5.1	Funzioni classiche	64
4.5.1.1	<code>map</code> (= <code>Map/lappy</code>)	64
4.5.1.2	<code>itertools.starmap</code>	64

4.5.1.3	<code>functools.reduce</code>	65
4.5.1.4	<code>itertools.accumulate</code>	65
4.5.1.5	<code>filter</code>	65
4.5.2	Partialling	66
4.5.2.1	<code>functools.partial</code>	66
4.5.2.2	<code>functools.partialmethod</code>	66
4.5.3	Function factory	67
4.5.4	Composizione di funzioni	67
4.5.5	Decorators	68
4.5.5.1	Definizione e utilizzo	68
4.5.5.2	Decoratori già disponibili	69
4.5.5.3	Esempi di creazione/utilizzo di custom	69
4.5.6	Single e multiple dispatch	71
4.5.6.1	Single dispatch: <code>functools singledispatch</code> e <code>functools singledispatchmethod</code>	71
4.5.6.2	Multiple dispatch	72
5	Input/Output	75
5.1	Lettura/scrittura file testuali	75
5.1.1	Testo semplice	75
5.1.1.1	Lettura	76
5.1.1.2	Scrittura	76
5.1.2	Formati tabulari (csv, tsv)	76
5.1.2.1	Lettura	77
5.1.2.2	Scrittura	77
5.1.3	JSON	77
5.1.3.1	Scrittura	77
5.1.3.2	Lettura	78
5.1.3.3	Formati custom	78
5.2	Accesso al filesystem	78
5.2.1	Ottenere/cambiare directory di lavoro	79
5.2.2	Listing di directory e glob files	79
5.2.3	Creazione/rimozione di file e directory	79
5.2.4	Manipolazione di path e metodi utili	79
5.2.5	Creazione filename temporaneo	81
5.2.6	Uso di file e directory temporanei	81
5.3	Esecuzione di programmi esterni	82
6	Debugging ed eccezioni	85
6.1	Debugging	85
6.1.1	Ispezione della traceback	85
6.1.2	Esecuzione in modalità debugging	86
6.1.3	Eseguire script in modalità debugging	86
6.1.4	Un equivalente di <code>browser</code>	86
6.2	Gestire eccezioni	87
6.2.1	Sintassi minimale: <code>try except</code>	87
6.2.2	<code>else</code> e <code>finally</code> in <code>try</code>	88
6.3	Sollevare eccezioni	89
6.4	Creare ed utilizzare eccezioni custom	90
6.5	Sollevare warnings senza stoppare l'esecuzione	90

7	Object Oriented Programming	91
7.1	Classi	91
7.1.1	Definizione e scoping	91
7.1.2	Metodi	93
7.1.2.1	Definizione e chiamata	93
7.1.2.2	Costruttore custom	93
7.1.3	Condivisione dati	93
7.1.4	Data hiding	94
7.2	Ereditarietà	95
7.2.1	Singola	95
7.2.2	Multipia	95
7.3	Classi notevoli	96
7.3.1	<code>dataclass</code>	96
7.3.1.1	<code>field</code> : dati mutabili, valori default, parametri	96
7.3.1.2	Eseguire codice post inizializzazione: <code>post_init</code>	97
7.3.1.3	Freezing di una <code>dataclass</code>	98
7.3.1.4	Altri parametri utili del decoratore	98
7.3.2	Iteratori	98
7.3.2.1	Funzionamento	99
7.3.2.2	Implementazione mediante classe	100
7.3.2.3	Implementazione mediante espressioni generatrici	101
7.3.2.4	Implementazione mediante generatori	101
7.3.2.5	Funzioni e operatori utili su iteratori	102
7.3.2.6	Il modulo <code>itertools</code>	103
7.3.3	Context Managers	105
7.3.3.1	Implementazione mediante classe	105
7.3.3.2	Implementazione mediante generatore	105
8	Moduli e pacchetti	107
8.1	Introduzione	107
8.2	Moduli	108
8.2.1	Nome e namespace	108
8.2.2	Importazione dei moduli	108
8.2.3	Path di ricerca dei moduli	109
8.2.4	Templates	109
8.2.4.1	Modulo libreria	109
8.2.4.2	Script della libreria	110
8.3	Pacchetti	111
8.3.1	Struttura e <code>__init__.py</code>	111
8.3.1.1	Struttura semplice	111
8.3.1.2	Struttura con subpackages	111
8.3.2	<code>__init__.py</code> , <code>__all__</code> e <code>import *</code> da pacchetto	113
8.4	Packaging e distribuzione di pacchetti	113
8.4.1	Flow	113
8.4.2	<code>pyproject.toml</code>	113
8.4.3	Aggiornamento toolchain	114
8.4.4	Creazione del tree del pacchetto	114
8.4.5	Installare un pacchetto in modalità <code>devel</code>	114
8.4.6	Build di <code>sdist</code> e <code>wheel</code>	114
8.4.7	Upload a <code>pypi</code>	114

8.5	Altre utilità	115
8.5.1	Inserimento di script nel pacchetto	115
8.5.2	Documentazione	115
8.5.2.1	Setup	115
8.5.2.2	Doc-writing e reStructuredText	115
8.5.2.3	Building	116
8.5.2.4	Setup di readthedocs	117
8.5.3	Testing	117
8.5.4	Timing/temporizzazione	118
8.5.5	Profiling	118
8.5.5.1	Tempo	119
8.5.5.2	Memoria	119
8.5.6	Altri strumenti utili	119
9	Testing	121
9.1	Introduzione e concetti	121
9.1.1	Tipologie di testing	121
9.1.2	Test driven development	122
9.2	<code>unittest</code>	122
9.2.1	Test del valore ritornato	122
9.2.2	Test eccezioni	124
9.2.3	Test fixtures	125
II	Scientific Stack	127
10	Numpy	129
10.1	<code>L'ndarray</code>	130
10.1.1	Creazione e copia	130
10.1.2	Tipi (<code>dtype</code>): coercizione e testing	131
10.1.3	Forma, dimensioni e reshape	132
10.1.3.1	Forma e dimensioni	132
10.1.3.2	Reshaping	133
10.1.3.3	Ravelling/flattening	133
10.2	Indexing	134
10.2.1	Array unidimensionali	134
10.2.2	Array multidimensionali	135
10.2.3	Subarray come viste vs copie	138
10.2.4	Assegnazione e unicità degli indici	139
10.3	Elaborazioni di array	139
10.3.1	Inserimento/rimozione elementi (<code>insert</code> , <code>delete</code>)	139
10.3.2	Aritmetica vettorizzata	139
10.3.3	Operazioni insiemistiche	140
10.3.4	Concatenazione (<code>concatenate</code> , <code>vstack</code> , <code>hstack</code>)	140
10.3.5	Splitting (<code>split</code> , <code>vsplit</code> , <code>hsplit</code>)	141
10.3.6	Ripetizione/binding (<code>repeat</code> , <code>tile</code>)	142
10.3.7	Sorting/ordine (<code>sort</code> , <code>argsort</code>)	143
10.4	Universal functions	144
10.4.1	Introduzione	144
10.4.2	Metodi delle ufunction (<code>reduce</code> , <code>accumulate</code> , <code>outer</code>)	144

10.4.3	Creazione di ufunctions	146
10.5	Broadcasting	147
10.6	Altri argomenti	151
10.6.1	Lavorare con booleani	151
10.6.2	Array di stringhe	152
11	Pandas	155
11.1	Index	156
11.2	Series	156
11.2.1	Creazione e contenuto (array, index, name, dtype)	157
11.2.2	Indexing ([], .loc, .iloc)	157
11.2.3	Funzionalità per indici (filter, reindex, reset_index, rename)	159
11.2.4	Modifica di valori	160
11.2.5	Rimozione elementi (drop, del)	160
11.2.6	Indici, elaborazione vettorizzata, allineata, reindexing	161
11.2.7	Coercizione di tipo (astype)	162
11.2.8	Valori condizionali (ifelse): np.where, pd.Series.where	163
11.2.9	Applicazione di funzioni (map)	163
11.2.10	Applicazione di più funzioni (.agg)	163
11.2.11	Recode (map e replace)	164
11.2.12	Test di appartenenza (in, isin)	164
11.2.13	Dati mancanti (isna, notna, dropna, fillna)	165
11.2.14	Gestione duplicati (duplicated, unique, drop_duplicates)	165
11.2.15	Sorting (sort_index, sort_values)	166
11.2.16	Discretizzazione/creazione di classi (cut, qcut)	166
11.2.17	Dummy variables (get_dummies, str.get_dummies, idxmax)	168
11.2.18	Stringhe: Series.str	169
11.2.19	Date/ore: Series.dt e funzioni varie	172
11.2.20	Dati categorici: Categorical e Series.cat	174
11.2.21	Indici gerarchici (MultiIndex) nelle serie	176
11.2.21.1	Definizione	176
11.2.21.2	Indexing	176
11.2.21.3	Reshape	177
11.3	DataFrame	178
11.3.1	Creazione e contenuto (info, shape, index, columns, values, name)	178
11.3.2	Indexing ([], .loc, .iloc)	179
11.3.3	Selezione di righe con query	184
11.3.4	Selezione di colonne sulla base di tipo	184
11.3.5	Accesso a singoli numeri: .at, .iat	185
11.3.6	Aggiunta di colonne (assegnazione, insert, assign)	185
11.3.7	Modifica di valori (indexing e assegnazione: loc, iloc, at, iat)	187
11.3.8	Rimozione righe/colonne (drop, del)	187
11.3.9	Rinominare indici/colonne (rename)	188
11.3.10	Funzionalità per indici, reindexing, MultiIndex	189
11.3.10.1	Creare indici da colonne e viceversa (set_index, reset_index)	189
11.3.10.2	Reindexing (reindex, loc)	190

11.3.10.3 MultiIndex	190
11.3.11 Elaborazione allineata	191
11.3.12 Coercizione di tipi (<code>astype</code> , <code>transform</code>)	193
11.3.13 Aggregazione (<code>agg</code>)	193
11.3.14 Applicazione di funzioni	194
11.3.14.1 A righe e colonne (<code>apply</code> , <code>transform</code>)	194
11.3.14.2 A tutto il DataFrame (<code>map</code> e <code>pipe</code>)	195
11.3.14.3 Sfruttando metodi delle colonne (<code>Series.map</code>)	196
11.3.15 Ciclo su righe/colonne (<code>iterrows</code> , <code>items</code>)	196
11.3.15.1 Righe (<code>itertuples</code>)	196
11.3.15.2 Colonne (<code>for</code> secco, <code>items</code>)	197
11.3.16 Merge (<code>merge</code>)	197
11.3.17 Binding (<code>concat</code>)	199
11.3.18 Reshape	199
11.3.18.1 Sulla base di index (<code>stack</code> , <code>unstack</code>)	199
11.3.18.2 Senza index (<code>pivot</code> , <code>melt</code>)	200
11.3.19 Test di appartenenza (<code>in</code> , <code>isin</code>)	202
11.3.20 Dati mancanti (<code>count</code> , <code>isna</code> , <code>notna</code> , <code>dropna</code> , <code>fillna</code>)	203
11.3.21 Gestione duplicati (<code>duplicated</code> , <code>drop_duplicates</code>)	204
11.3.22 Sorting di righe/colonne (<code>sort_values</code> , <code>sort_index</code>)	205
11.4 Data I/O	206
11.5 Cookbook	208
11.5.1 Stampare localmente tutto il contenuto di un DataFrame	208
12 Matplotlib	209
12.1 Introduzione	209
12.2 Salvataggio figura	210
12.3 Impostazione layout figura ed esempi	212
12.3.1 Layout standard	212
12.3.2 Layout custom	214
12.4 Configurazioni	217
12.4.1 Ottenimento e modifica	217
12.4.2 Ripristino impostazioni default	218
12.4.3 Cambiare stile	218
12.5 Grafici utili	218
12.5.1 Linee	219
12.5.2 Diagramma a barre	219
12.5.3 Istogramma	220
12.5.4 Scatterplot	220
12.5.5 Matrice di scatterplot	223
12.5.6 Boxplot	223
12.5.7 Correlation matrix	226
III Cookbook	229
13 Algebra lineare	231
13.1 Setup	231
13.2 Algebra lineare con numpy	231

14	Statistica descrittiva	233
14.1	Info generali	234
14.2	Univariate	234
14.2.1	Statistiche varie	234
14.3	Bivariate	235
14.3.1	Tabelle di contingenza	235
14.3.2	Covarianza e correlazione	236
14.3.3	Tabelle pivot	236
14.3.4	Tabella trial	237
14.3.5	Stratificate	238
14.3.5.1	Splitting	239
14.3.5.2	Convertire indici di grouping in variabili	240
14.3.5.3	Memorizzare in un dict lo split	241
14.3.5.4	Iterazione sui gruppi	241
14.3.5.5	Scelta delle variabili di analisi	243
14.3.5.6	Applicare funzioni di aggregazione custom	243
14.3.5.7	Elaborazioni custom	243
14.3.5.8	Trasformazioni group-wise	244
15	Probabilità e simulazione	247
15.1	Combinatoria	247
15.2	Generazione di numeri casuali con <code>numpy.random</code>	248
15.3	Variabili casuali in <code>scipy.stats</code>	249
15.3.1	Funzioni e parametri principali	249
15.3.2	Uso interattivo rapido	250
15.3.3	Freezing di una distribuzione	250
15.3.4	Uso del generatore di <code>numpy</code>	251
15.4	Altri argomenti	251
15.4.1	Bootstrap CI	251
16	Test statistici	253
16.1	Setup	254
16.2	Medie	254
16.2.1	Test t: 1 gruppo vs valore teorico	254
16.2.2	Test t: 2 gruppi indipendenti	254
16.2.3	Anova (2+ gruppi indipendenti)	254
16.2.4	Test t: 2 gruppi appaiati	256
16.2.5	Anova per misure ripetute (2+ gruppi appaiati)	256
16.3	Non parametric	257
16.3.1	Wilcoxon	257
16.3.2	Mann Whitney	257
16.3.3	Kruskal Wallis	257
16.3.4	Friedman test	258
16.4	Proporzioni	258
16.4.1	Test binomiale e CI clopper pearson	258
16.4.2	Test di Fisher	258
16.4.3	Chisquare	259
16.4.4	McNemar	259
16.4.5	Q di Cochran	259
16.5	Tassi	260

16.5.1	Comparazione 2 tassi	260
16.6	Correlazione	260
16.6.1	Pearson	260
16.6.2	Spearman	261
16.6.3	Tests	261
16.7	Varianze	261
16.7.1	Test di Bartlett	261
16.7.2	Test di Levene	261
16.7.3	Test di Fligner	261
16.8	Sopravvivenza	261
16.8.1	Logrank test	262
16.9	Agreement	262
16.9.1	Cohen's K	262
16.9.2	Fleiss K	262
16.9.3	Lin coefficient	262
16.10	Reliability/consistency	263
16.10.1	Cronbach α	263
16.10.2	ICC	263
16.11	Multiplicity	263
16.12	Test simulativi	263
17	Integrazione con R	265
17.1	Interscambio dataset	265
17.1.1	Da R a Python	265
17.1.2	Da Python a R	266
17.2	Chiamare R da Python: <code>rpy2</code>	266
17.2.1	Importazione di pacchetti	267
17.2.2	Ottenimento di dati con <code>rpy2</code>	267
17.2.3	Valutare stringhe di R	268
17.2.4	Creazione di vettori	268
17.2.5	Conversione <code>DataFrame</code> a R	268
17.2.6	Utilizzo di funzioni	268
18	Misc cookbook	271
18.1	Validazione <code>DataFrame</code> con <code>pandera</code>	271
18.2	Logging	273
18.3	SymPy	273
18.4	Ottenere codice di oggetti	274
18.5	Esecuzione parallela	274
18.6	File di configurazione	275
18.7	Calendario	276
18.8	Tcl/Tk	277
18.9	Telegram	277

Parte I

Linguaggio

Capitolo 1

Introduzione

Contents

1.1	Introduzione	15
1.1.1	Caratteristiche del linguaggio	15
1.1.2	Setup	16
1.2	Esecuzione	16
1.3	Ottenere aiuto	18
1.4	Gestione sistema	18
1.4.1	Aggiornamento di sistema periodico	18
1.4.2	Formati pacchetto	19
1.4.3	Pacchetti	19
1.4.4	Virtual environments	19
1.5	ipython	20
1.5.1	Configurazione	20
1.5.2	Magic commands utili	21
1.5.3	Comandi di shell	21

1.1 Introduzione

1.1.1 Caratteristiche del linguaggio

Il python è un linguaggio OOP di alto livello:

- l'indentazione determina il parsing del codice;
- si utilizza # per commento;
- per spezzare una istruzione su più righe si usa \ al termine della prima;
- ; al termine di una istruzione non serve (a meno che non si vogliano porre due istruzioni sulla stessa linea per separarle);

1.1.2 Setup

Pacchetti:

```
apt install python3 python3-pip python3-virtualenv python-is-python3 \
    texlive-extra-utils
```

1.2 Esecuzione

Vi sono due modi per utilizzare l'interprete classico:

- in via batch, alternativamente:
 - creando un file con estensione `.py`, che contenga istruzioni python valide a seconda della versione, ed eseguendolo attraverso `python file.py`
 - creando un file senza estensione con le seguenti sha-bang, dargli i permessi di esecuzione e porlo nel path degli eseguibili


```
#!/usr/bin/env python
#!/usr/bin/env python3
```
- in modalità interattiva:
 - entrando nell'interprete mediante `python` o `python3`
 - editando un file `.py` in Emacs, questi va in `python-mode`. Far partire un processo python in un buffer con `C-c C-p` e passando all'interprete i comandi descritti in tabella 1.1.
Se si desidera utilizzare `ipython` come interprete, porre quanto segue in `.emacs`

```
(require 'python)
(setq python-shell-interpreter "ipython")
(setq python-shell-interpreter-args "--simple-prompt")
```


Sequenza	Comando	Descrizione
C-c C-r	<code>python-shell-send-region</code>	invia la regione selezionata
C-c C-e	<code>python-shell-send-statement</code>	manda la regione selezionata o lo statement della linea
C-c C-s	<code>python-shell-send-string</code>	invia una riga da specificare
C-c C-c	<code>python-shell-send-buffer</code>	invia tutto il buffer corrente
C-c C-l	<code>python-shell-send-file</code>	tipo <code>source</code> di bash
C-c C-d	<code>python-describe-at-point</code>	descrivi la cosa
C-c C-v	<code>python-check</code>	usa qualche tester da impostare con <code>python-check-command</code>
C-c C-t c	<code>python-skeleton-class</code>	introduci una classe da template
C-c C-t d	<code>python-skeleton-def</code>	introduci una funzione da template
C-c C-t f	<code>python-skeleton-for</code>	introduci un for da template
C-c C-t i	<code>python-skeleton-if</code>	introduci un if da template
C-c C-t m	<code>python-skeleton-import</code>	introduci un import da template
C-c C-t t	<code>python-skeleton-try</code>	introduci un try da template
C-c C-t w	<code>python-skeleton-while</code>	introduci un while da template
C-c <	<code>python-indent-shift-left</code>	indenta a sinistra
C-c >	<code>python-indent-shift-right</code>	indenta a destra

Tabella 1.1: Comandi `python-mode` di emacs

1.3 Ottenere aiuto

`help` fa uso di docstring e si usa alternativamente come

```
help()          # help di sistema (moduli, keyword, simboli, topics)
help(oggetto)  # help specifico di un oggetto - docstring
```

Si può accedere alla documentazione anche dalla shell mediante `pydoc`

```
pydoc nome  # equivale a help(nome) dall'interprete
```

Infine in `ipython` il punto di domanda `?` svolge funzione simile

```
In [13]: ?len
Signature: len(obj, /)
Docstring: Return the number of items in a container.
Type:      builtin_function_or_method
```

È possibile usare wildcards, ad esempio

```
In [22]: ?*Warning
BytesWarning
DeprecationWarning
FutureWarning
...
```

Nel caso si usi `??` viene stampata ancora più informazione e nel caso di codice, viene riportato

```
In [15]: def fun():
...:     pass
...:
In [16]: ??fun
Signature: fun()
Docstring: <no docstring>
Source:
def fun():
    pass
File:      ~/.sintesi/cs/<ipython-input-15-a86dfd40a7ff>
Type:      function
```

1.4 Gestione sistema

In questa parte come installare e disinstallare pacchetti dal sistema, creando installazioni tra loro indipendenti mediante i virtual environments. La reference principale è questa: <https://packaging.python.org/en/latest/tutorials/installing-packages/>

1.4.1 Aggiornamento di sistema periodico

```
python3 -m pip install --upgrade pip setuptools wheel
```

1.4.2 Formati pacchetto

Il python ha due formati di pacchetto, il sorgente Source Distribution (`sdist`, che sono poi `.tar.gz`) e il formato binario `wheel` (estensione `.whl`, preferito da `pip` se disponibile perché più veloce).

1.4.3 Pacchetti

Per l'installazione di pacchetti da PyPI (Python Package Index) serve il tool `pip`, in Debian disponibile mediante `python3-pip`. Di default viene applicata l'installazione di sistema, a meno che:

- non si stiano utilizzando virtual environment;
- non si stia aggiungendo il parametro `--user`: questo fa sì che avvenga in `.local/lib/pythonX.X`

```
## Lista/mostra
pip list -vvv      # pacchetti installati e dove sono
pip freeze         # pacchetti installati (formato requirements)
pip show sphinx    # mostra info su pacchetto installato

## Installazione
pip install project_name                # ultima versione
pip install project_name==1.4           # determinata versione
pip install -r requirements.txt          # le dipendenze di un pacchetto
pip install ./myproject/path            # da repo locale
pip install "git+https://github.com/lbraglia/pymimo.git" # da github
pip install "git+https://github.com/lbraglia/pymimo.git@refs/pull/123/head" # da github,
                                                    # un pull request

## Aggiornamento pacchetto (non ancora disponibile aggiorna tutti)
pip install --upgrade project_name

## Disinstallazione pacchetto
pip uninstall project_name
```

1.4.4 Virtual environments

I virtual environments fanno sì che i pacchetti Python, ma anche il singolo interprete, possano essere installati in una locazione isolata per una data applicazione, invece di essere installati globalmente. Si usa il pacchetto `venv` e tipicamente la directory dove si installano questi environments è `.venv`

Creazione Per la creazione di un `venv` si comanda

```
l@m740n:~$ cd /tmp/
l@m740n:/tmp$ python -m venv venv_test
l@m740n:/tmp$ ls -l venv_test/
totale 20
drwxr----- 2 1 1 4096 12 apr 09.32 bin
drwxr----- 2 1 1 4096 12 apr 09.32 include
drwxr----- 3 1 1 4096 12 apr 09.32 lib
```

```
lrwxrwxrwx 1 1 1      3 12 apr 09.32 lib64 -> lib
-rw-r----- 1 1 1     69 12 apr 09.32 pyvenv.cfg
drwxr----- 3 1 1 4096 12 apr 09.32 share
```

Viene creata la cartella `venv_test` che contiene eseguibili di Python e `pip` (in `bin`) e librerie (in `lib/pythonX.Y/site-packages`, inizialmente vuota).

Utilizzo Il virtual environment va attivato, per fare sì che si usi esso e non il sistema complessivo per installazioni/disinstallazioni. Questo si fa mediante

```
l@m740n:/tmp$ source venv_test/bin/activate
```

Quello che avviene è che cambia il prompt per indicarci che tutto è avvenuto correttamente;

```
(venv_test) l@m740n:/tmp$
```

si può dunque iniziare ad installare roba senza compromettere il sistema

```
(venv_test) l@m740n:/tmp$ pip install codicefiscale
```

Per uscire dal virtual environment usare `deactivate`, che ci riporta ad utilizzare l'installazione di sistema

```
(venv_test) l@m740n:/tmp$ deactivate
```

Per eliminare il virtual environment, semplicemente cancellare la directory

1.5 ipython

Per l'installazione

```
pip install --user ipython
```

Per l'avvio `ipython` e per avere una guida rapida ai comandi

```
%quickref
```

1.5.1 Configurazione

Il file di configurazione bianco viene creato mediante

```
ipython profile create [profilename]
```

Se non è specificato un `profilename` viene creato il default e il file di nostro interesse è `~/ipython/profile_default/ipython_config.py`. In questo file si settano parametri di configurazione dell'oggetto `c`. Ad esempio alcune configurazioni utili da decommentare/modificare

```
# pdb di default
c.InteractiveShell.pdb = True
# non chiedere conferma se si esce con Ctrl+D
c.TerminalInteractiveShell.confirm_exit = False
```

Per altre vedere il file e la documentazione qui.

Creare e utilizzare profili specifici

```
ipython profile create secret_project
# edit
$ ipython --profile=secret_project
```

1.5.2 Magic commands utili**Lista dei magic command**

```
%lsmagic # compatta
%magic    # verbosa
```

Ottenere help dei magic command Si usa l'help

```
?%run
```

Esecuzione di uno script Per eseguire uno script in un namespace vuoto si usa:

```
%run path/script.py
```

Il comportamento dovrebbe essere lo stesso di `python script.py` da linea di comando.

Viceversa se si desidera importare codice da uno script nella sessione seguente

```
%load path/script.py
```

1.5.3 Comandi di shell

Preponendo un `!` ad un comando shell, ipython lo esegue in una sottoshell

```
!ls
```

Di bello c'è che si possono passare dati da e verso la shell

```
In [27]: dir = !pwd
```

```
In [28]: print(dir)
['/home/l/cs']
```

```
In [9]: message = "hello from Python"
```

```
In [10]: !echo {message}
hello from Python
```

Magic command da shell I comandi con `!` sono eseguiti in una sottoshell temporanea. Questo fa sì che se si vuole cambiare directory di lavoro cose come `!cd` non funzionino. Per eseguire comandi di shell usare il simbolo percentuale in `%cd`, `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm`, and `%rmdir`.

Se poi si comanda

```
In [33]: %automagic on
```

```
Automagic is ON, % prefix IS NOT needed for line magics.
```

è possibile evitare di apporre % davanti ai comandi, rendendo interfacciarsi con shell e python più seamless.

Capitolo 2

Dati

Contents

2.1	Introduzione	23
2.2	Tipi nativi	24
2.3	Classificazione dei tipi di base	25
2.4	Numeri	26
2.5	Date e ore	27
2.6	Sequenze: stringhe, liste e tuple	28
2.6.1	Operatore di slice (selezione da sequenza) e indici . .	28
2.6.2	Stringhe	30
2.6.3	Liste	34
2.6.4	Tuple	38
2.6.5	Sequence unpacking	38
2.7	Classi mapping e set	39
2.7.1	Dict	39
2.7.2	Sets	40
2.8	Altri tipi utili	42
2.8.1	namedtuple	42
2.8.2	Counter	42
2.9	Type annotation	43
2.9.1	Sintassi	43
2.9.2	Checking	44
2.9.3	Tipi utilizzabili per variabili	44
2.9.4	Creazione di alias	45
2.9.5	Annotazione di funzioni	46
2.9.6	Annotazione di metodi in classi	46

2.1 Introduzione

Assegnazione Gli oggetti del linguaggio vengono creati mediante l'**assegnazione**, che avviene attraverso l'uso di `=`, e non vi è necessità di dichiarare precedentemente il tipo della variabile (dinamically typed):

```
message = 'Hi friend'
pi = 3.1415926535897932
```

Keyword linguaggio I nomi non utilizzabili come identificatori sono:

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	in	raise	nonlocal	
continue	finally	is	return	
def	for	lambda	try	

Eliminazione oggetti La rimozione del binding ad aree di memoria (pre intervento del garbage collector) avviene mediante la keyword `del`

```
a = 1
del a
```

Operazioni su oggetti Di ogni oggetto è possibile:

- conoscere la **classe di appartenenza** mediante `type` o `isinstance`

```
>>> type(1)
<class 'int'>
>>> isinstance(1, int)
True
```

- **listare dati e metodi disponibili** (ai quali si accede mediante l'operatore punto), derivanti dalla classe di appartenenza mediante `dir`

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
```

- ottenere un **identificatore univoco** (della singola istanziazione), ottenuto mediante la funzione `id` applicata all'oggetto (questa non restituisce altro che l'indirizzo in memoria dell'oggetto)

```
>>> a = 1
>>> id(a)
10734128
```

2.2 Tipi nativi

I tipi più di base che il python mette a disposizione sono:

- **bool**: variabili booleane come `True` o `False`
- **int**: gli interi
- **float**: numeri con virgola mobile
- **str**: stringhe di caratteri unicode (non modificabili) compresi tra virgolette
- **bytes**: per la manipolazione di binario

A partire da questi tipi di base si possono creare oggetti composti tra i quali:

- **set**: insiemi di elementi non ordinati
- **list** sono sequenze ordinate e modificabili di elementi
- **tuple** sono sequenze ordinate e non modificabili di elementi
- **dict**: sono array coppie chiave - valore

Infine altri tipi builtin (che servono soprattutto nell'ottica della programmazione) sono:

- **type**: la classe di un oggetto è essa stessa un oggetto di classe **type**
- **None**: serve per indicare un valore vuoto ed ha classe **NoneType**. Non presenta attributi.
- funzioni
- classi
- moduli

Il nome del singolo tipo (ad esempio **int**) serve generalmente, se utilizzato come funzione¹, per **coercire da un tipo all'altro**:

```
>>> int(1.1)
1
>>> float(1)
1.0
>>> str(123)
'123'
```

2.3 Classificazione dei tipi di base

I tipi di base possono essere classificati a seconda di:

- **storage** model: quanti oggetti base possono essere contenuti in un oggetto? in base a questo distinguiamo
 - *oggetti scalari*: contengono un singolo oggetto base
 - *oggetti container*: contengono molteplici oggetti singoli. Il fatto che contengano molteplici oggetti pone poi che questi debbano essere o meno della stessa tipologia. In python tutti i tipi container base possono esser formati da oggetti base di tipo diverso.
- **update** model: una volta creato l'oggetto può esser modificato? Distinguiamo oggetti *mutabili* e *immutabili*
- **access** model: come si accede ad un singolo elemento facente parte dell'oggetto? distinguiamo

¹Questo perchè il nome di una classe usato come funzione, serve come costruttore.

Data type	Storage	Update	Access
Numbers	Scalar	Immutable	Direct
Strings	Scalar	Immutable	Sequence
Lists	Container	Mutable	Sequence
Tuples	Container	Immutable	Sequence
Dictionaries	Container	Mutable	Mapping

Tabella 2.1: Classificazione dei tipi

- accesso *direct*: caratteristico di alcuni oggetti atomici per i quali non si pone problemi di accesso particolare
- accesso *sequence*: accesso mediante indice numerico
- accesso *mapping*: accesso mediante key alfanumerica

Tabella 2.1 sintetizza la classificazione seguendo i criteri presentati.

2.4 Numeri

Vi sono tre tipi numerici: interi, floating point e complessi; i booleani sono un sottotipo di intero. Tutti i tipi numerici ad eccezione dei complessi supportano le seguenti operazioni, le quali hanno maggior priorità che gli operatori di comparazione

```

x + y    somma
x - y    differenza
x * y    prodotto
x / y    quoziente
x // y   parte intera della divisione
x % y    resto della divisione
divmod(x, y) la coppia (x // y, x % y)
x ** y   elevamento a potenza
pow(x, y) elevamento a potenza
abs(x)   valore assoluto
int(x)   x convertito a intero
float(x) x convertito a virgola mobile
complex(re, im) numero complesso con re parte reale e im immaginaria
c.conjugate() conjugate of the complex number c

```

int e float supportano

```

math.trunc(x)    parte intera
round(x[, n])    x arrotondato a n digits (se omissso default a 0)
math.floor(x)    maggiore intero <= x
math.ceil(x)     minor intero >= x

```

Infine l'unico metodo che sembra veramente utile per i float è `is_integer` (gli int non hanno metodi di interesse soprattutto in ambito reale)

```

>>> f1 = 2.0
>>> f1.is_integer()

```

True

```
>>> f2 = 1.2
>>> f2.is_integer()
False
```

Per operazioni aggiungive vedere i moduli `math` e `cmath`

2.5 Date e ore

il modulo `datetime` mette a disposizione le classi di base per date, ore, dateore etc

```
>>> import datetime as dt

>>> # definire date/ore
>>> birth = dt.date(1983, 11, 4) # year, month, day
>>> birth_time = dt.time(15, 50, 37) # hour, minute, second
>>> birth_dt = dt.datetime(1983, 11, 4, 15, 50, 37) # tutto

>>> # parsing/importazione da stringhe
>>> dt.datetime.strptime("11/11/2011", "%d/%m/%Y") # custom
datetime.datetime(2011, 11, 11, 0, 0)
>>> dt.datetime.strptime("11/11/2011 12:12:12", "%d/%m/%Y %H:%M:%S") # custom
datetime.datetime(2011, 11, 11, 12, 12, 12)
>>> dt.date.fromisoformat('2019-12-04') # iso
datetime.date(2019, 12, 4)
>>> dt.datetime.fromisoformat('2011-11-04T00:05:23') # iso
datetime.datetime(2011, 11, 4, 0, 5, 23)

>>> # oggi/ora
>>> oggi = dt.date.today() # oggi: data
>>> ora = dt.datetime.now() # adesso: datetime

>>> # accesso a singoli elementi
>>> [birth.year, birth.month, birth.day]
[1983, 11, 4]
>>> [birth_time.hour, birth_time.minute, birth_time.second]
[15, 50, 37]
>>> [birth_dt.year, birth_dt.month, birth_dt.day,
...  birth_dt.hour, birth_dt.minute, birth_dt.second]
[1983, 11, 4, 15, 50, 37]

>>> # convertire datetime in date
>>> ora.date()
datetime.date(2025, 5, 11)

>>> # differenze di date
>>> diff = oggi - birth
>>> diff
datetime.timedelta(days=15164)
```

```

>>> diff.days / 365.25
41.516769336071185

>>> # differenza di tempi
>>> a = dt.datetime(2000, 11, 11, 2, 50, 30)
>>> b = dt.datetime(2001, 11, 13, 2, 50, 0)
>>> diff_t = b - a
>>> diff_t
datetime.timedelta(days=366, seconds=86370)
>>> diff_t.total_seconds() # secondi di differenza totali
31708770.0

>>> # far scorrere il tempo
>>> un_giorno = dt.timedelta(days = 1)
>>> ieri = oggi - un_giorno
>>> ieri
datetime.date(2025, 5, 10)
>>> domani = oggi + un_giorno
>>> domani
datetime.date(2025, 5, 12)

>>> # esportazione/formattazione a stringa
>>> ora.strftime("%d-%m-%Y") # custom
'11-05-2025'
>>> ora.strftime("%d-%m-%Y - %H:%M:%S") # custom
'11-05-2025 - 10:46:39'
>>> oggi.isoformat() # iso
'2025-05-11'
>>> ora.isoformat() # iso
'2025-05-11T10:46:39.941631'

```

2.6 Sequenze: stringhe, liste e tuple

Introduciamo prima gli operatori e le funzioni builtin che funzionano con tutte le sequenze per affrontare le peculiarità di ognuna in sezione separata.

Gli operatori presentati in tabella 2.2 si applicano a tutte le sequenze. Altre funzioni di utilità per tutte le sequenze² sono quelle di tabella 2.3 Agli iterabili (di cui le sequenze fanno parte) si possono applicare le funzioni di tabella 2.4 per coercirli a sequenze.

2.6.1 Operatore di slice (selezione da sequenza) e indici

Le parentesi `[]` (operatore di slice) servono per effettuare estrazione da una sequenza. Le sequenze hanno indici che, alternativamente

- vanno da 0 (indice del primo elemento) a `n-1` dove `n` è la lunghezza della sequenza, restituita da `len` (analogamente a quanto avviene nel C).
- vanno da `-n` a `-1` per riferirsi agli oggetti dal primo all'ultimo con indici negativi

²A parte `len`, `reversed` e `sum` queste si applicano agli iteratori in genere

Operatore	Funzione
<code>seq[::]</code>	accedi ad elementi specifici di <code>seq</code>
<code>seq[ind1:ind2]</code>	da <code>ind1</code> incluso sino a <code>ind2</code> escluso
<code>seq[ind1:ind2:ind3]</code>	da <code>ind1</code> incluso a <code>ind2</code> escluso, facendo passi di <code>ind3</code>
<code>seq * expr</code>	ripeti la sequenza <code>expr</code> volte
<code>seq1 + seq2</code>	concatena le due sequenze
<code>obj in seq</code>	testa se l'oggetto <code>obj</code> è presente nella sequenza <code>seq</code>
<code>obj not in seq</code>	contrario del precedente test

Tabella 2.2: Operatori comuni per le sequenze

Funzione	Attività
<code>enumerate(iter)</code>	ritorna un oggetto enumerato
<code>len(seq)</code>	ritorna la lunghezza della sequenza
<code>max(iter)</code>	ritorna il massimo dell'iterabile
<code>min(iter)</code>	ritorna il minimo dell'iterabile
<code>reversed(seq)</code>	ritorna un iteratore che attraversa la sequenza in ordine inverso
<code>sorted(iter)</code>	ritorna una lista ordinata dell'iterabile fornito
<code>sum(seq)</code>	somma della sequenza

Tabella 2.3: Operatori per la coercizione a sequenze

Funzione	Attività
<code>list(iter)</code>	converti l'iterabile ad una lista
<code>str(iter)</code>	converti l'iterabile ad una stringa
<code>tuple(iter)</code>	converti l'iterabile ad una tuple

Tabella 2.4: Operatori per la coercizione a sequenze

```

len = 6
+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+
  0   1   2   3   4   5
-6  -5  -4  -3  -2  -1

```

La sintassi dello slicing prevede al massimo tre indici

```
seq[partenza:termine:passo]
```

- indice di partenza: specifica da che elemento partire, se mancante (o `None`) si intende di partire dall'inizio della sequenza
- indice di termine: specifica sino a quale elemento (escluso) terminare con l'estrazione. Se mancante (o `None`) si intende il termine della sequenza.
- indice di passo: specifica lo step dell'estrazione: se mancante lo step è di singola unità

2.6.2 Stringhe

La creazione avviene normalmente mediante assegnazione; la stringa può essere ugualmente inclusa tra apici singoli o doppi. Vediamo alcuni metodi comuni

Stringhe valide

```

>>> stringa = "asdomar"
>>> stringa_multilinea = """
... bla bla bla
... qua qua qua
... za za za
... """
>>> stringa_multilinea2 = (
...     "Queste stringhe tra parentesi, senza virgola, "
...     "saranno concatenate "
...     "come avviene nel C, "
...     "sono utili per spezzare."
... )
>>> print(stringa_multilinea2)
Queste stringhe tra parentesi, senza virgola, saranno concatenate come avviene nel C,

```

Formattazione ed f-strings

```

>>> import datetime

>>> s = "test"
>>> n = 13
>>> p = 2/3
>>> a_mill = 1000000
>>> dt = datetime.datetime.now()
>>> the_format = ".3f"

```

```

>>> address = "Via Tal dei tali"
>>> name = "Mario"

>>> class MyClass:
...     def __format__(self, format_spec) -> str:
...         return "object"
...
>>> class MyClass2:
...     def __format__(self, format_spec) -> str:
...         match format_spec:
...             case 'upper':
...                 return "OBJECT"
...             case 'lower':
...                 return "object"
...             case _:
...                 raise ValueError(f"{format_spec} not accepted")
...
>>> obj1 = MyClass()
>>> obj2 = MyClass2()

>>> # = per debugging: stampa a sinistra una espressione e a destra il risultato
>>> f"{s = }, {n = }"
"s = 'test', n = 13"
>>> f"{n % 2 = }" # qualsiasi espressione a sinistra viene valutata
'n % 2 = 1'
>>> f"{isinstance(n, int) = }" # e il rispettivo valore restituito
'isinstance(n, int) = True'

>>> # : per formattazione. Quello messo a dx di : dipende dal tipo
>>> f"{s} {n}" # standard..
'test 13'
>>> f"{p:.2f}" # float con numero decimali
'0.67'
>>> f"{p:.0f}" # rimuovere decimali da un float
'1'
>>> f"{p:{the_format}}" # formato nested
'0.667'
>>> f"{p:.2%}" # percentuali con due decimali
'66.67%'
>>> f"{p:e}" # notazione scientifica
'6.666667e-01'
>>> f"{a_mill:,}" # separatore di migliaia
'1,000,000'
>>> f"{a_mill:,.2f}" # separatore di migliaia e specifica dec
'1,000,000.00'

>>> f"{dt:%d/%m/%Y}" # date
'11/05/2025'
>>> f"{dt:%Y-%m-%d (%H:%M:%S)}" # datetime
'2025-05-11 (10:46:39)'
>>> f"today is a {dt:%A of %B}" # giorno della settimana e mese

```

```

'today is a Sunday of May'

>>> multiline = (
...     f"Hi, this is a trick "
...     f"to split lines long {a_mill} words"
...     f"is something manageable."
... )
>>> multiline
'Hi, this is a trick to split lines long 1000000 wordsis something manageable.'

>>> f"{obj1}"          # classe con formato
'object'
>>> f"{obj2:upper}"     # classe con formato e parametro
'OBJECT'
>>> f"{obj2:lower}"     #
'object'

>>> # padding/alignment di stringhe e numeri
>>> f"{s:8}"           # allineato a sinistra in campo da 8
'test      '
>>> f"{s:>8}"          # allineato a destra
'      test'
>>> f"{s:*<8}"         # allineato a sx riempiendo con *
'test*****'
>>> f"{s:_^8}"         # centrato riempiendo con _
'__test__'
>>> f"{s:_^{n]}"       # uso di una variabile per n. di padding
'___test____'
>>> f'{address:20}{name:10}' # tabulating without libs
'Via Tal dei tali      Mario      '
>>> f"{n:06}"          # aggiungere zeri prima di un numero
'000013'

>>> # numeric base conversion
>>> f"{n:b}"           # binary
'1101'
>>> f"{3:010b}"        # binary with padding
'0000000011'
>>> f"{n:o}"           # octal
'15'
>>> f"{n:x}"           # hexadecimal
'd'
>>> f"{n:X}"           # hexadecimal uppercase
'D'

```

Altri metodi utili

```

>>> # Formattazione
>>> chi = "Luca"
>>> quanti = 25
>>> pi = 3.14159265359

```



```

>>> f"{chi} ha {quanti} anni"
'Luca ha 25 anni'
>>> f"pi è {pi:.4f} ecc"
'pi è 3.1416 ecc'
>>> "{0} ha {1} anni".format(chi, quanti)
'Luca ha 25 anni'
>>> "{0[1]}".format(['x','y'])
'y'
>>> "{0:.2f}".format(3)
'3.00'

>>> # Altra formattazione
>>> "123".zfill(4)      # Aggiunta di 0 iniziali
'0123'
>>> "   test".lstrip() # Rimozione spazi iniziali/finali
'test'
>>> "test   ".rstrip()
'test'

>>> # Join di iterabili
>>> "".join(["a", "b", "c"])
'abc'

>>> # Checks
>>> "PROVA".islower()      # Lower/upper case
False
>>> "PROVA".isupper()
True
>>> "BRGLCU83S04H223C".isalnum() # alfanumerico/alfabetico
True
>>> "BRGLCU83S04H223C".isalpha()
False
>>> "prova".startswith("f")    # inizia/termina con
False
>>> "prova".endswith("a")
True

>>> # Coercizioni utili
>>> "PROVA".lower()
'prova'
>>> "prova".upper()
'PROVA'
>>> "uno due tre prova".capitalize()
'Uno due tre prova'
>>> "uno due tre prova".title()
'Uno Due Tre Prova'

>>> # Ricerca/rimpiazzo
>>> "aiuola".find("a")      # Ricerca da sx (prima occorrenza)
0
>>> "aiuola".find("u")

```

```

2
>>> "aiuola".find("x")
-1
>>> "aiuola".rfind("a")  #Ricerca da destra (ultima occorrenza)
5
>>> "prova".replace("a", "e")  # Rimpiazzo
'prove'

>>> # Splitting
>>> "amicici".partition("c")
('ami', 'c', 'ici')
>>> "amicici".rpartition("c")
('amici', 'c', 'i')
>>> "amicici".split("c")
['ami', 'i', 'i']
>>> "linea1\nlinea2".splitlines()
['linea1', 'linea2']

```

2.6.3 Liste

Le **liste** una sequenza di valori, non necessariamente dello stesso tipo, separati da virgole e racchiusi tra parentesi quadre. Le liste sono modificabili

2.6.3.1 Definizione

La definizione di una lista avviene come segue

```

>>> squares = [1, 2, 4, 9, 16, 25]
>>> letters = ['a', 'b', 'c', 'd']
>>> squares
[1, 2, 4, 9, 16, 25]
>>> letters
['a', 'b', 'c', 'd']

```

2.6.3.2 Manipolazione e modifica

Le liste si comportano similmente alle stringhe per ciò che riguarda il **subset**:

```

>>> squares[0]
1
>>> squares[-1]
25
>>> squares[-3:]
[9, 16, 25]

```

e operatori (concatenazione, moltiplicazione)

```

>>> squares + squares
[1, 2, 4, 9, 16, 25, 1, 2, 4, 9, 16, 25]
>>> squares*2
[1, 2, 4, 9, 16, 25, 1, 2, 4, 9, 16, 25]

```

A differenza delle stringhe sono modificabili:

```
>>> squares[0] = 0 # modifica di un valore
>>> squares
[0, 2, 4, 9, 16, 25]
>>> letters[1:2] = [] # eliminazione di valori
>>> letters
['a', 'c', 'd']
```

Le liste sono oggetti e hanno **metodi** per le operazioni più classiche. Si veda `help(list)`.

2.6.3.3 Metodi utili per le liste

```
>>> l = []
>>> a = ["a", "c", "d"]

>>> # Aggiunta
>>> l.append(1) # un elemento in coda
>>> a.insert(1, "b") # un elemento prima dell'indice specificato
>>> l.extend([1,2,3]) # elementi da un iterabile in coda

>>> # Rimozione
>>> a.remove("c") # rimuove la prima occorrenza di un elemento
>>> x = a.pop(1) # rimuove il valore all'indice specificato e lo ritorna
>>> l.clear() # rimuove tutti gli elementi

>>> # Ricerca like
>>> ["a", "x", "c", "x"].index("x") # indice di un elemento (prima occorrenza)
1
>>> [1,2,1,3].count(1) # conta le occorrenze di un dato elemento
2

>>> # Ordinamento
>>> x = ["w", "j", "a"]
>>> x.sort() # ordina
>>> x.reverse() # inverte
```

2.6.3.4 Liste nested

Le liste possono essere nested, e nel caso il subsetting necessita di una parentesi graffa per ogni livello:

```
>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c']
>>> c = [a, b, 3]
>>> c
[[1, 2, 3], ['a', 'b', 'c'], 3]
>>> c[0]
[1, 2, 3]
>>> c[1]
['a', 'b', 'c']
>>> c[2]
3
```

```
>>> c[1][2]
'c'
```

2.6.3.5 List comprehensions

Sono un trick del linguaggio per creare liste in maniera concisa.

Definizione La versione più generale è

```
newlist = [expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN
            if conditionN+1
            if conditionN+2
            ...
            if conditionN+M
            ]
```

con:

1. **expression** è una espressione contenente **item1..itemN**;
2. seguita da uno statement **for** (obbligatorio) che si riferisca ad un iterabile (volendo filtrato mediante **if**);
3. al quale seguono 0+ più ulteriori statement **for** con altrettanti iterabili (per ciclare su altro, eventualmente filtrando con **if**);
4. al quale seguono 0+ statement **if** (opzionalmente per selezionare gli elementi da porre nell'output).

Gli **iterable*** non necessariamente debbono essere della stessa lunghezza, perché sono iterate da sinistra a destra, non in parallelo: per ogni elemento in **iterable1**, viene fatto loop su **iterable2** e tutte le rimanenti a cascata

Esempi

```
>>> ## Esempio base
>>> doubled = [x * 2 for x in range(10)]
>>> doubled
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

>>> ## Esempio con if
>>> combs = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
>>> # che equivale a
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
... 
```

```
>>> X = [1, 2, 3]
>>> Y = [4, 5, 6]
>>> res = [[x, y, x*y]
...         for x in X if x > 1
...         for y in Y if y < 6
...         if y % x == 0]
```

Espressioni più complesse e trick utili a seguire

```
>>> vec = [-4, -2, 0, 2, 4]
```

```
>>> ## Selezionare gli elementi >= 0
>>> [x for x in vec if x >= 0]
[0, 2, 4]
```

```
>>> ## Applicare una funzione a tutti gli elementi di una lista
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
```

```
>>> ## utilizzo di più di un if: stampa dei numeri divisibili per 2 e per 5
>>> ## tra 0 e 100
>>> [y for y in range(100) if y % 2 == 0 if y % 5 == 0]
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
>>> ## Le espressioni possono essere molto generali;
>>> ## if .. else con list comprehension
>>> [">=0" if i>=0 else "<0" for i in vec]
['<0', '<0', '>=0', '>=0', '>=0']
```

```
>>> ## creiamo una lista composta da tuple
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

```
>>> ## esegui un metodo su ogni elemento
>>> fruit = ['banana', 'strawberry', 'passion fruit']
>>> [y.capitalize() for y in fruit]
['Banana', 'Strawberry', 'Passion fruit']
```

List comprehensions nested Espressione di una list comprehension può esser qualunque cosa, quindi anche una list comprehension. Questo può essere utile per creare liste di liste, che possono avere applicazione.

```
>>> ## Calcolo tabelline del 7 e dell'8 (nb:cicla per prima è la lista
>>> ## esterna, poi quella interna)
>>> nl = [[i*j for j in range(1, 11)] for i in range(7, 9)]
>>> nl
[[7, 14, 21, 28, 35, 42, 49, 56, 63, 70], [8, 16, 24, 32, 40, 48, 56, 64, 72, 80]]
```

```
>>> ## Trasposizione di matrice creata mediante lista di liste
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
```

```

...             [9, 10, 11, 12]
...         ]
>>> t = [[row[i] for row in matrix] for i in range(4)]
>>> t
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

2.6.4 Tuple

Le **tuple** sono insiemi di valori separati da virgole (buona norma porle tra parentesi tonde per chiarezza) e non modificabili una volta create

```

>>> ## Definizione
>>> atuple = ('robots', 77, 93, 'try')
>>> atuple
('robots', 77, 93, 'try')

>>> ## subset
>>> atuple[:3]
('robots', 77, 93)

>>> ## tuple nested
>>> a = (1, 2)
>>> b = ('x', 'y')
>>> c = (a, b)
>>> c
((1, 2), ('x', 'y'))

>>> ## modifica diretta? no: da errore
>>> atuple[1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> ## tuttavia è possibile creare tuple che contengono oggetti modificabili
>>> btuple = ([1,2], "abc")
>>> btuple[0][0] = 3
>>> btuple
([3, 2], 'abc')

>>> ## Metodi utili, uguali a quelli delle liste
>>> atuple.count(93)
1
>>> atuple.index("try")
3

```

2.6.5 Sequence unpacking

Consiste nell'assegnare gli elementi di una sequenza (posta come *rvalue*) a variabili separate (*lvalue*)

```

>>> ## Esempio con lista
>>> a, b, c = [1, 2, 'goodbye']

```

```
>>> ## Esempio con tupla
>>> t = (12345, 54321, 'hello!')
>>> t1, t2, t3 = t

>>> ## Esempio con stringa
>>> x, y, z = 'cia'
```

2.7 Classi mapping e set

2.7.1 Dict

Sono un insieme non ordinato di coppie `key:value`, con `key` univoche all'interno di un dict (e immutabili), utile per memorizzare un dato e ritornarlo attraverso la sua chiave.

```
>>> ## Definizione, accesso e modifica
>>> d = {'planet': 'earth', 'region': 'europe', 'prefix': 39}
>>> d # stampa complessiva
{'planet': 'earth', 'region': 'europe', 'prefix': 39}
>>> d['prefix'] # accesso in lettura ad un elemento
39
>>> d[1] # errore: non si usano indici numerici
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> d['prefix'] = 45 # modifica consentita
>>> d['foo'] = "bar" # nuovo inserimento consentito

>>> ## esempio con chiavi numeriche
>>> d2 = {1: "a", 2: "b"}
>>> d2[1]
'a'

>>> ## Ottenere le chiavi mediante il metodo keys
>>> d.keys() # ottenere le chiavi
dict_keys(['planet', 'region', 'prefix', 'foo'])
>>> 'foo' not in d.keys()
False

>>> # Per mappare una chiave a valori multipli usare liste o set
>>> d = {
...     'a' : [1, 2, 3],
...     'b' : [4, 5]
... }
>>> d['b'][0]
4

>>> # Metodi alternativi per la definizione di dict, uso della funzione omonima
>>> x = dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
>>> d = dict(sape = 4139, guido = 4127, jack = 4098)
```

2.7.1.1 Metodi utili

```
>>> d = {'planet':'earth', 'region':'europe', 'prefix':39}

>>> # ottiene un valore data la chiave
>>> d.get('planet')
'earth'

>>> # indici e valori come iterabili (direi)
>>> d.keys()
dict_keys(['planet', 'region', 'prefix'])
>>> d.values()
dict_values(['earth', 'europe', 39])

>>> # coppia indice, valore; usabile per unpacking
>>> d.items()
dict_items([('planet', 'earth'), ('region', 'europe'), ('prefix', 39)])

>>> # rimuove tutti gli item
>>> d.clear()
```

2.7.1.2 Dict comprehension

Hanno sintassi analoga a quella delle liste e possono tornare talvolta utili come forma di mapping

```
>>> pow2 = {x: x**2 for x in (2, 4, 6)}
>>> pow2[2]
4
```

2.7.2 Sets

Sono una collezione di elementi senza ordine e duplicati. Si usano tipicamente per:

- verificare l'appartenenza di un qualcosa ad un insieme (mediante `in`);
- per eliminare duplicati di altre strutture dati
- per effettuare operazioni insiemistiche

Si definiscono mediante parentesi graffe o la funzione `set`.

```
>>> basket = {'apple', 'orange', 'apple', 'pear'}
>>> basket ## elementi doppi vengono eliminati
{'apple', 'pear', 'orange'}

>>> ## definizione mediante set: dalla sequenza fornita sono eliminati i doppi
>>> a = set('abracadabra') ## stringa
>>> a
{'c', 'b', 'd', 'a', 'r'}
>>> a = set(['asd', 'foo', 'bar', 'asd']) ## lista
>>> a
```



```

{'bar', 'asd', 'foo'}
>>> a = set( ('asd', 'foo', 'bar', 'asd') )  ## tuple
>>> a
{'bar', 'asd', 'foo'}

>>> ## emptyset: si usa set, non due graffe (usate per dict vuoto)
>>> empty = set()

>>> ## Test appartenenza
>>> 'orange' in basket
True
>>> 'strawberry' in basket
False

>>> ## aggiunta, rimozione, azzeramento
>>> empty.add(1)
>>> empty.remove(1)
>>> empty
set()
>>> empty.add(1)
>>> empty.add(2)
>>> empty.clear()

```

2.7.2.1 Operatori/metodi utili

```

>>> ## Applicazione operatori numerici e logici
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'c', 'b', 'd', 'a', 'r'}
>>> b
{'c', 'm', 'l', 'z', 'a'}

>>> ## Operazioni insiemistiche
>>> a - b # lettere in a ma non in b
{'b', 'd', 'r'}
>>> a | b # lettere in a o in b
{'c', 'm', 'l', 'z', 'b', 'd', 'a', 'r'}
>>> a & b # lettere sia in a che in b
{'c', 'a'}
>>> a ^ b # xor: lettere in a o in b ma non in entrambi
{'l', 'm', 'b', 'd', 'z', 'r'}

>>> ## Test insiemistici
>>> # intersezione
>>> {1,2}.isdisjoint({3,4})
True
>>> {1,2}.isdisjoint({2,3})
False
>>> # sottinsieme e sovrainsieme
>>> a = {1, 2}

```

```
>>> b = {1}
>>> a.issuperset(b)
True
>>> b.issubset(a)
True
```

2.7.2.2 Set comprehension

Le comprehension sono ammesse anche nei set

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

2.8 Altri tipi utili

2.8.1 namedtuple

Espongono le tuple (immutabili) con l'uso di key dei dict per facilità di accesso

```
>>> from collections import namedtuple

>>> # Problem with tuples and dictionary
>>> # color = (55, 155, 255) # immutable but standard tuple has no names, less readable
>>> # color = {"red": 55, "green": 155, "blue": 255} # dict is mutable

>>> # the right mix
>>> Color = namedtuple("Color", ["red", "green", "blue"]) # create the object
>>> color1 = Color(blue = 55, green = 155, red = 255) # create an instance
>>> color2 = Color(255, 255, 255) # another instance

>>> # Accessing data two way (still read-only)
>>> color1
Color(red=255, green=155, blue=55)
>>> color1.red # no color1["red"]
255
>>> color1.red = 123 # error: tuples are still immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

2.8.2 Counter

Sono versioni speciali di dict che servono per contare frequenze di occorrenze

```
>>> from collections import Counter
>>> import random

>>> c = Counter() # create an empty counter

>>> # it's dict like, no initialization needed
```

```

>>> c["a"] += 1    # increase count of an element (starting from 0)
>>> c["a"] += 3    # increase count of an element (starting from 0)
>>> c["b"] = 2
>>> c["c"] = 1
>>> c["x"]          # no elements counted
0

>>> # typical methods
>>> sorted(c.elements()) # accesso a tutti gli elementi, con ripetizione
['a', 'a', 'a', 'a', 'b', 'b', 'c']
>>> c.total() # sum of all frequencies
7
>>> c.most_common(2) # most common modalities
[('a', 4), ('b', 2)]

>>> # # other are as in dict
>>> # c.keys()
>>> # c.values()
>>> # c.items()

>>> # example 2: 100d4
>>> d4 = [1, 2, 3, 4]
>>> c2 = Counter()
>>> for i in range(100):
...     roll = random.choice(d4)
...     c2[roll] += 1
...
>>> c2
Counter({3: 35, 1: 23, 4: 23, 2: 19})

>>> # example3: use a counter on elements of a list
>>> rolls = random.choices(d4, k = 200) # list of 200 d4 rolls
>>> c3 = Counter(rolls)
>>> c3
Counter({1: 56, 4: 53, 3: 47, 2: 44})

```

2.9 Type annotation

Qui sono presentati concetti che richiedono conoscenze più approfondite rispetto ad una prima lettura, nella quale si possono tralasciare.

Sebbene Python sia un dynamic language con variabili aventi un tipo non stabilito a priori/che può variare nel corso dell'esecuzione del programma, è possibile specificare il tipo assunto da una variabile nonché il tipo ritornato da una funzione in maniera tale da usare tool esterni che analizzino il codice e riportino utilizzi impropri.

2.9.1 Sintassi

```

>>> ## -----

```

```

>>> ## File test.py
>>> ## -----

>>> # formati per variabile: in unico colpo
>>> x: int = 4

>>> # spezzato
>>> x: int
>>> x = "4" # qua viene dato errore nei programmi giusti, non nell'interprete

>>> # esempio con funzioni, notare i parametri e il tipo restituito
>>> # z è un parametro opzionale (può essere int al massimo) di default a None
>>> def repeat(x: str, y: int = 2, z: int|None = None) -> None:
...     if z is None:
...         print(x * y)
...     else:
...         print(x * y * z)
...
>>> repeat(x = 'foo')
foofoo

```

2.9.2 Checking

La sintassi di cui sopra è tool indipendente quindi può essere potenzialmente gestita da diversi eseguibili. Qui utilizziamo mypy. Per installarlo

```
pip install --user mypy
```

Dopodiché per controllare il file di cui sopra

```
mypy test.py
```

Per controllare una pacchetto posizionarsi nella directory base (quella con requirements.txt e compagnia) e comandare

```
mypy .
```

2.9.3 Tipi utilizzabili per variabili

Tutti i tipi di base quindi str, int, float, bool, ma anche None per indicare che la funzione non ritorna nulla.

Per i tipi compositi: sotto alcuni esempi di assegnazione corretta

```

# lista di sole stringhe
x: list[str] = ["foo", "bar"]

# insieme di interi
x: set[int] = {6, 7}

# per dict fornire il tipo di key e value
x: dict[str, float] = {"field": 2.0}

# tuple di dimensione fissa si specifica il tipo di ogni elemento

```

```

x: tuple[int, str, float] = (3, "yes", 7.5)

# tuple di dimensione variabile: si usa un tipo e l'ellissi
x: tuple[int, ...] = (1, 2, 3)

# se ad esempio vogliamo essere generali
# e specificare un iterabile (lista, tuple, set, altro) di interi

from typing import Iterable
x: Iterable[str] = ...

# i tipi Mapping sono dict-like non mutabili (con metodo __getitem__)
from typing import Mapping
x: Mapping[int, str] = {3: 'yes', 4: 'no'}
x[5] = 'foo' # qui mypy si lamenta

# MutableMapping sono dict-like mutabili (con metodo __setitem__)
x: MutableMapping[int, str] = {3: 'yes', 4: 'no'}
x[5] = 'foo' # qui mypy non si lamenta

# accettare tipi molteplici
x: list[int | str] = [3, 5, "test", "fun"]
# questo è utile in funzioni per specificare parametri opzionali

# Optional per dati che possono essere anche None
from typing import Optional
x: Optional[str] = "something" if some_condition() else None

```

mypy conosce i tipi della standard library e fornisce suggerimenti sui pacchetti da installare nel caso non sia così

```

prog.py:2: error: Library stubs not installed for "requests"
prog.py:2: note: Hint: "python3 -m pip install types-requests"

```

2.9.4 Creazione di alias

La digitazione di tipi complessi più volte può essere evitata mediante la creazione di alias, fatti semplicemente mediante assegnazione. Ad esempio

```

Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# utile anche per rendere i tipi più leggibili/compatti/riutilizzabili
ConnectionOptions = dict[str, str]
Address = tuple[str, int]
Server = tuple[Address, ConnectionOptions]
def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

```

2.9.5 Annotazione di funzioni

Abbiamo già visto un esempio abbastanza generico di funzione

```
def show(value: str, excitement: int = 10) -> None:
    print(value + "!" * excitement)
```

Per la programmazione funzionale può essere necessario specificare un tipo Callable (funzione):

```
from typing import Callable

def repeat(x: str, y: int = 2) -> None:
    print(x * y)

# variabile: x può essere una funzione di tipo def xx(str, int): -> None
x: Callable[[str, int], None] = repeat
```

Per funzioni generatrici che restituiscono un iteratore di int si può fare così

```
def gen(n: int) -> Iterator[int]:
    i = 0
    while i < n:
        yield i
        i += 1
```

2.9.6 Annotazione di metodi in classi

self non va caratterizzato (nei parametri, se restituito si usa Self), poi spesso le funzioni di classi modificano dati internamente non restituendo nulla quindi si userà None come dato restituito

```
from typing import Self

class BankAccount:
    def __init__(self, account_name: str, initial_balance: int = 0) -> None:
        self.account_name = account_name
        self.balance = initial_balance

    def deposit(self, amount: int) -> None:
        self.balance += amount

    def withdraw(self, amount: int) -> None:
        self.balance -= amount

    def returnme(self) -> Self:
        return self
```

Le classi definite dall'utente sono tipi validi da usare per le annotazioni

```
account: BankAccount = BankAccount("Alice", 400)
def transfer(src: BankAccount, dst: BankAccount, amount: int) -> None:
    src.withdraw(amount)
    dst.deposit(amount)
```

Funzioni che accettano una classe, accetteranno anche classi derivate senza problemi:

```
class BankAccountForYoungs(BankAccount):  
    ....  
  
timmysba = BankAccountForYoungs("Timmy", 2)  
transfer(account, timmysba, 100) # type checks!
```


Capitolo 3

Controllo del flusso

Contents

3.1	Costrutti condizionali: if e while	49
3.1.1	if	49
3.1.2	match	50
3.1.3	while	50
3.1.4	break, continue ed else	50
3.2	Condizioni e test logici	50
3.2.1	Test verità	50
3.2.2	Operatori booleani	51
3.2.3	Comparazioni	51
3.2.4	Comparazioni concatenate	52
3.2.5	Check appartenenza: in e not in	52
3.2.6	Comparare sequenze e altri tipi	52
3.3	Looping su oggetti: for	53
3.3.1	Looping in sequenze (stringhe, liste, tuple)	53
3.3.2	Looping nei dict	54
3.3.3	Looping sui set	54
3.3.4	L'utilizzo di range	54

3.1 Costrutti condizionali: if e while

3.1.1 if

Ha la seguente sintassi con <> a indicare un contenuto obbligatorio, [] uno facoltativo e * la possibilità di ripetizione:

```
if <condizione>:
    <istruzioni>
[elif <condizione>:
    <istruzioni>]*
[else:
    <istruzioni>]
```

3.1.2 match

Simile allo switch di altri linguaggi, `match` prende una espressione e la compara ad una casistica di altre espressioni (poste dopo `case`) per eseguire azioni specificate:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 401 | 403 | 404:
            return "Not allowed"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

Si notano che si possono specificare match multipli con `|`, mentre `_` matcha sempre e quindi può essere usato come caso default (eventualmente) quando nessun altro caso ha matchato.

3.1.3 while

python ha solo l'istruzione `while` per implementare l'iterazione nel senso di altri linguaggi come il Pascal o il C:

```
while <condizione>:
    <istruzioni>
[else:
    <istruzioni>]
```

3.1.4 break, continue ed else

Sia in `while` che `for` (che vedremo poi):

- `continue` permette di saltare le rimanenti istruzioni del ciclo passando all'iterazione successiva;
- `break` termina il ciclo;
- se è presente la clausola `else` le sue istruzioni vengono eseguite qualora il ciclo termini normalmente, mentre non vengono eseguite se il ciclo termina a causa dell'istruzione `break`.

3.2 Condizioni e test logici

3.2.1 Test verità

Un oggetto può essere testato come `True/False` in un `if` o in un `while`: in generale un oggetto è considerato `True` a meno che, alternativamente

- la sua classe definisce il metodo `__bool__`, che restituisce `False`

- il metodo `__len__` ritorna qualcosa, se chiamato sull'oggetto

Quindi:

- `None` è valutato `False`
- i numeri restituiscono tutti `True` ad eccezione dello 0 (`int` o `float` che sia) che è `False`
- stringa, lista, tuple, set e dict **vuoti restituiscono False**, altrimenti (con almeno un elemento) viene restituito `True` (indipendentemente dal contenuto)

Mediante una funzioni del genere possiamo testare la valutazione booleana di oggetti di natura diversa:

```
>>> def is_it_true(anything):
...     if anything:
...         print("yes, it's true")
...     else:
...         print("no, it's false")
... 
```

3.2.2 Operatori booleani

Le comparazioni possono esser elaborate mediante operatori booleani `and` `or` e `not`.

È sempre meglio aggiungere le parentesi per indirizzare ordine/priorità di valutazione. In assenza tra gli operatori `not` ha la priorità maggiore, `or` la minore. Pertanto:

`A and not B or C`

equivale a

`(A and (not B)) or C`

3.2.3 Comparazioni

Vi sono otto operatori di comparazione, hanno tutti la stessa priorità (che è superiore a quella degli operatori booleani)

<code><</code>	minore
<code><=</code>	minore o uguale
<code>></code>	maggiore
<code>>=</code>	maggiore o uguale
<code>==</code>	equal
<code>!=</code>	not equal
<code>is</code>	due oggetti sono identici
<code>is not</code>	due oggetti sono diversi

Alcune regole:

- oggetti di tipo differente (esclusi numeri) son sempre diversi

- oggetti non identici di una stessa classe sono diversi, a meno che forniscano un metodo `__eq__` che li battezzi come uguali
- istanze di una classe non possono essere ordinate tra loro a meno che la classe non definisca `__lt__` ed `__eq__` (si può definire volendo `__lt__`, `__le__`, `__gt__`, `__ge__`)
- il funzionamento di `is` e `is not` non può esser modificato ed è supportato da iterabili o oggetti che implementano il metodo `__contains__`

3.2.4 Comparazioni concatenate

Le comparazioni possono essere concatenate, come in:

```
a < b == c
```

che equivale a in pratica

```
(a < b) and (b == c)
```

3.2.5 Check appartenenza: `in` e `not in`

`in` e `not in` controllano se un valore è presente o meno in una sequenza

3.2.6 Comparare sequenze e altri tipi

Oggetti di tipo sequenza possono esser comparati con oggetti del medesimo tipo; la comparazione avviene in maniera ricorsiva, con ordinamento lessicografico (in base a unicode). Vi sono alcune peculiarità:

- se tutti gli item di due sequenze sono uguali, le sequenze sono considerate uguali
- se una sequenza costituisce l'inizio di un'altra sequenza più lunga, la sequenza più corta è considerata minore

La comparazione restituisce un valore `True` o `False`; ad esempio nei seguenti casi viene restituito sempre `True`:

```
>>> (1, 2, 3) < (1, 2, 4)
True
>>> [1, 2, 3] < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2) < (1, 2, -1)
True
>>> (1, 2, 3) == (1.0, 2.0, 3.0)
True
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
True
```

3.3 Looping su oggetti: for

Nel python serve per iterare sugli elementi di una sequenza (come lista stringa o tuple) e presenta questa sintassi

```
for <variabile> in <sequenza>:
    <istruzioni>
[else:
    <istruzioni>]
```

Il funzionamento interno del `for` verrà spiegato nella sezione di OOP; qui si riassume l'utilizzo standard coi dati più comuni.

3.3.1 Looping in sequenze (stringhe, liste, tuple)

Nelle sequenze possiamo:

- fare loop sul singolo elemento (ponendo la sequenza nel `for` direttamente)
- ottenere un progressivo numerico indice e il contenuto della sequenza con `enumerate`
- fare loop sull'oggetto ordinato con `sorted`
- fare loop sull'oggetto ordinato in maniera decrescente con `reversed`

Gli esempi presentano liste, ma il funzionamento è speculare anche per stringhe e tuple:

```
>>> games = ['monopoli', 'risiko', 'dnd']
```

```
>>> for g in games:
...     print(g)
...
monopoli
risiko
dnd
>>> for i, g in enumerate(games):
...     print(i, g)
...
0 monopoli
1 risiko
2 dnd
>>> for g in sorted(games):
...     print(g)
...
dnd
monopoli
risiko
>>> for g in reversed(games):
...     print(g)
...
dnd
risiko
monopoli
```

3.3.2 Looping nei dict

Di un dict possiamo volere le chiavi (usiamo l'oggetto direttamente o ne chiamiamo/esplicitiamo il metodo `keys`), i contenuti (risp `values`) o tutti e due (`items`):

```
>>> knights = {"gallahad": "the pure", "ciro": "the brave"}

>>> for k in knights:
...     print(k)
...
gallahad
ciro
>>> for k in knights.keys():
...     print(k)
...
gallahad
ciro
>>> for v in knights.values():
...     print(v)
...
the pure
the brave
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
ciro the brave
```

3.3.3 Looping sui set

La forma più semplice, si pone il set nel for

```
>>> S = {2, 3, 5, 7}
>>> for i in S:
...     print(i)
...
2
3
5
7
```

3.3.4 L'utilizzo di range

Se è necessario iterare su una sequenza di interi la funzione `range` torna utile

```
>>> range(5)                # 0, 1, 2, 3, 4
range(0, 5)
>>> range(5, 10)           # 5, 6, 7, 8, 9
range(5, 10)
>>> range(0, 10, 3)         # 0, 3, 6, 9
range(0, 10, 3)
```

```
>>> range(-10, -100, -30)  # -10, -40, -70
range(-10, -100, -30)
```


Capitolo 4

Funzioni

Contents

4.1	Definizione	57
4.1.1	Argomenti	58
4.1.2	Stringa di documentazione	58
4.2	Chiamata	59
4.2.1	Valutazione dei valori di default	59
4.2.2	Valore ritornato	60
4.2.3	Liste/tuple/dict come parametri di chiamata	60
4.3	Regole di scope	61
4.3.1	Namespace	61
4.3.2	Attributi	61
4.3.3	Scoping	62
4.4	lambda e funzioni anonime	63
4.5	Programmazione funzionale	63
4.5.1	Funzioni classiche	64
4.5.2	Partialling	66
4.5.3	Function factory	67
4.5.4	Composizione di funzioni	67
4.5.5	Decorators	68
4.5.6	Single e multiple dispatch	71

4.1 Definizione

Avviene mediante `def`, seguita dal nome della funzione e ponendo tra tonde i parametri:

```
def nome_funzione(arg1, arg2 = default2, *args, **kwargs):  
    codice  
    ...
```

4.1.1 Argomenti

Gli argomenti devono essere specificati nell'ordine di cui sopra, e sono:

1. *positional argument* (**arg1**) son definiti con solamente il nome dell'argomento. In chiamata è obbligatorio specificarli inserendo alternativamente **valore** o **nome = valore** (in questo secondo caso sarà possibile seguire un ordine di argomenti diverso da quello dato in definizione);
2. *keyword argument* (**arg2**) son definiti con un valore di default: in sede di chiamata non sono obbligatori da specificare;
3. *arbitrary argument list*: se si specifica ***args** in definizione, la variabile **args** memorizzerà una tupla con gli argomenti posizionali della chiamata (specificati mediante il solo **valore**) esclusi i valori associati ad altri argomenti posizionali: ad esempio sopra prima viene riempito **arg1**, poi **args**);
4. *arbitrary keyword argument list*: se si specifica ****kwargs** in definizione, la variabile **kwargs** memorizzerà un dizionario con i keyword argument (**nome=valore**) specificati in sede di chiamata (esclusi quelli che matchano keyworded argument specificati, es **arg2**)

Un esempio

```
>>> def args_demo(arg1, *args, **kwargs):
...     print("arg1 = ", arg1)
...     print("args = ", args)
...     print("kwargs = ", kwargs)
... 
```

Parametri speciali /* per imporre la chiamata Gli argomenti possono esser passati alle funzioni sia per posizione che utilizzando la keyword. Si può (leggibilità/performance) in definizione imporre che la chiamata di alcuni possa avvenire in un modo, nell'altro o in entrambi, specificando gli argomenti opzionali / e *

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           Positional or keyword |
    |                                     +-- Keyword only
    +-- Positional only
```

4.1.2 Stringa di documentazione

Da porre come prima istruzione all'interno del corpo, come segue:

```
>>> def sq(n):
...     """
...     Returns the square of n.
...     """
...     return(n * n)
... 
```

mediante `sq.__doc__` si accede alla documentazione della funzione.

4.2 Chiamata

In chiamata si:

- specifica tutti gli argomenti per i quali in definizione non sono dati default;
- pone gli argomenti posizionali prima dei keyworded;
- i posizionali posti senza **nome** verranno associati nell'ordine dato in definizione; se si specificano tutti **nome = valore** l'ordine può differire dalla definizione

```
>>> args_demo(1, 2, 3, foo = 6, baz = 7)
arg1 = 1
args = (2, 3)
kwargs = {'foo': 6, 'baz': 7}
```

4.2.1 Valutazione dei valori di default

Questa:

- avviene al punto in cui la funzione è stata *definita*, non quando viene chiamata

```
>>> x = 5
```

```
>>> def f(arg = x):
...     print(arg)
...
>>> x = 6
>>> f()
5
```

- il valore valutato viene conservato per le future chiamate; se modificabile (lista, dict, classe) e modificato, la versione cambiata sarà conservata per le prossime chiamate (non sempre voluto)

```
>>> # modifica di valore di default imm modificabile è locale e non
>>> # conservato per le prossime chiamate
```

```
>>> def f(x = 1):
...     print(x)
...     x = 2
...
>>> f()
1
>>> f()
1
```

```
>>> # modifica di valore di default modificabile
>>> def f(x, L = []): # una lista è modificabile
...     L.append(x)
```

```

...     return L
...
>>> f(1)
[1]
>>> f(2)
[1, 2]
>>> f(3)
[1, 2, 3]

```

Se non si vuole che il valore di default sia condiviso tra successive chiamate scrivere una funzione come la seguente

```

def f(x, L = None):
    if L is None:
        L = []
    L.append(x)
    return L

```

4.2.2 Valore ritornato

Se:

- si utilizza l'istruzione `return` viene restituito dalla funzione alla chiamante un determinato dato fornito come argomento di `return`;
- se non si specifica nulla (o `return` senza argomenti) viene restituito `None`.

4.2.3 Liste/tuple/dict come parametri di chiamata

Per usarle aggiungere rispettivamente `*` (lista, tuple) e `**` (dict) prima del nome della variabile

```

>>> def foo(x,y,z):
...     print("x=" + str(x))
...     print("y=" + str(y))
...     print("z=" + str(z))
...
>>> l = [1,2,3]
>>> t = (4, 5, 6)
>>> d = {'z': 'foo', 'x': 'bar', 'y': 'baz'}

>>> foo(*l) # same foo(*t)
x=1
y=2
z=3
>>> foo(**d)
x=bar
y=baz
z=foo

```

Vediamo le regole di scoping, ovvero come Python dove cerca i valori delle variabili dati i nomi forniti in programmazione.

Un *namespace* è abbinamento di nomi ad oggetti presenti in memoria; sono *creati in diversi momenti* e hanno *differente durata*. I principali sono:

- Per ottenere la lista di nomi di un namespace si utilizza `dir`:

4.3.2 Attributi

Qualsiasi nome che segue un punto: ad esempio in `z.real`, `real` è un attributo di `z` (qualsiasi cosa sia).

- è possibile assegnarvi qualcosa mediante `x.attributo = valore`;
- è possibile eliminarli mediante `del x.attributo`, che rimuove attributo da `x`.

¹Le funzioni ricorsive hanno un namespace per ogni chiamata

4.3.3 Scoping

All'interno della chiamata di una funzione, la ricerca di un nome avviene nel seguente ordine:

1. nel namespace della *funzione corrente*;
2. nella funzione *enclosing* (quella all'interno del quale la funzione corrente è stata definita²); dopodichè la *enclosing della enclosing*, e così via;
3. il *namespace del modulo* corrente, sia esso `main` o importato
4. il namespace delle *builtin functions*.

Eccezioni: `nonlocal` e `global` Alcuni casi particolari:

- `nonlocal` serve per dichiarare nomi che devono essere presi non dal namespace corrente ma da quelli enclosing

```
>>> a = 1

>>> def scope_local():
...     a = 2
...     print(a)
...
>>> scope_local()
2

>>> def scope_nonlocal_read():
...     a = 3
...     def nested():
...         nonlocal a
...         print(a)
...     nested()
...
>>> scope_nonlocal_read()
3

>>> def scope_nonlocal_write():
...     a = 3
...     def nested():
...         nonlocal a
...         a = 4
...     nested()
...     print(a)
...
>>> scope_nonlocal_write()
4
```

- se un nome è dichiarato mediante `global` la lettura e scrittura da tale variabile va a inficiare il namespace del modulo, indipendentemente da dove essa sia stata effettuata

²Questo è quello che permette le *function factory*.

```
>>> a = 1

>>> def scope_global():
...     global a
...     print(a)
...
>>> scope_global()
1
```

4.4 lambda e funzioni anonime

Le funzioni anonime:

- sono create mediante la keyword `lambda`. Ad esempio la seguente ritorna la somma dei due argomenti passati:

```
lambda a, b: a + b
```

Possono essere assegnate (volendo) comportandosi come normali funzioni

```
>>> x = lambda a, b : a * b
>>> x(5, 6)
30
```

- possono essere usate dove è necessaria una funzione (il suo nome). Ad esempio anche con:

```
>>> (lambda a, b: a+b)(1, 2)
3
```

- sono limitate a *una singola espressione* come corpo

4.5 Programmazione funzionale

Le funzioni sono first class object (quindi possono essere date in input e ritornate come output) e ciò rende possibile la programmazione funzionale. In questa si possono sfruttare caratteristiche del linguaggio come:

- funzioni builtin tipiche della pf: `map`, `filter` etc
- funzioni anonime definite mediante `lambda`
- per i dati analizzati: *iterabili* e *iteratori* (quindi anche generatori ed espressioni generatrici) e le funzioni utili su di essi (es `enumerate`, `sorted`, `any`, `all`, `zip`)

A livello di pacchetti

- `itertools`: iteratori comuni e funzioni per elaborarli
- `functools`: high order functions (funzioni che processano funzioni modificandole)
- `operator`: funzioni che corrispondono agli operatori del python e possono aiutare in un approccio funzionale (evitandoci di scrivere funzioni triviali per effettuare singole operazioni)

4.5.1 Funzioni classiche

4.5.1.1 `map` (= `Map/lapply`)

La builtin

```
map(f, iterA, iterB, ...)
```

restituisce un iteratore sulla sequenza che applica la funzione con argomenti presi dagli iterabili passati:

```
f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), ....
```

Può essere usata anche con un solo iterabile, e nel caso funziona tipo `lapply` di R. Ad esempio:

```
>>> def upper(s):
...     return s.upper()
...
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
```

Altro esempio, per applicare un set di n funzioni su un set di n input:

```
>>> def per2(x):
...     return [y*2 for y in x]
...
>>> def per3(x):
...     return [y*3 for y in x]
...
>>> f = (per2, per3)
>>> i = ([1,2,3], [4,5,6])

>>> def apply(f, i):
...     return f(i)
...
>>> list(map(apply, f, i))
[[2, 4, 6], [12, 15, 18]]
```

4.5.1.2 `itertools.starmap`

Applicata ad un singolo iterabile crea un iteratore che valuta la funzione utilizzando argomenti ottenuti dall'iterabile; da utilizzare al posto di `map` quando gli input sono già stati raggruppati (iterabile di iterabili) o zippati nell'iterabile passato

```
>>> from itertools import starmap

>>> def custom_f(a, b, c):
...     return(a * b - c)
...
>>> x = (2,5,2)
>>> y = (3,2,1)
>>> z = (10,3,2)
```



```
>>> # nel primo caso prende a,b,c separatamente da x (poi y e z)
>>> # nel secondo da tutti e tre

>>> list(starmap(custom_f, [x, y, z]))
[8, 5, 28]
>>> list(starmap(custom_f, zip(x, y, z)))
[-4, 7, 0]
```

4.5.1.3 `functools.reduce`

Applica una funzione (che prende in input due parametri e ne restituisce uno) agli elementi di una sequenza:

- la funzione è applicata ai primi due, il risultato applicato insieme al terzo elemento, e così via
- se `initializer` è presente viene piazzato prima della sequenza nel calcolo e funge da default se la sequenza è vuota;

```
>>> from functools import reduce
>>> from operator import add

>>> reduce(add, [1,2,3])
6
>>> reduce(add, [1,2,3], 5)
11
```

4.5.1.4 `itertools.accumulate`

A differenza di `functools.reduce` (che da il risultato finale) restituisce un iteratore sui risultati parziali:

```
>>> from itertools import accumulate
>>> from operator import add

>>> res = accumulate([1,2,3], add)
>>> list(res)
[1, 3, 6]
```

4.5.1.5 `filter`

```
filter(predicate, iter)
```

ritorna un iteratore su tutti gli elementi che rispettano un `predicate`, ossia una funzione che restituisce `True` o `False`. Per funzionare con `filter`, il predicato deve prendere in input un singolo parametro

```
>>> def is_even(x):
...     return (x % 2) == 0
...
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

Su iteratori funzionalità analoghe si hanno con `filterfalse` e `takewhile` di `itertools`

4.5.2 Partialling

Consiste nel creare una copia di una funzione con uno o più argomenti settati ad un default

4.5.2.1 `functools.partial`

Per parzializzare funzioni

```
>>> from functools import partial

>>> def spam(a, b, c, d):
...     print(a, b, c, d)
...
>>> s1 = partial(spam, 1)           # a = 1
>>> s1(4, 5, 6)
1 4 5 6

>>> s2 = partial(spam, d = 42)    # d = 42
>>> s2(4, 5, 6)
4 5 6 42

>>> s3 = partial(spam, 1, 2, d = 42) # a = 1, b = 2, d = 42
>>> s3(5)
1 2 5 42
```

4.5.2.2 `functools.partialmethod`

Per parzializzare metodi

```
from functools import partialmethod

class Cell:

    def __init__(self):
        self._alive = False

    @property
    def alive(self):
        return self._alive

    # general method ...
    def set_state(self, state):
        self._alive = bool(state)

    # ... and partialled methods
    set_alive = partialmethod(set_state, True)
    set_dead = partialmethod(set_state, False)

c = Cell()
c.alive # False
```



```
>>> f = compose(add2, mul2) # 2x + 2
>>> g = compose(mul2, add2) # 2*(x+2)

>>> X = [1,2,3]
>>> [f(x) for x in X]
[4, 6, 8]
>>> [g(x) for x in X]
[6, 8, 10]
```

Alternativamente, `functools.reduce` può essere utilizzata per comporre una lista di funzioni, ad esempio:

```
>>> import functools

>>> def compose(f, g):
...     return lambda x: f(g(x))
...
>>> funcs = [lambda s: s + "a",
...          lambda s: s + "j",
...          lambda s: s + "e",
...          lambda s: s + "j",
...          lambda s: s + "e"]
>>> # questo è necessario se compose è definita
>>> # applicando la funzione sinistra come seconda (come in math) invece
>>> # di prima, come pensiamo a livello di logica sopra
>>> funcs.reverse()

>>> worker = functools.reduce(compose, funcs)
>>> worker("brazorv_")
'brazorv_ajeje'
```

4.5.5 Decorators

Un decorator è una funzione, solitamente³, che wrappa un'altra funzione modificandone il comportamento standard in qualche modo

4.5.5.1 Definizione e utilizzo

Nel Python è possibile definirli una volta ed utilizzare una sintassi speciale/compatta per applicarli a molteplici funzioni. Di base il funzionamento è il seguente:

```
def foo():
    # do something

def decorator(fun):
    # manipulate fun
    return fun
```

Una volta definito possiamo usarlo come

³Decorator può essere qualsiasi callable, ovvero un oggetto che presenta il metodo `__call__`

```
foo = decorator(foo)  # Manually decorate
```

```
@decorator
def bar():
    # Do something
    # bar() is decorated
```

4.5.5.2 Decoratori già disponibili

Alcuni utili sono

- `functools.cache` memorizza i risultati di una funzione in un dict e controlla alla seconda chiamata che non sia già disponibile (tipo `memoize` usata sotto)
- ...

4.5.5.3 Esempi di creazione/utilizzo di custom

Un semplice decorator Ad esempio supponiamo di avere una funzione che calcoli l'*i*-esimo numero di Fibonacci (con index che partono da 0) in maniera particolarmente inefficiente

```
>>> def fib(n):
...     if n in (0,1):
...         return n
...     else:
...         return fib(n - 1) + fib(n - 2)
...
>>> fib(3)
2
```

per renderla più veloce (specialmente in seguito ad utilizzo ripetuto) utilizziamo la tecnica di *memoization* ovvero salvare risultati parziali in una cache

```
>>> def memoize(f):
...     cache = {}
...     def helper(x):
...         if x not in cache:
...             cache[x] = f(x)
...         return cache[x]
...     return helper
...
```

Per applicare la memoizzazione si farebbe:

```
>>> memoized_fib = memoize(fib)
>>> memoized_fib(3) # ritorna 2
2
```

La funzione `memoize` funge da decoratore della funzione `fib`; come sintassi specifica del python, una volta definita `memoize` potevamo decorare la definizione di `fib` (alternativamente a creare `memoized_fib`) come segue:

```
>>> @memoize
... def fib(n):
...     # Stesso codice di prima
...     if n in (0,1):
...         return n
...     else:
...         return fib(n - 1) + fib(n - 2)
...
>>> fib(3) # memoized
2
```

Decorator multipli I decorator possono essere concatenati, facendo sì che ad una funzione base vengano applicati più decorator contemporaneamente (aggiungendo feature in maniera pulita).

Ad esempio se vogliamo aggiungere sia la memoizzazione che il logging alla funzione `fib` di cui prima, definiamo prima i due decorator `memoize` e `trace`, dopodichè decoriamo la definizione di `fib`

```
>>> def trace(f):
...     def helper(x):
...         call_str = "{0}({1})".format(f.__name__, x)
...         print("Calling {0} ...".format(call_str))
...         result = f(x)
...         print("... returning from {0} = {1}".format( call_str, result))
...         return result
...     return helper
...
>>> @memoize
... @trace
... def fib(n):
...     # stesso codice di prima
...     if n in (0,1):
...         return n
...     else:
...         return fib(n - 1) + fib(n - 2)
...
>>> fib(3)
Calling fib(3) ...
Calling fib(2) ...
Calling fib(1) ...
... returning from fib(1) = 1
Calling fib(0) ...
... returning from fib(0) = 0
... returning from fib(2) = 1
... returning from fib(3) = 2
2
>>> fib(1)
1
>>> fib(4)
Calling fib(4) ...
... returning from fib(4) = 3
```

3

4.5.6 Single e multiple dispatch

4.5.6.1 Single dispatch: `functools singledispatch` e `functools singledispatchmethod`

Le funzioni con single dispatch sono tipo le S3 di R, che stabiliscono come comportarsi (evitando `if/elif`) sulla base dell'input fornito.

`singledispatch` Per definire una funzione generica la si decora con `singledispatch` di `functools`.

```
>>> from functools import singledispatch

>>> @singledispatch
... def f(arg, verbose = False):
...     print("Unhandled arg")
...
>>> @f.register
... def _(arg: int, verbose = False):
...     print(arg, "is integer")
...     if verbose:
...         print("f call was verbose too")
...
>>> @f.register
... def _(arg: str, verbose = True):
...     print(arg, "is a string")
...     if verbose:
...         print("f call was verbose too")
...
>>> f(3)
3 is integer
>>> f("foo")
foo is a string
f call was verbose too
>>> f("bar", verbose = False)
bar is a string

>>> ## La definizione si può veder scritta anche come segue, dove il tipo
>>> ## di arg è passato a f.register

>>> @f.register(list)
... def _(arg, verbose = True):
...     print(arg * 2)
...
>>> f([1,2,3])
[1, 2, 3, 1, 2, 3]

>>> ## Si possono registrare lambda e funzioni pre-esistenti utilizzando
>>> ## register in chiamata
```

```

>>> def none_dispatch(arg, verbose = False):
...     print("You passed nothing.")
...
>>> f.register(type(None), none_dispatch)
<function none_dispatch at 0x7fb4d44d22a0>

>>> f(None)
You passed nothing.

>>> ## register può essere usata in stack per definire una medesima
>>> ## funzione per diversi tipi, o in combinazione con altri decoratori
>>> ## (es per test indipendenti)

>>> @f.register(int)
... @f.register(float)
... def _(arg, verbose = False):
...     print(arg * 2)
...
>>> f(2)
4
>>> f(3.5)
7.0

>>> ## se passiamo un oggetto non conosciuto dalla funzione ritorna alla
>>> ## base
>>> a_dict = {'planet':'earth', 'region':'europe', 'prefix':39}
>>> f(a_dict)
Unhandled arg

```

singledispatchmethod TODO

4.5.6.2 Multiple dispatch

Si implementa mediante `multimethod` che fornisce decoratori per il multiple dispatch. Un esempio che riproduce il funzionamento di `*` con numeri e stringhe

```

>>> from multimethod import multimethod # pip install -U multimethod
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'multimethod'

>>> @multimethod
... def test(a, b): # default senza specificare il tipo
...     print("boo")
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'multimethod' is not defined. Did you mean: 'staticmethod'?
>>> @multimethod
... def test(a: int, b: int): # particularizzando il tipo

```



```

...     return a + b
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'multimethod' is not defined. Did you mean: 'staticmethod'?
>>> @multimethod
... def test(a: str, b: int):
...     return a * b
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'multimethod' is not defined. Did you mean: 'staticmethod'?
>>> test(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'test' is not defined
>>> test("a", 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'test' is not defined
>>> test("e", "qui")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'test' is not defined

```

Un esempio nel caso di metodi di una classe, la funzione sceglie il metodo sulla base dei tipi degli attributi dell'oggetto

```

from dataclasses import dataclass
from multimethod import multimethod

@dataclass
class Test:
    x: int|str
    y: int|str|None = None

    def do(self):
        self._dispatch(self.x, self.y)

    @multimethod
    def _dispatch(self, a, b): # default case
        print("dunno")

    @multimethod
    def _dispatch(self, a: int, b: None):
        print("int x none")

    @multimethod
    def _dispatch(self, a: str, b: None):
        print("str x none")

    @multimethod

```

```
def _dispatch(self, a: int, b: str):
    print("int x str")

    @multimethod
    def _dispatch(self, a: str, b: str):
        print("str x str")

    # etc..

## what is printed is always the default case
Test(1).do() # prints int x none
Test("test").do() # prints str x none
Test(1, "test").do() # prints int x str
Test("test", "test").do() # prints str x str
```

Capitolo 5

Input/Output

Contents

5.1	Lettura/scrittura file testuali	75
5.1.1	Testo semplice	75
5.1.2	Formati tabulari (csv, tsv)	76
5.1.3	JSON	77
5.2	Accesso al filesystem	78
5.2.1	Ottenere/cambiare directory di lavoro	79
5.2.2	Listing di directory e glob files	79
5.2.3	Creazione/rimozione di file e directory	79
5.2.4	Manipolazione di path e metodi utili	79
5.2.5	Creazione filename temporaneo	81
5.2.6	Uso di file e directory temporanei	81
5.3	Esecuzione di programmi esterni	82

5.1 Lettura/scrittura file testuali

5.1.1 Testo semplice

La funzione `open` prende in input un path e un mode (`r` per la lettura, `w` per la scrittura, `a` per l'append) restituisce un file object, quando si è finito di operare chiamare il metodo `close`:

```
f = open(file = '/tmp/test.txt', mode = 'r')
f.operazioni
f.close()
```

o meglio/più compattamente

```
with open(file = '/tmp/test.txt', mode = 'r') as f:
    f.operazioni
```

in questo secondo caso il file viene chiuso automaticamente all'uscita dal blocco anche in caso di eccezioni (warning, error).

È possibile anche utilizzare oggetti `pathlib.Path`, che hanno il metodo `open`:

```
from pathlib import Path
p = Path("/etc/motd")
with p.open() as f:
    lines = f.readlines()
print(lines)
```

5.1.1.1 Lettura

Alcuni metodi:

- `read` legge tutto il file e lo restituisce come stringa. Quando si raggiunge la fine del file, una chiamata successiva a `read` restituisce una stringa vuota

```
with open(file = '/tmp/test.txt', mode = 'r') as f:
    whole_file = f.read()
```

- `readline` legge una singola riga; anche qui quando si giunge alla fine del file restituisce una stringa vuota. `readlines` legge tutte le righe e le restituisce come lista

```
with open(file = '/tmp/test.txt', mode = 'r') as f:
    lines = f.readlines()
```

- se si vuole processare linea per linea si può ciclare su `f` come segue, dato che il file è un iteratore sulle righe

```
with open(file = '/tmp/test.txt', mode = 'r') as f:
    for line in f:
        # process line, es per stampa a video
        print(line)
```

5.1.1.2 Scrittura

Per scrittura su file si può usare `write` o `writelines` a seconda che l'input sia una stringa o una lista di stringhe; (alternativamente `print`)

```
# scrittura di stringa (o """...""")
with open(file = '/tmp/test.txt', mode = 'w') as f:
    f.write("test") # write restituisce il numero di caratteri scritti
    # print("test", file = f) # alternativamente

# scrittura di una lista di linee
L = ["line1", "line2"]
with open(file = '/tmp/test.txt', mode = 'w') as f:
    f.writelines(L)
```

5.1.2 Formati tabulari (csv, tsv)

Il modulo `csv` contiene le funzioni `reader` e `writer` per leggere formati testuali tabulari di vario tipo (anche separati da tab).

L'apertura/chiusura del file avviene come qualsiasi file di testo, come visto prima.

5.1.2.1 Lettura

Per importare un file e processarlo una riga alla volta `csv.DictReader` restituisce ciascuna riga come dict (altrimenti se può bastare utilizzare `csv.reader` che ritorna una lista)

```
# CSV esempio
#
# dir,repo,link    # attenzione a non lasciare spazi
# ~/.av/, https://lbraglia@bitbucket.org/lbraglia,
# ~/.configs/, https://lbraglia@github.com/lbraglia/.configs, ~/.configs/
# ~/.dnd/, https://lbraglia@github.com/lbraglia/.dnd, ~/.dnd

import csv
with open('setup.csv') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        dir = row['dir']    # attenzione a non lasciare spazi
        repo = row['repo']
        link = row['link']
        ...
```

5.1.2.2 Scrittura

Quello che si fa è creare un writer in cui si impostano le formattazioni di delimitatore, carattere di quoting e così via, per poi utilizzarlo per scrivere le righe. Un esempio con un dataset separato da tab

```
import csv
with open(file = '/tmp/dataset.tab', mode = 'w') as f:
    dataset = csv.writer(f,
                        delimiter = '\t',
                        quotechar = '"',
                        quoting = csv.QUOTE_NONNUMERIC)
    header = ['x', 'y', 'z']
    data = [1, 2, 'a']
    dataset.writerow(header)
    dataset.writerow(data)
```

5.1.3 JSON

L'omonimo modulo `json` di python permette l'interfaccia. Le funzioni principali sono `dump` e `load` per scrivere su file, o `dumps` e `loads` (dump-s) per scrivere e leggere stringhe. Qua vediamo l'interfaccia con i file.

5.1.3.1 Scrittura

L'encoding dei tipi base (e loro combinazioni) segue tabella 5.1

```
>>> import json

>>> data = {
...     'name' : ['ACME', "FOO", "BAR"],
```

Python	JSON
<code>dict</code>	object
<code>list, tuple</code>	array
<code>str</code>	string
<code>int, float</code>	number
<code>True</code>	true
<code>False</code>	false
<code>None</code>	null

Tabella 5.1: Encoding JSON

JSON	Python
object	<code>dict</code>
array	<code>list</code>
string	<code>str</code>
number (int)	<code>int</code>
number (real)	<code>float</code>
true	<code>True</code>
false	<code>False</code>
null	<code>None</code>

Tabella 5.2: Decoder JSON

```
...  'shares' : [100, 200, 300],
...  'price'  : (321, 9, 8912)
...  }

>>> with open("/tmp/data.json", "w") as f:
...     json.dump(data, f)
... 
```

5.1.3.2 Lettura

Il decoding dei tipi di base segue tabella 5.2.

```
>>> with open("/tmp/data.json", "r") as f:
...     data2 = json.load(f)
...
>>> print(data2)
{'name': ['ACME', 'FOO', 'BAR'], 'shares': [100, 200, 300], 'price': [321, 9, 8912]}
```

5.1.3.3 Formati custom

Per tipi di dati più elaborati (es classi custom, numeri complessi) vedere alcuni esempi qui: <https://realpython.com/python-json/>

5.2 Accesso al filesystem

Oggigiorno si usa `pathlib`, per alcune cose marginali rimasto `os` e `shutil`.

```
>>> from pathlib import Path
>>> import os
>>> import shutil
```

5.2.1 Ottenere/cambiare directory di lavoro

```
>>> # ottenere la directory
>>> os.getcwd()          # quick way, otherwise pathlib.Path.cwd()
'/home/l/.sintesi/sintesi_cs'
>>> Path.cwd()
PosixPath('/home/l/.sintesi/sintesi_cs')

>>> # cambio directory
>>> os.chdir('/tmp')
```

5.2.2 Listing di directory e glob files

```
>>> d = Path("/home/l/.sintesi")

>>> # listing di directory
>>> list(d.iterdir()) # paths del contenuto (iteratore)
[PosixPath('/home/l/.sintesi/sintesi_biostat'), PosixPath('/home/l/.sintesi/lab'), PosixPath('/

>>> # glob/matching nome file
>>> pdf1 = sorted(d.glob("*/*.pdf")) # cerca nella sottodirectory figlie
>>> pdf2 = sorted(d.glob("**/*.pdf")) # cerca in tutto l'albero
>>> pdf3 = sorted(d.rglob("*.pdf"))  # stessa cosa di sopra ma più compatta
```

5.2.3 Creazione/rimozione di file e directory

```
>>> # Creazione path/file
>>> d = Path('/tmp/barbaz/whahaa/yeah')

>>> # Creazione directory
>>> d.mkdir(parents = True) # parents se vogliamo creare tutto l'albero

>>> # Rimozione directory con tutto il contenuto
>>> rm = Path('/tmp/barbaz')
>>> # rm.rmdir() # errore: cartella deve essere vuota...
>>> shutil.rmtree(rm) # vedere dopo per shutil

>>> # Creazione rimozione di file
>>> f = Path('/tmp/asdomar.txt')
>>> f.touch()
>>> f.unlink()
```

5.2.4 Manipolazione di path e metodi utili

```
>>> f = Path("/usr/bin/python")
>>> d = Path("~/sintesi")
>>> f2 = Path("/etc/apt/sources.list")
```

```

>>> f3 = Path("Makefile")
>>> subdir = "sintesi_cs"

>>> ## questi path possono esser usati dove è accettato un os.PathLike
>>> ## alternativamente per ottenere la rappresentazione in stringa

>>> str(d)
'~/sintesi'
>>> str(f)
'/usr/bin/python'

>>> # sostituire la tilde per l'amor del cielo in paths
>>> de = d.expanduser()

>>> # test esistenza
>>> f.exists()
True
>>> d.exists() # attenzione a ~
False
>>> de.exists()
True

>>> # test tipologia
>>> f.is_file()
True
>>> d.is_dir() # attenzione a ~
False
>>> de.is_dir()
True

>>> # links
>>> f.is_symlink()
True
>>> f.readlink() # follow just one redirection
PosixPath('python3')
>>> os.path.realpath(f) # follow all redirections
'/usr/bin/python3.13'

>>> # estrarre parti di un path
>>> f2.name # nome file
'sources.list'
>>> f2.stem # nome file senza estensione
'sources'
>>> f2.suffix # estensione - per tar.gz usare suffixes
'.list'

>>> # controllare l'estensione (mediante glob)
>>> f2.match("*.list")
True

>>> # creazione di path mediante modifica di esistente

```



```
>>> d / subdir # concatenazione, alternativamente d.joinpath(subdir)
PosixPath('~/.sintesi/sintesi_cs')
>>> f.with_name("foo.list") # nome cambiato
PosixPath('/usr/bin/foo.list')
>>> f.with_stem("ssd") # stem cambiato, stessa estensione
PosixPath('/usr/bin/ssd')
>>> f.with_suffix(".deb") # estensione cambiata, stesso stem
PosixPath('/usr/bin/python.deb')
>>> f2.relative_to("/etc") # path relativo, escludendo quanto passato
PosixPath('apt/sources.list')
>>> f3.absolute() # crea l'assoluto a partire da un relativo
PosixPath('/tmp/Makefile')
>>> f3.resolve() # stessa cosa ma fixa anche symlink
PosixPath('/tmp/Makefile')
```

5.2.5 Creazione filename temporaneo

Questo crea anche il file e non lo elimina alla fine (di default lo crea in /tmp quindi viene). Si può però utilizzare il nome in un secondo momento scrivendo sul file.

```
import tempfile
tempfile = tempfile.mkstemp()
fname = tempfile[1]
```

5.2.6 Uso di file e directory temporanei

```
>>> from tempfile import TemporaryFile
>>> from tempfile import TemporaryDirectory

>>> # File
>>> # ----
>>> with TemporaryFile('w+t') as f:
...     f.write("Hello World\n")
...     f.write("Testing\n")
...     # ritorna ad inizio file e leggi quanto scritto
...     f.seek(0)
...     data = f.read()
...
12
8
0
>>> # qui il file non c'è più ma ..
>>> data
'Hello World\nTesting\n'

>>> # Directory
>>> # -----
>>> with TemporaryDirectory() as dirname:
...     print("dirname is: ", dirname)
...     ## use the directory
```

```
...
dirname is: /tmp/tmps1h9x_ms
>>> # qui directory e suo contenuto non c'è più
```

Il mode è `w+t` per il testo o `w+b` per binario: si è posto `'w'` per permettere sia lettura che scrittura che è utile qua, dato che chiudere il file implicherebbe distruggerlo.

5.3 Esecuzione di programmi esterni

Si usa la library `subprocess`:

- se si vuole **attendere il termine del processo** utilizzare `run` (funzione di convenienza consigliata). Si specificano eventuali comandi composti da più token come lista

```
>>> import subprocess

>>> # senza prendere i risultati in input
>>> subprocess.run("ls") # comando semplice
CompletedProcess(args='ls', returncode=0)
>>> subprocess.run(["ls", "-l"]) # più elementi
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> # prendendo i risultati (testuali) in input
>>> subprocess.run(["ls", "/home/l/cs/python"],
...                 capture_output = True,
...                 text = True)
CompletedProcess(args=['ls', '/home/l/cs/python'], returncode=0, stdout='control_
>>> print(subprocess.run(["ls", "/home/l/cs/python"],
...                       capture_output = True,
...                       text = True).stdout)
control_flow.tex
dati.tex
debugging.tex
descriptive.tex
funzioni.tex
grafici.tex
img
inference.tex
integrazione_r.tex
intro.tex
io.tex
linalg.tex
matplotlib.tex
misc_cookbook.tex
numpy.tex
oop.tex
packages.tex
pandas.tex
prob_sim.tex
```

```
_region_.tex  
testing.tex
```

- **altrimenti** Popen (classe più generale):

```
subprocess.Popen(["rm", "-rf", "/tmp/asdomarasdasd"])  
subprocess.Popen(["sleep", "30"])  
print("ciao")
```

Si può attendere anche con Popen se chiamiamo il metodo `communicate` sull'oggetto ritornato; il flusso python si stopperà fino a che il processo ritorna

```
ls_output=subprocess.Popen(["sleep", "30"])  
ls_output.communicate() # questo effettivamente stoppa python per 30 secondi
```


Capitolo 6

Debugging ed eccezioni

Contents

6.1	Debugging	85
6.1.1	Ispezione della traceback	85
6.1.2	Esecuzione in modalità debugging	86
6.1.3	Eseguire script in modalità debugging	86
6.1.4	Un equivalente di <code>browser</code>	86
6.2	Gestire eccezioni	87
6.2.1	Sintassi minimale: <code>try except</code>	87
6.2.2	<code>else</code> e <code>finally</code> in <code>try</code>	88
6.3	Sollevare eccezioni	89
6.4	Creare ed utilizzare eccezioni custom	90
6.5	Sollevare warnings senza stoppare l'esecuzione	90

6.1 Debugging

Quando uno script fallisce, quello che fa è sollevare una eccezione, il modo per dire che qualcosa è andato storto: e informazioni sulla causa dell'errore si ritrovano nella *traceback* (la serie di chiamate che ha condotto all'errore) stampata.

6.1.1 Ispezione della traceback

Se si usa `ipython` mediante `xmode` si può settare il livello di dettaglio fornito nella traceback

```
%xmode Plain      # compatto (comunque più verboso dell'interprete vanilla)
%xmode Context    # default/standard
%xmode Verbose    # aggiunge i valori dei parametri in chiamata

def d(a, b):
    return a / b

def f(x):
```

```

a = x
b = x - 1
return d(a, b)

# sotto da eseguire in modalità interattiva
# f(1)
# %xmode Verbose
# f(1)

```

6.1.2 Esecuzione in modalità debugging

Se l'ispezione della traceback non basta si può desiderare eseguire il codice linea per linea (con possibilità di interagire con l'ambiente) per capire dove il problema. Il tool standard di python è `pdb`, `ipython` fornisce la versione potenziata `ipdb`. Entrambi hanno svariati modi con cui possono esser lanciati. In `ipython` il modo più conveniente è forse l'uso del magic command `debug`: se chiamato dopo che è stata sollevata una eccezione viene aperto un prompt nel punto dell'eccezione, dal quale è possibile stampare variabili apponendo `!` o `p`, muoversi nella stack (per salire o scendere nelle chiamate si usa `up` e `down`) e uscire mediante `quit` (altri comandi utili in modalità debugging sono riportati in tabella 6.1):

```

f(1)      # generare l'errore
%debug    # siamo nella stack di d
dir()     # ['a', 'b']
!a        # 1
!b        # 0
u         # andiamo nella stack di f
dir()     # ['a', 'b', 'x']
quit      # uscire dalla modalità debugging

```

Se si desidera che la modalità debugging sia attivata automaticamente se viene sollevata una eccezione usare la magic `pdb` (come toggle)

```

# attivazione
%pdb
# disattivazione
%pdb

```

6.1.3 Eseguire script in modalità debugging

Per eseguire uno script in modalità debugging

```

% run -d nomescrpt.py
next    # per andare alla prossima istruzione

```

6.1.4 Un equivalente di browser

Per un equivalente di browser (R) usare:

```

import ipdb
ipdb.set_trace()  ## piazzarlo dove serve

```

Command	Description
<code>u(p)</code>	sali nella stack
<code>d(own)</code>	scendi nella stack
<code>list</code>	Show the current location in the file
<code>h(elp)</code>	Show a list of commands, or find help on a specific command
<code>q(uit)</code>	Quit the debugger and the program
<code>c(ontinue)</code>	Quit the debugger, continue in the program
<code>n(ext)</code>	Go to the next step of the program
<code>ENTER</code>	Repeat the previous command
<code>p(rint)</code>	Print variables
<code>s(tep)</code>	Step into a subroutine
<code>r(eturn)</code>	Return out of a subroutine

Tabella 6.1: Comandi debug

6.2 Gestire eccezioni

È possibile gestire le eccezioni, per evitare che terminino l'esecuzione necessariamente. Per farlo si usa `try`.

6.2.1 Sintassi minimale: `try except`

Un esempio

```
try:
    x = int("prova")
except ValueError:
    print("Ops valore non coercibile")
    # se vogliamo interrompere l'interprete (es errore grave)
    sys.exit(1)
    # se vogliamo solo impedire che l'esecuzione prosegua normalmente (ma
    # lasciare l'interprete vivo)
    informative_msg = "some info"
    raise ValueError(informative_msg)
    # Altrimenti senza sys.exit o raise il codice prosegue normalmente
```

Nell'ordine:

- viene eseguito lo statement tra `try` ed `except` (try clause)
- se non viene sollevata alcuna eccezione, lo statement `try` termina
- se una eccezione viene sollevata, quando ciò avviene si blocca l'esecuzione e:
 - se il tipo dell'eccezione mostrata matcha con uno di quelli programmati, viene eseguito il relativo codice (clausola `except`, dopodichè si esce dal blocco `try`;
 - se il tipo non matcha, essa viene trasmessa a eventuali istruzioni `try` di livello superiore; se non viene trovata una clausola che le gestisca, si tratta di un'eccezione non gestita e l'esecuzione si ferma con un messaggio

Un'istruzione `try` può avere più clausole `except`, (per specificare gestori di differenti eccezioni) o si può specificare un unico handler per molteplici eccezioni, come segue:

```
except (RuntimeError, TypeError, NameError):
    pass
```

6.2.2 else e finally in try

`else` è opzionale e va posta dopo tutte le `except`: serve per eseguire codice quando `try` va a buon fine

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

`finally` è opzionale e serve per azioni di pulizia che vengono eseguite in ogni caso

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Nelle applicazioni reali, la clausola `finally` è utile per rilasciare risorse esterne (file, network connection) indipendentemente dal fatto che l'uso della risorsa sia andata a buon fine.

6.3 Sollevare eccezioni

Lo statement `raise` permette di sollevare problemi;

```
>>> raise NameError('Cosa stai dicendo?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Cosa stai dicendo?
```

La gerarchia delle eccezioni disponibili allo stato attuale, cercare di utilizzare l'eccezione più appropriata, da conoscere mediante `help(nomeeccezione)`.

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |       +-- BrokenPipeError
    |       +-- ConnectionAbortedError
    |       +-- ConnectionRefusedError
    |       +-- ConnectionResetError
    +-- FileExistsError
    +-- FileNotFoundError
    +-- InterruptedError
    +-- IsADirectoryError
    +-- NotADirectoryError
    +-- PermissionError
    +-- ProcessLookupError
    +-- TimeoutError
```

```

+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
|   +-- DeprecationWarning
|   +-- PendingDeprecationWarning
|   +-- RuntimeWarning
|   +-- SyntaxWarning
|   +-- UserWarning
|   +-- FutureWarning
|   +-- ImportWarning
|   +-- UnicodeWarning
|   +-- BytesWarning
|   +-- ResourceWarning

```

6.4 Creare ed utilizzare eccezioni custom

Se l'utente vuol definire delle eccezioni custom deve implementarle mediante classi; in questo modo è possibile creare gerarchie estensibili di eccezioni.

Una volta definita (può esser anche vuota mediante `pass`) vi sono due modi per sollevare una eccezione:

```

raise Classe
raise Istanza

```

6.5 Sollevare warnings senza stoppare l'esecuzione

```

import warnings
warnings.warn("Warning message")

```

Capitolo 7

Object Oriented Programming

Contents

7.1	Classi	91
7.1.1	Definizione e scoping	91
7.1.2	Metodi	93
7.1.3	Condivisione dati	93
7.1.4	Data hiding	94
7.2	Ereditarietà	95
7.2.1	Singola	95
7.2.2	Multipla	95
7.3	Classi notevoli	96
7.3.1	<code>dataclass</code>	96
7.3.2	Iteratori	98
7.3.3	Context Managers	105

7.1 Classi

L'implementazione della programmazione ad oggetti si basa su trick di scope (cfr sezione 4.3).

La definizione di una classe semplicemente pone un altro namespace all'interno di quello nel quale ci si trova.

7.1.1 Definizione e scoping

La definizione di una classe fa uso di:

- `class` associato ad un nome;
- una stringa di documentazione opzionale
- vari `statement` di assegnazione di variabili/definizione di funzioni (con una sintassi particolare, i *metodi* della classe)

```
class NomeClasse:
    """
    Documentation
    """
    <statement-1>
    .
    .
    .
    <statement-N>
```

Una volta valutata la definizione viene creato un *oggetto classe*, ossia un *namespace* contenuto in quello dove è stato definito, che supporta supporta due cose:

1. riferirsi ai suoi attributi in lettura o scrittura (nel Python una classe può esser modificata dopo che è stata creata);
2. creare oggetti usando il nome della classe come se fosse una funzione: così facendo si creano *oggetti istanza*.
3. **scoping**: una volta istanziato un oggetto, la ricerca dei entro una classe (a parte le variabili locali ad un metodo) continua nel modulo dove la classe è definita (sia esso `__main__` o importato).

```
>>> b = "ciao"
```

```
>>> # definizione di classe
```

```
>>> class firstClass:
...     """La mia prima classe"""
...     a = 0
...     b = "miao"
...     def get_a(self):
...         return self.a
...     def set_a(self, x):
...         self.a = x
...     def test_scoping(self):
...         print(b)
... 
```

```
>>> firstClass.a = 1          ## modifica di un attributo di una classe, volendo
```

```
>>> print(firstClass.__doc__) ## stampa la docstring della classe
```

```
La mia prima classe
```

```
>>> x = firstClass()          ## istanziamento usando il costruttore di default
```

```
>>> x.a                        ## nulla impedisce di accedere direttamente
```

```
1
```

```
>>> x.get_a()
```

```
1
```

```
>>> x.set_a(3)
```

```
>>> x.get_a()
```

```
3
```

```
>>> x.test_scoping()          # "ciao", non "miao" (per questo avremmo dovuto self.b)
```

```
ciao
```

7.1.2 Metodi

7.1.2.1 Definizione e chiamata

I metodi di una classe presentano una definizione particolare:

- il primo argomento serve per riferirsi all'oggetto istanziato: il nome `self` è arbitrario ma preferibile (standard);
- gli altri sono parametri normali da passare in chiamata del metodo

In sede di chiamata poi:

- `object.function()` è equivalente a chiamare `classe.function(object)`
- chiamare `object.function(x, y, z)` equivale a `classe.function(object, x, y, z)`

7.1.2.2 Costruttore custom

Se si desidera specificare un costruttore custom diverso da quello di default, si definisce una funzione di nome `__init__` che si occupa di inizializzare i valori (nel quale abbiamo messo anche inizializzatori di default):

```
>>> class Complex:
...     def __init__(self, realpart = 0, imagpart = 0):
...         self.r = realpart
...         self.i = imagpart
...     def value(self):
...         return('' + str(self.r) + '+' + str(self.i) + 'i')
...
>>> x = Complex()
>>> x.value()
'0+0i'
>>> y = Complex(1, 2)
>>> y.value()
'1+2i'
```

7.1.3 Condivisione dati

I dati di una classe iniziano ad esistere dal momento in cui vengono assegnati, sia ciò in definizione della classe o usando un metodo nell'oggetto istanza. Pertanto:

- gli argomenti **inizializzati nella definizione della classe** sono *comuni* a tutti gli oggetti generati;
- gli argomenti **inizializzati mediante un metodo** sono *caratteristici* dell'oggetto istanziato.

```
>>> class Dog:
...     kind = 'canine'          # class variable shared by all instances
...     def __init__(self, name):
...         self.name = name    # instance variable unique to each instance
...
>>> d = Dog('Fido')
```

```

>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'

```

Attenzione a quando come *dato di classe* vi è un *tipo mutabile*: la chiamata di metodi da diverse istanze andrà a modificare un dato comune (e non è spesso quello che si vuole).

```

>>> class Dog:
...     tricks = []                # Sbagliato: questo sarà condiviso da tutti i ca
...     def __init__(self, name):
...         self.name = name
...     def add_trick(self, trick):
...         self.tricks.append(trick)
...
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over', 'play dead']

```

```

>>> class Dog:                    # Versione corretta
...     def __init__(self, name):
...         self.name = name
...         self.tricks = []      # ok
...     def add_trick(self, trick):
...         self.tricks.append(trick)
...
>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']

```

7.1.4 Data hiding

Nel Python

- non vi è un concetto di *data hiding*; di un oggetto istanziato si può accedere ai valori/funzioni senza problemi

- se si desidera **celare un elemento** gli si può dare un nome che inizi con “_” (cose che dovrebbero essere considerate come “private”)
- per **incrementare la celatura**, se si da un nome che inizia con __, viene preceduto dal nome della classe ed underscore, come segue (*name mangling*):

```
>>> class Asd:
...     __a = 0  ## nome della variabile più due underscore
...
>>> foo = Asd()
>>> foo._Asd__a  ## accediamo ad a comunque, ma è faticoso farlo
0
```

7.2 Ereditarietà

7.2.1 Singola

Si ha che:

- la sintassi per definire una classe che eredita da un'altra è

```
class NomeClasseDerivata(NomeClasseBase):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- nella ricerca nomi, nel caso nel caso non vengano trovati nella classe derivata si procederà nella classe base (e ricorsivamente nelle classi ad essa base);
- le classi derivate possono *specializzare i metodi*, ridefinendoli con il medesimo nome della classe base.

7.2.2 Multipla

- la definizione avviene come segue:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- la classe eredita da tutte e tre le classi di base;
- la risoluzione di nomi funziona prima in profondità per **Base1** (cercando anche nelle classi dalle quali questa eredita), poi passando a **Base2** (e quindi in profondità) e a **Base3**(e in profondità), evitando di cercare due volte nella stessa classe se vi sono sovrapposizioni nell'albero genealogico.

7.3 Classi notevoli

7.3.1 dataclass

Il modulo `dataclasses` fornisce un decoratore da utilizzare con le classi che vogliamo battezzare come `dataclass`

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Quello che questo decoratore fa è aggiungere un iniziatore di default del tipo seguente, senza bisogno di specificarlo,

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

La `dataclass` aggiunge anche una `__repr__` gratuita per la stampa dei dati dell'oggetto (specificare `repr=False` nella funzione `field` esclude il dato dalla stampa)

7.3.1.1 field: dati mutabili, valori default, parametri

Se occorre specificare dati mutabili a livello di singola istanza (non condivisi) tra tutti gli oggetti di una determinata classe bisogna utilizzare `field` specificando la funzione utilizzata per creare l'istanza. Ad esempio se vogliamo un campo indirizzi email (lista di stringhe) che non sia condiviso tra tutti gli oggetti della classe dobbiamo programmare qualcosa del genere

```
from dataclasses import dataclass
from dataclasses import field

@dataclass
class Person:
    name: str
    address: str
    # email_addresses = [] # condiviso e sbagliato
    email_addresses: list[str] = field(default_factory=list)
```

```
l = Person(name = "Luca", address = "Via XYZ")
```

`field` e `default_factory` possono essere utilizzate per specificare una funzione che crea il valore assegnato di default se l'utente non specifica in chiamata, quindi ha anche altre applicazioni. Nel seguito la generazione di un id casuale


```

import random
import string

from dataclasses import dataclass
from dataclasses import field

# genera un id casuale
def generate_id() -> str:
    return "".join(random.choices(string.ascii_uppercase, k=12))

@dataclass
class Person:
    name: str
    address: str
    email_addresses: list[str] = field(default_factory=list)
    id: str = field(default_factory=generate_id)

```

Se si desidera che un parametro sia impostato ad un valore di default ma che non possa essere scelto dall'utente in chiamata si specifica `init=False` in `field`. Ad esempio per far sì che l'utente non possa scegliere l'id ma che questo sia generato da una funzione

```

@dataclass
class Person:
    name: str
    address: str
    email_addresses: list[str] = field(default_factory=list)
    id: str = field(init=False, default_factory=generate_id)

# questo da errore
l = Person(name="Luca", address="asd", id="foo")

```

Infine `field` accetta:

- un valore di default (utilizzato se l'utente non può o non vuole fornire in inizializzazione) con `default` (es `default=0` per un bank account saldo iniziale)
- `compare` (di default a `True`) che specifica se l'attributo è utilizzato nella comparazione (es per stabilire l'uguaglianza) o meno

7.3.1.2 Eseguire codice post inizializzazione: `post_init`

Se vogliamo eseguire del codice automaticamente post inizializzazione (es generare dati o inizializzare) definiamo la funzione `__post_init__` che verrà eseguita come nome succede. Ad esempio per creare una stringa per la ricerca a partire dai dati forniti in inizializzazione

```

class Person:
    name: str
    address: str
    email_addresses: list[str] = field(default_factory=list)
    id: str = field(init=False, default_factory=generate_id)

```

```

search_string: str = field(init=False) # stringa che non può essere

def __post_init__(self) -> None:
    self.search_string = f"{self.name} {self.address}"

```

7.3.1.3 Freezing di una dataclass

Significa impostarla che una volta inizializzata i suoi dati non possano essere modificati (es mediante semplice assegnazione)

```

@dataclass(frozen=True)
class Person:
    name: str
    address: str
    ...

l = Person(...)
l.name = "foo" # da errore

```

Ocio che anche i `post_init` falliranno se come sopra assegnano alla classe

7.3.1.4 Altri parametri utili del decoratore

Oltre a `frozen` vi sono altri parametri interessanti per il decoratore:

- `kw_only=True` in inizializzazione bisognerà specificare per esteso nome = valore e non verrà accettato il valore associato alla posizione della chiamata
- con `match_arg=False` si disabilita il structural pattern matching (uso con `match`) abilitato di default
- `slots=True`: sotto la scocca una dataclass è un dizionario molto avanzato. Se si abilita `slot` vi è un accesso molto più rapido e di base (miglioramenti del 20% a seconda dei casi). Non si usa di default perché `slots` rompe tutto quando si usa ereditarietà multipla, es quanto segue non funziona perché non si possono sommare dataclass basate su `slots`:

```

@dataclass(slots=True)
class Person:
    name: str
    address: str
    email: str

@dataclass(slots=True)
class Employee:
    dept: str

class Worker(Person, Employee):
    pass

```

7.3.2 Iteratori

Il `for` del python è molto conciso e flessibile, si pensi a:

```

>>> List = [1, 2, 3]
>>> Set = (1, 2, 3)
>>> Dict = {'one': 1, 'two': 2}
>>> String = "asd"

>>> for element in List:
...     print(element)
...
1
2
3
>>> for element in Set:
...     print(element)
...
1
2
3
>>> for key in Dict:
...     print(key)
...
one
two
>>> for char in String:
...     print(char)
...
a
s
d

```

Quello che lo rende flessibile è il fatto di gestire in maniera standard quelli che in Python sono chiamati iterabili, ossia oggetti passibili di iterazione.

7.3.2.1 Funzionamento

Si ha che:

- *iterabile* è un oggetto che presenta un metodo `__iter__`, il quale restituisce un iteratore, oggetto che rappresenta un flusso di dati dell'iterabile considerato;
- un *iteratore* è un oggetto che rappresenta un flusso di dati e mediante il metodo `__next__` li ritorna un elemento alla volta (oppure ritorna `StopIteration` se non ve ne sono altri).

Ora vi sono due funzioni di utilità che servono per implementare il protocollo del `for`:

- `iter` prende in input un oggetto arbitrario e cerca di restituire un iteratore sui suoi dati (chiamando `__iter__`), oppure `TypeError` se non possibile;
- sull'iteratore possiamo utilizzare `next` (che chiama il `__next__`)

```

>>> s = 'abc'

```

```

>>> it = iter(s)
>>> it
<str_ascii_iterator object at 0x7fb4d44afc10>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> iter(123) # questo da errore perché un intero non è iterabile
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable

```

Quindi complessivamente, lo statement `for`:

1. chiama innanzitutto `iter()` sull'oggetto che è in ciclo (il contenitore) ottenendone un iteratore;
2. chiama via via `next()` sull'iteratore permettendo così il processo di iterazione;
3. quando non vi sono altri elementi sui quali iterare, `next()` solleva l'eccezione `StopIteration` e `for` termina.

7.3.2.2 Implementazione mediante classe

Spesso si vuole creare classi/strutture di dati che siano iterabili; per farlo occorre:

- definire una classe con metodo `__iter__` (iterabile) che ritorni un oggetto con un metodo `__next__` senza parametri (rendendo l'oggetto ritornato un iteratore).
- caso classico: una classe ha sia `__next__` che `__iter__`. In tal caso è sufficiente che `__iter__` ritorni `self`.

```

>>> class Reverse:
...     """Iterator for looping over a sequence backwards."""
...     def __init__(self, data):
...         self.data = data
...         self.index = len(data)
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.index == 0:
...             raise StopIteration
...         self.index = self.index - 1

```

```

...         return self.data[self.index]
...
>>> for char in Reverse('spam'):
...     print(char)
...
m
a
p
s

```

7.3.2.3 Implementazione mediante espressioni generatrici

Generatori semplici possono essere ottenuti mediante le espressioni generatrici, espressioni poste tra tonde (invece che quadre) che utilizzano una sintassi simile alle list comprehension. Alcuni esempi:

```

>>> squares = ((i*i for i in range(10)))
>>> type(squares)
<class 'generator'>
>>> sum(squares)
285

>>> data = 'golf'
>>> reversed = (data[i] for i in range(len(data) - 1, -1, -1))
>>> list(reversed)
['f', 'l', 'o', 'g']

```

Alcuni confronti:

- le espressioni generatrici sono compatte ma meno versatili rispetto alla definizione di un generatore
- rispetto a una list comprehension, le espressioni generatrici ritornano un iteratore che calcola il valore al bisogno; le list comprehension sono inutili o meno efficienti se si lavora con iteratori che ritornano uno stream infinito di valori o un numero molto alto

7.3.2.4 Implementazione mediante generatori

I generatori sono funzioni che creano degli iteratori:

- l'unica differenza dalle funzioni classiche è che usano `yield` quando vogliono ritornare dei dati;
- tutto quello che è possibile fare mediante la definizione di iteratori mediante classi è possibile farlo anche con i generatori; quello che rende questi ultimi interessanti è il fatto che i metodi `__iter__` e `__next__` sono generati automaticamente e complessivamente la programmazione è molto più chiara/compatta.

```

>>> def reverse(data):
...     for index in range(len(data) - 1, -1, -1):
...         yield data[index]

```

```

...
>>> # crea un iterabile e iteratore, ossia un oggetto che ha sia iter che next
>>> r = reverse('golf')
>>> r.__iter__
<method-wrapper '__iter__' of generator object at 0x7fb4d44665e0>
>>> r.__next__
<method-wrapper '__next__' of generator object at 0x7fb4d44665e0>

>>> for char in r:
...     print(char, end = ' ')
...
f l o g

```

Vediamo la differenza di funzionamento rispetto a una funzione classica:

- nel chiamare una funzione classica viene creato un namespace che contiene i dati e al `return` questo viene distrutto; una chiamata successiva alla stessa riparte con un namespace nuovo. I generatori possono essere pensati come funzioni dove il namespace non viene gettato all'uscita ma è disponibile alla successiva chiamata
- alla chiamata un generatore non ritorna un singolo valore: invece ritorna un oggetto generatore che supporta il protocollo degli iteratori. Quando si esegue `yield` il generatore restituisce l'espressione, ma a differenza di `return` l'esecuzione della funzione si sospende e le variabili locali sono preservate; alla prossima chiamata di `__next__` la funzione riprenderà l'esecuzione da capo.

7.3.2.5 Funzioni e operatori utili su iteratori

Funzioni Alcune funzioni builtin:

- su iteratori con dati logici `all` e `any` ritornano `True` rispettivamente se tutti gli elementi sono `True` o almeno uno lo è
- su iteratori con dati confrontabili, `max`, `min` ritornano l'elemento maggiore o minore, `sorted` ordina l'iteratore
- `enumerate` restituisce un oggetto involucro con un id progressivo
- `zip` prende in input più iterabili e restituisce un iteratore che genera tuple con un elemento di ciascun oggetto di partenza alla volta. Nel caso gli iterabili abbiano lunghezza diversa verrà prodotto
- per la programmazione funzionale sono utili `filter`, `map`

```

>>> all([True, True])
True
>>> any([False, False])
False
>>> max([1, 2, 10])
10

```

```
>>> A = [1,2,3]
>>> B = "letters"

>>> for a, b in zip(A, B):
...     print(a, b)
...
1 1
2 e
3 t
```

Operatori `in` e `not in`: la sintassi `X in iterator` ritorna `True` se `X` è ritrovato nell'iteratore

7.3.2.6 Il modulo `itertools`

Contiene funzioni per la creazione e gestione di iteratori

Creazione di nuovi iteratori

```
itertools.count()          ## 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.cycle([1, 2, 3, 4, 5]) ## 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 ...
itertools.repeat('abc')    ## abc, abc, abc, abc, abc, abc ...
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) ## a, b, c, 1, 2, 3
itertools.islice(range(10), 2, 8) ## 2, 3, 4, 5, 6, 7
```

Selezione

```
itertools.filterfalse(predicate, iterable)
```

Crea un iteratore che elimina elementi da un iterabile laddove un predicato ad essi applicato è falso. Lo applichiamo ad una lista come esempio:

```
>>> import itertools

>>> def selector(x):
...     return x < 5
...
>>> res = itertools.filterfalse(selector, [1, 4, 6, 4, 1])
>>> list(res)
[6]
```

```
itertools.compress(data, selectors)
```

crea un iteratore che filtra gli elementi di `data` ritornando solo quelli che valuno `True` in `selectors`

```
>>> res = itertools.compress('ABCDEF', [1,0,1,0,1,1])
>>> list(res)
['A', 'C', 'E', 'F']
```

Grouping Vediamo:

```
itertools.groupby(iter, key_func = None)
```

si ha:

- `iter` un iterabile
- `key_func` è una funzione che restituisce un id per ogni elemento dell'iterabile

`groupby`

- assume che il contenuto di `iter` sia ordinato per chiave
- mette assieme tutti gli elements dell'iterable con stessa chiave, e ritorna uno stream di tuple di due elementi, con primo elemento la chiave e secondo elemento l'iteratore su tutti gli elementi con tale chiave

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
             ...
            ]
```

```
def get_state(city_state):
    return city_state[1]
```

```
itertools.groupby(city_list, get_state) =>
('AL', iterator-1),
('AK', iterator-2),
('AZ', iterator-3), ...
```

where

```
iterator-1 =>
('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

Combinazioni e permutazioni

```
>>> list(itertools.combinations([1, 2, 3, 4, 5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
>>> list(itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2))
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 2), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5), (4, 4), (4, 5), (5, 5)]
>>> list(itertools.permutations([1, 2, 3, 4, 5]))
[(1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5), (1, 2, 4, 5, 3), (1, 2, 5, 3, 4), (1, 2, 5, 4, 3), (1, 3, 2, 4, 5), (1, 3, 2, 5, 4), (1, 3, 4, 2, 5), (1, 3, 4, 5, 2), (1, 3, 5, 2, 4), (1, 3, 5, 4, 2), (1, 4, 2, 3, 5), (1, 4, 2, 5, 3), (1, 4, 3, 2, 5), (1, 4, 3, 5, 2), (1, 4, 5, 2, 3), (1, 4, 5, 3, 2), (1, 5, 2, 3, 4), (1, 5, 2, 4, 3), (1, 5, 3, 2, 4), (1, 5, 3, 4, 2), (1, 5, 4, 2, 3), (1, 5, 4, 3, 2), (2, 1, 3, 4, 5), (2, 1, 3, 5, 4), (2, 1, 4, 3, 5), (2, 1, 4, 5, 3), (2, 1, 5, 3, 4), (2, 1, 5, 4, 3), (2, 3, 1, 4, 5), (2, 3, 1, 5, 4), (2, 3, 4, 1, 5), (2, 3, 4, 5, 1), (2, 3, 5, 1, 4), (2, 3, 5, 4, 1), (2, 4, 1, 3, 5), (2, 4, 1, 5, 3), (2, 4, 3, 1, 5), (2, 4, 3, 5, 1), (2, 4, 5, 1, 3), (2, 4, 5, 3, 1), (2, 5, 1, 3, 4), (2, 5, 1, 4, 3), (2, 5, 3, 1, 4), (2, 5, 3, 4, 1), (2, 5, 4, 1, 3), (2, 5, 4, 3, 1), (3, 1, 2, 4, 5), (3, 1, 2, 5, 4), (3, 1, 4, 2, 5), (3, 1, 4, 5, 2), (3, 1, 5, 2, 4), (3, 1, 5, 4, 2), (3, 2, 1, 3, 5), (3, 2, 1, 5, 4), (3, 2, 3, 4, 5), (3, 2, 3, 5, 4), (3, 2, 4, 3, 5), (3, 2, 4, 5, 3), (3, 2, 5, 3, 4), (3, 2, 5, 4, 3), (3, 3, 1, 4, 5), (3, 3, 1, 5, 4), (3, 3, 4, 2, 5), (3, 3, 4, 5, 2), (3, 3, 5, 2, 4), (3, 3, 5, 4, 2), (3, 4, 1, 2, 5), (3, 4, 1, 5, 2), (3, 4, 2, 1, 5), (3, 4, 2, 5, 1), (3, 4, 3, 1, 5), (3, 4, 3, 5, 1), (3, 4, 5, 1, 2), (3, 4, 5, 2, 1), (3, 5, 1, 2, 4), (3, 5, 1, 4, 2), (3, 5, 2, 1, 4), (3, 5, 2, 4, 1), (3, 5, 3, 1, 4), (3, 5, 3, 4, 1), (3, 5, 4, 1, 2), (3, 5, 4, 2, 1), (4, 1, 2, 3, 5), (4, 1, 2, 5, 3), (4, 1, 3, 2, 5), (4, 1, 3, 5, 2), (4, 1, 5, 2, 3), (4, 1, 5, 3, 2), (4, 2, 1, 3, 5), (4, 2, 1, 5, 3), (4, 2, 3, 1, 5), (4, 2, 3, 5, 1), (4, 2, 5, 1, 3), (4, 2, 5, 3, 1), (4, 3, 1, 2, 5), (4, 3, 1, 5, 2), (4, 3, 2, 1, 5), (4, 3, 2, 5, 1), (4, 3, 4, 1, 5), (4, 3, 4, 5, 1), (4, 3, 5, 1, 2), (4, 3, 5, 2, 1), (4, 4, 1, 3, 5), (4, 4, 1, 5, 3), (4, 4, 3, 1, 5), (4, 4, 3, 5, 1), (4, 4, 5, 1, 2), (4, 4, 5, 2, 1), (4, 5, 1, 2, 3), (4, 5, 1, 3, 2), (4, 5, 2, 1, 3), (4, 5, 2, 3, 1), (4, 5, 3, 1, 2), (4, 5, 3, 2, 1), (5, 1, 2, 3, 4), (5, 1, 2, 4, 3), (5, 1, 3, 2, 4), (5, 1, 3, 4, 2), (5, 1, 4, 2, 3), (5, 1, 4, 3, 2), (5, 2, 1, 3, 4), (5, 2, 1, 4, 3), (5, 2, 3, 1, 4), (5, 2, 3, 4, 1), (5, 2, 4, 1, 3), (5, 2, 4, 3, 1), (5, 3, 1, 2, 4), (5, 3, 1, 4, 2), (5, 3, 2, 1, 4), (5, 3, 2, 4, 1), (5, 3, 4, 1, 2), (5, 3, 4, 2, 1), (5, 4, 1, 2, 3), (5, 4, 1, 3, 2), (5, 4, 2, 1, 3), (5, 4, 2, 3, 1), (5, 4, 3, 1, 2), (5, 4, 3, 2, 1), (5, 5, 1, 2, 3), (5, 5, 1, 3, 2), (5, 5, 2, 1, 3), (5, 5, 2, 3, 1), (5, 5, 3, 1, 2), (5, 5, 3, 2, 1), (5, 5, 4, 1, 2), (5, 5, 4, 2, 1), (5, 5, 5, 1, 2), (5, 5, 5, 2, 1), (5, 5, 5, 3, 1), (5, 5, 5, 4, 1), (5, 5, 5, 5, 1), (5, 5, 5, 5, 2), (5, 5, 5, 5, 3), (5, 5, 5, 5, 4), (5, 5, 5, 5, 5)]
```


Prodotto cartesiano

```
>>> list(itertools.product('ABCD', 'xy'))
[('A', 'x'), ('A', 'y'), ('B', 'x'), ('B', 'y'), ('C', 'x'), ('C', 'y'), ('D', 'x'), ('D', 'y')]
```

7.3.3 Context Managers

Un context manager è un oggetto che fornisce informazioni contestuali o esecuzione automatica ad una azione ed è quello che funziona col protocollo/keyword **with**. Il più conosciuto è **open** grazie al quale **f** sarà automaticamente chiusa all'uscita dal manager (il blocco):

```
with open('file.txt') as f:
    contents = f.read()
```

Un context manager può essere implementato mediante classe o mediante generatore; la classe è meglio se vi è un tot di dati/logica da incapsulare, la funzione è meglio se abbiamo a che fare con casi più semplici.

7.3.3.1 Implementazione mediante classe

Per duplicare **open** semplicemente si programma:

```
class CustomOpen(object):
    def __init__(self, filename):
        self.file = open(filename)

    def __enter__(self):
        return self.file

    def __exit__(self, ctx_type, ctx_value, ctx_traceback):
        self.file.close()

with CustomOpen('file') as f:
    contents = f.read()
```

I due metodi speciali utilizzati da **with** sono **__enter__** e **__exit__**. Il funzionamento:

- **CustomOpen** è inizializzata con **__init__**
- **__enter__** è chiamata e qualsiasi cosa ritorni è assegnato a **f**
- quando il blocco di **with** finisce, viene chiamata **__exit__**

7.3.3.2 Implementazione mediante generatore

Bisogna utilizzare **contextmanager** dalla libreria **contextlib**:

```
from contextlib import contextmanager

@contextmanager
def custom_open(filename):
    f = open(filename)
```

```
    try:
        yield f
    finally:
        f.close()

with custom_open('file') as f:
    contents = f.read()
```

Il funzionamento:

- `custom_open` è eseguita fino a `yield`
- ritorna il controllo allo statement `with`; ciò che è stato dato da `yield` viene assegnato ad `f` (nel pezzo `as f`)
- la clausola `finally` assicura che `close` sia chiamata che vi sia stata una eccezione all'interno del blocco `with` o meno

Capitolo 8

Moduli e pacchetti

Contents

8.1	Introduzione	107
8.2	Moduli	108
8.2.1	Nome e namespace	108
8.2.2	Importazione dei moduli	108
8.2.3	Path di ricerca dei moduli	109
8.2.4	Templates	109
8.3	Pacchetti	111
8.3.1	Struttura e <code>__init__.py</code>	111
8.3.2	<code>__init__.py</code> , <code>__all__</code> e <code>import *</code> da pacchetto . . .	113
8.4	Packaging e distribuzione di pacchetti	113
8.4.1	Flow	113
8.4.2	<code>pyproject.toml</code>	113
8.4.3	Aggiornamento toolchain	114
8.4.4	Creazione del tree del pacchetto	114
8.4.5	Installare un pacchetto in modalità devel	114
8.4.6	Build di sdist e wheel	114
8.4.7	Upload a pypi	114
8.5	Altre utilità	115
8.5.1	Inserimento di script nel pacchetto	115
8.5.2	Documentazione	115
8.5.3	Testing	117
8.5.4	Timing/temporizzazione	118
8.5.5	Profiling	118
8.5.6	Altri strumenti utili	119

Per la creazione di pacchetti, la guida aggiornata si trova qui: <https://packaging.python.org/en/latest>

8.1 Introduzione

Alcune distinzioni:

modulo file `.py` che può definire classi, funzioni e variabili globali importabili da un altro modulo mediante `import`

pacchetto una directory contenente moduli e un file `__init__.py`. I pacchetti sono un modo per organizzare moduli con scopi affini.

Di base `import` applicato ad un pacchetto esegue innanzitutto `__init__.py` per eventuali configurazioni/inizializzazioni.

8.2 Moduli

8.2.1 Nome e namespace

Ogni modulo ha:

- un *nome*, contenuto nella variabile globale `__name__` che solitamente coincide con:
 - `"nomefile"` senza l'estensione `.py` se il modulo è importato mediante `import`
 - `"__main__"`, qualora il modulo sia eseguito come script attraverso `python nomefile.py`
- un *namespace*: ogni modulo ha il suo che viene utilizzato da tutte le funzioni definite in esso. Le funzioni di un modulo per riferirsi ad altri oggetti definiti nello stesso, possono evitare di precedere il nome del modulo all'oggetto richiesto.

La presenza di un nome del modulo fa sì che si possa eseguire codice a seconda che il modulo sia importato o eseguito come script. Ad esempio nel file `miomodulo.py` dopo tutte le definizioni possiamo dare

```
if __name__ == "__main__":
    checks()
```

in questo caso i `checks()` verranno eseguite solamente se eseguiamo il modulo mediante `python miomodulo.py` (altrimenti mediante `import` il nome del modulo è impostato a `miomodulo` e i `checks` non vengono eseguiti).

8.2.2 Importazione dei moduli

I moduli possono importare altri moduli per ottenerne funzionalità in due modi differenti per la gestione del namespace. Supponiamo di aver creato un file `mymodule.py` nella directory corrente che contiene la funzione `do_complicated_stuff` e la variabile `strange_var`. Possiamo comandare:

- `import mymodule <as abbreviazione>`

Viene creato un oggetto-modulo chiamato `mymodule` che contiene quanto definite; per utilizzarle

```
mymodule.do_complicated_stuff()
mymodule.strange_var
```

Specificando l'abbreviazione si crea un alias per il `mymodule` (verosimilmente più piccola e veloce da ditzeggiare)

- `from mymodule import do_complicated_stuff <as abbrev>`
`from mymodule import strange_var <as abbrev>`

Si importano solamente la funzione (o la costante) ed è possibile riferirvisi in seguito con un più veloce `do_complicated_stuff` senza specificare il nome del modulo di provenienza.

- `from mymodule import *`

Si importa tutto quello che è definito nel modulo (non è considerato buona pratica)

8.2.3 Path di ricerca dei moduli

Quando viene richiesta l'importazione di un modulo di nome `spam.py` mediante `import spam` (o simili), Python cerca nell'ordine:

1. nei moduli builtin distribuiti col linguaggio;
2. nella lista di directory specificate in `sys.path`, nell'ordine specificato:

```
import sys
sys.path
```

La variabile contiene solitamente la directory corrente al primo posto (come stringa vuota), facendole assumere priorità tra quelle del `sys.path`. Se si desidera aggiungere una directory come prima si può usare

```
sys.path.insert(0, '/path/to/mod_directory')
```

8.2.4 Templates

8.2.4.1 Modulo libreria

```
#!/usr/bin/env python3          # (1) sha bang, volendo poter eseguire anche con ./
"""                             # (2) doc: disponibile mediante nomemodulo.__doc__
Documentazione del modulo
Sommario funzionalità, classi,
funzioni, variabili.
"""

import sys                     # (3) importazione di altri moduli
import os

debug = True                   # (4) variabili globali, se necessarie

class FooClass():              # (5) dichiarazione classi
    "Foo class"
    pass

def test():                     # (6) dichiarazione funzioni (qui che possono
```

```

    "Test fun"                                #    utilizzare le classi)
    foo = FooClass()
    if debug:
        print 'ran test()'

if __name__ == '__main__':    # (7) corpo main
    test()

```

L'ultima linea controlla se il codice è stato eseguito direttamente o importato; se eseguito direttamente esegue la funzione di test.

8.2.4.2 Script della libreria

Da testare: analogo al precedente ma con `argparse` terminante con esecuzione della `main`

```

import argparse

def main():                                # (6) funzione main
    """
    Main function

    winston_sends "ciao" user::lucailgarb
    winston_sends file.pdf user::lucailgarb
    winston_sends file.png group::da_salvare
    """

    parser = argparse.ArgumentParser()
    parser.add_argument("what")
    parser.add_argument("to")
    args = parser.parse_args()
    what = args.what
    to = args.to

    # ...

if __name__ == '__main__':    # (7) corpo main
    main()

```

I benefici sono:

- si segnala a chi legge che lo script è inteso per essere eseguito da linea di comando
- ponendo codice nella funzione `main` lo isola meglio
- posso importare la funzione `main` e sarebbe come eseguirla da script da un altro file
- in alcuni casi (cicli `for`) vi sono benefici di performance nell'esecuzione di codici all'interno di funzioni

8.3 Pacchetti

8.3.1 Struttura e `__init__.py`

Al minimo, un pacchetto è una directory contenente un file `__init__.py` e i file `.py` che costituiscono i moduli.

I files `__init__.py` sono necessari per far sì che Python tratti la directory come un pacchetto;

- nel caso più semplice (considerata best practice) può essere un file vuoto;
- alternativamente si possono eseguire inizializzazioni o impostare la variabile `__all__` di cui si parla in seguito

8.3.1.1 Struttura semplice

Ad esempio, con una siffatta situazione:

```
mypackage/
|-- __init__.py
|-- module_a.py
+-- module_b.py
```

L'utilizzo (in uno script al di fuori della directory `mypackage`) può avvenire come:

```
import mypackage.module_a
```

8.3.1.2 Struttura con subpackages

In situazioni più complesse si possono prevedere subpackage, ossia subdirectory con moduli affini e un `__init__.py` per ogni directory/subdirectory. Ad esempio per un pacchetto che gestisce l'audio (di nome `sound`):

<code>sound/</code>	Top-level package
<code>__init__.py</code>	Initialize the sound package
<code>formats/</code>	Subpackage for file format conversions
<code>__init__.py</code>	
<code>wavread.py</code>	
<code>wavwrite.py</code>	
<code>aiffread.py</code>	
<code>aiffwrite.py</code>	
<code>auread.py</code>	
<code>auwrite.py</code>	
<code>...</code>	
<code>effects/</code>	Subpackage for sound effects
<code>__init__.py</code>	
<code>echo.py</code>	
<code>surround.py</code>	
<code>reverse.py</code>	
<code>...</code>	
<code>filters/</code>	Subpackage for filters
<code>__init__.py</code>	

```

equalizer.py
vocoder.py
karaoke.py
...

```

References da esterno L'importazione di funzionalità (a partire da uno script residente all'esterno di `sound`) può essere avvenire con `import` o `from`:

- `import` in `import a.b.c` ogni item tranne l'ultimo deve essere un pacchetto (directory con `__init__.py`), l'ultimo (`c`) può essere un pacchetto o un modulo
- con `from` possiamo importare moduli e funzioni, controlla che l'oggetto sia definito nel pacchetto, se sì lo importa, se no ipotizza sia un modulo e lo cerca

```

# modulo versione 1: con import è necessaria poi la full reference
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

# modulo vrs 2: con from si può usare la relative reference
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4) # from allows relative ref

# con from si possono importare anche funzioni
from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7, atten=4)

```

Intra-package references Quando i pacchetti sono strutturati in sottopacchetti si possono usare sia referenza assoluta che relativa. Ad esempio

- se `sound.filters.vocoder` necessita del modulo `sound.effects.echo` nelle sue prime linee si può usare alternativamente

```

from sound.effects import echo
from ..effects import echo

```

•

se `sound.effects.surround` necessita `sound.effects.echo` `sound.formats` o `sound.filters.equalizer`

```

from . import echo
from .. import formats
from ..filters import equalizer

```

Gli import relativi si basano sul nome del modulo corrente: dato che il nome del modulo main è sempre `__main__`, i file `.py` (moduli) che debbono essere eseguiti come modulo main (script) debbono utilizzare sempre import assoluti.

8.3.2 `__init__.py`, `__all__` e `import *` da pacchetto

Se in `sounds/effects/__init__.py` si imposta

```
__all__ = ["echo", "surround", "reverse"]
```

e si comanda

```
from sound.effects import *
```

verranno importati `echo.py`, `surround.py` e `reverse.py`.

8.4 Packaging e distribuzione di pacchetti

Qua ci si riferisce prevalentemente a <https://packaging.python.org/en/latest/flow/> e <https://packaging.python.org/en/latest/tutorials/packaging-projects/> cui vi è da fare riferimento per pratiche standard.

8.4.1 Flow

Per pubblicare un pacchetto occorre avere:

- il codice sotto git;
- preparare un file di metadati descrittivi e di istruzione di building del pacchetto. Nella maggior parte dei casi questo è il file `pyproject.toml` nella directory radice del progetto. Questo deve almeno contenere una sezione `[build-system]` che specifica il sistema di building backend adottato (`hatch` è nuovo, `setuptools` è vecchio, poi ve ne sono altri). Qui si usa `hatch`;
- effettuare la build che produce il pacchetto di sorgenti (`sdist`) e il binario (`wheel`), detti build artifacts
- fare l'upload dei build artifacts su PyPi

8.4.2 `pyproject.toml`

In `pyb/pyproject.toml` specificare le main config del repo in accordo a. Per utilizzare `hatch` come build system

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

Specificare le config rimanenti in accordo a:

- lo standard su come specificare metadati <https://packaging.python.org/en/latest/specifications/declaring-project-metadata/>;
- la documentazione del build system (`hatch`).

8.4.3 Aggiornamento toolchain

```
python3 -m pip install --upgrade pip
python3 -m pip install --upgrade hatch # building backend
python3 -m pip install --upgrade build # building frontend
python3 -m pip install --upgrade twine # upload
```

8.4.4 Creazione del tree del pacchetto

Questo crea il template di directory corretto:

```
l@m740n:~/src/pypkg$ hatch new pylb
```

Porre:

- il contenuto del pacchetto in `pylb/src/pylb`
- il gitignore per un progetto python

```
wget https://raw.githubusercontent.com/github/gitignore/main/Python.gitignore
mv Python.gitignore .gitignore
```

Ora aggiungere codice, aggiungere documentazione e test, affrontati in seguito.

8.4.5 Installare un pacchetto in modalità devel

Per effettuare prove veloci su un nuovo pacchetto conviene installarlo in modalità editabile; questo permette di modificare il sorgente senza dover reinstallare il pacchetto per vedere i risultati in via di test.

Per farlo spostarsi nella directory del pacchetto e comandare

```
pip install -e .
```

8.4.6 Build di sdist e wheel

Se si vuole creare entrambi, semplicemente

```
l@m740n:~/src/pypkg$ python3 -m build pylb
```

che creerà il pacchetto dist (`tar.gz`) e il pacchetto wheel (`whl`) nella subdirectory `dist`.

8.4.7 Upload a pypi

Occorrerà utilizzare le info di login di pypi ovviamente:

```
l@m740n:~/src/pypkg$ twine upload pylb/dist/*
Uploading distributions to https://upload.pypi.org/legacy/
```

8.5 Altre utilità

8.5.1 Inserimento di script nel pacchetto

Per inserire script (es che rendano facilmente disponibili funzionalità del pacchetto da linea di comando) un template da seguire:

- creare una cartella `scripts` sotto `src/pylb`
- creare un modulo col nome dell'eseguibile desiderato (non obbligatorio ma comodo), ad esempio

```
l@ambrogio:~/src/pypkg/pylb$ cat src/pylb/scripts/pylbtestapp.py
def main():
    print("Hi There! This is pylbtestapp.")
```

- inserire in `pyproject.toml` una sezione e riga del genere

```
[project.scripts]
pylbtestapp = "pylb.scripts.pylbtestapp:main"
```

All'installazione del pacchetto, lo script verrà posto in `.local/bin`.

8.5.2 Documentazione

Si fa utilizzo di `sphinx` e si pubblica su <https://readthedocs.org/>. I tutorial da considerare sono <https://packaging.python.org/en/latest/tutorials/creating-documentation/> e <https://docs.python-guide.org/writing/documentation/>.

8.5.2.1 Setup

Iniziamo ad installare `sphinx`

```
python3 -m pip install --upgrade sphinx # documentazione
```

Creiamo la directory di documentazione e facciamo partire il tutto

```
l@m740n:~/src/pypkg$ mkdir pylb/docs
l@m740n:~/src/pypkg$ cd pylb/docs
l@m740n:~/src/pypkg/pylb/docs$ sphinx-quickstart
```

Questo farà domande sul progetto per andare a creare il file `index.rst` e un `conf.py` (configurabili), più un `Makefile` di servizio.

8.5.2.2 Doc-writing e reStructuredText

`sphinx` converte restructured text ad altri linguaggi di markup, e utilizza reStructuredText come linguaggio di markup. Per la sintassi riferirsi a <https://www.sphinx-doc.org/en/master/usage/restructuredtext/> per le convenzioni di documentazione a quelle del progetto Numpy.

8.5.2.3 Building

Setup autobuilding API Per fare sì che la documentazione di funzioni/classi etc, venga generata automaticamente occorre utilizzare l'estensione `autodoc`. Modificare `conf.py` come segue:

- aggiungere il path della libreria nel `sys.path` all'inizio di `conf.py`. Il package deve essere importabile per poter essere elaborato:

```
import os
import sys
sys.path.insert(0, os.path.abspath('../src'))
```

- modificare per aggiungere le seguenti estensioni sotto general configuration:

```
extensions = [
    'sphinx.ext.autodoc',      # generazione automatica api
    'sphinx.ext.viewcode',    # aggiungi link ipertestuali al codice
    'sphinx.ext.napoleon'     # supporta sintassi a-la numpy
]
```

- volendo si può cambiare il tema html installandolo

```
pip install --user sphinx_rtd_theme
```

e modificando la linea del tema html (sostituendo `alabaster` di default)

```
html_theme = 'sphinx_rtd_theme'
```

Al termine del setup per preparare la documentazione di funzioni etc:

```
l@m740n:~/src/pypkg/pylb/docs$ sphinx-apidoc -f ../src/pylb/ -o .
Creating file ./pylb.rst.
Creating file ./pylb.experiments.rst.
Creating file ./modules.rst.
```

Aggiungere `modules.rst` in `index.rst`

```
Welcome to pylb's documentation!
=====
```

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   modules
```

Build definitivo Per buildare definitivamente la documentazione complessivamente si usa `make` con il formato di interesse (nella cartella `docs`). Per l'html

```
l@m740n:~/src/pypkg/pylb/docs$ make html
The HTML pages are in _build/html.
l@m740n:~/src/pypkg/pylb/docs$ firefox _build/html/index.html
```

Per il latex

```
l@740n:~/src/pypkg/pylb/docs$ make latex
The LaTeX files are in _build/latex.
Run 'make' in that directory to run these through (pdf)latex
(use `make latexpdf' here to do that automatically).
l@740n:~/src/pypkg/pylb/docs$ make latexpdf
l@740n:~/src/pypkg/pylb/docs$ okular _build/latex/pylb.pdf
```

8.5.2.4 Setup di readthedocs

Seguire il tutorial per importare il progetto e generare/hostare automaticamente la documentazione

8.5.3 Testing

Il testing di hatch è fatto mediante `pytest`:

- porre tutte i test nella directory `tests` seguendo una struttura directory simile a `src` e con `__init__.py` in ogni sottodirectory. Ad esempio per fare i test del modulo `pytest` in `experiments`, creiamo la cartella `experiments` sotto `tests` e aggiungiamo il file `test_pytest.py` in essa, oltre a `__init__.py`

```
src/pylb
|-- __init__.py
|-- __about__.py
|-- experiments
|   |-- __init__.py
|   |-- pytest.py
|   `-- sphinx.py
`-- scripts
    |-- __init__.py
    `-- pylbtestapp.py
tests/
|-- experiments
|   |-- __init__.py
|   `-- test_pytest.py
`-- __init__.py
```

Per l'esempio a mano programiamo i due file come segue. Il codice `src/pylb/experiments/pytest.py`

```
def add(a, b):
    return a+b
```

mentre il codice di test

```
from pylb.experiments.pytest import add

def test_add():
    assert add(1,2) == 3
```

- Infine per comandare il testing

```
l@ambrogio:~/pylb$ hatch run test
===== test session starts =====
platform linux -- Python 3.11.2, pytest-7.3.1, pluggy-1.0.0
rootdir: /home/l/.src/pypkg/pylb
collected 1 item

tests/experiments/test_pytest.py

===== 1 passed in 0.00s =====
```

Lo unit testing sarà sviluppato maggiormente nel prossimo paragrafo

8.5.4 Timing/temporizzazione

Per misurare il tempo

- manualmente usiamo `time.perf_counter`

```
import time
start = time.perf_counter()
# operations ...
stop = time.perf_counter()
stop - start # difference in seconds
```

- in maniera più strutturata (non basandosi solo su una singola esecuzione) usando il magic command `timeit` di `ipython`, il quale esegue codice in maniera ripetuta e stampa statistiche descrittive. Il timing può essere fatto con o senza istruzioni di setup (tenute nella conta):

- in modalità a singola riga si temporizza la riga (si possono spezzare più istruzioni con `;`)

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]
1000 loops, best of 3: 325 µs per loop
```

- in modalità cella la prima riga è usata come codice di setup (eseguito ma non temporizzato) e il corpo è temporizzato. Per questo si usa il doppio `%`

```
In [9]: %%timeit
...: L = []
...: for n in range(1000):
...:     L.append(n ** 2)
...:
```

8.5.5 Profiling

Il profiling serve per individuare le macrosezioni di codice dove si spende più tempo e/o si usa più memoria. Anche qui usiamo le funzionalità di `ipython`

8.5.5.1 Tempo

<https://wesmckinney.com/book/ipython.html> <https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html>

8.5.5.2 Memoria

<https://wesmckinney.com/book/ipython.html> <https://jakevdp.github.io/PythonDataScienceHandbook/01.07-timing-and-profiling.html>

8.5.6 Altri strumenti utili

Controllo del codice Effettuarlo mediante il tool `flake8`

```
flake8 project_dir
```


Capitolo 9

Testing

Contents

9.1	Introduzione e concetti	121
9.1.1	Tipologie di testing	121
9.1.2	Test driven development	122
9.2	unittest	122
9.2.1	Test del valore ritornato	122
9.2.2	Test eccezioni	124
9.2.3	Test fixtures	125

9.1 Introduzione e concetti

9.1.1 Tipologie di testing

Si divide classicamente il testing in:

unit testing di singole funzionalità (es funzione/classe)

functional test di funzionalità più generali/complessive (derivante dall'interazione di più funzioni di base)

regression : test che l'output di un programma non cambi a successive versioni (a meno che ciò non sia intenzionale). Basati su output dell'esecuzione di versioni passate (validate ad occhio) o di programmi benchmark

I test debbono:

- essere specifici, indipendenti e svolti in maniera automatica
- testare in caso di input corretto, l'output corretto
- testare in caso di input incorretto, la gestione corretta delle eccezioni
- dovrebbero teoricamente testare tutto l'input possibile

9.1.2 Test driven development

Se l'impostazione dei test sulle funzionalità avvenga *prima* che queste funzionalità vengano implementate abbiamo il *test driven development* (TDD).

La filosofia del TDD è:

1. scrivi test completi che falliscono
2. scrivi codice fino a farli passare

Vantaggi sono:

- si specifica a priori tutti i comportamenti specifici che il nostro software deve avere/rispettare
- evitano l'overcoding: una volta che i test passano siamo a posto e non vi è bisogno di aggiungere altro
- in sede di refactoring possiamo lavorare tranquillamente garantendoci che le nuove versioni si comportino come le vecchie

9.2 unittest

Il modulo classico per lo unit testing in Python è `unittest`. Supponiamo di avere un modulo di nome `testme` e lo sottoponiamo a test nel modulo `tests`.

9.2.1 Test del valore ritornato

Nel `tests.py`, ad un livello minimale, abbiamo:

```
import testme
import unittest

class add2Tests(unittest.TestCase):

    known_value = ((1,3), (2,4))

    def test_right_input(self):
        '''add2 should add 2 to the proper input'''
        for _input, _output in self.known_value:
            result = testme.add2(_input)
            self.assertEqual(_output, result)

if __name__ == '__main__':
    unittest.main()
```

Alcune peculiarità:

- per creare un test case (ovvero un gruppo di test legati in qualche modo tra loro e identificati dal nome della classe), creiamo una classe che eredita dalla classe `unittest.TestCase`, la quale definisce diversi metodi utili per lo unit testing

- ogni metodo della classe definita costituisce un singolo test ed è identificato dal nome della funzione (quindi anche qui l'importanza di nomi esplicativi): nel caso di sopra il test che abbiamo impostato si verifica che l'output fornito dalla funzione sia uguale a quello specificato come corretto in `known_value` (la di cui correttezza di contenuto deve essere verificata/validata a mano);
- la classe `TestCase` fornisce metodi di utilità generale per il testing: qui abbiamo utilizzato `assertEqual` che si occupa di verificare che due valori siano uguali. Altri metodi utili sono `assertTrue` e `assertFalse`;
- `unittest.main` cerca in tutti i simboli del namespace globale quali sono le classi che ereditano da `unittest.TestCase`; per ciascuna di queste classi trova tutti i metodi che di nome iniziano con `test` e per ognuno di essi istanzia un nuovo oggetto (per creare un contesto pulito ogni volta) ed esegue la singola funzione.

Per ogni singolo test di ogni test suite:

1. stampa la docstring ed esegue il codice
2. dice se il test è **passato** (esecuzione completata, risultato corretto), **fallito** (esecuzione completata, risultato incorretto) o da **errore** (esecuzione non completata)
3. nel caso di problemi viene stampata la traceback
4. fa alcune statistiche complessive

In seguito, nel file `testme.py`:

```
def add2(x):
    pass
```

In questa fase:

- definiamo solamente l'api della funzione
- partiamo con una bozza
- ci assicuriamo che i test falliscano: i test dovrebbero fallire perchè si ritorna `None` (che è diverso da 3)

Per effettuare i test basta eseguire il file `tests.py`, se si vuole *verbose* con l'opzione `-v` specificata alla fine; se effettuiamo i test effettivamente quello che otteniamo è:

```
l@740n:~/cs/python/code$ python3 tests.py -v
test_right_input (__main__.add2Tests)
add2 should add 2 to the proper input ... FAIL
```

```
=====
FAIL: test_right_input (__main__.add2Tests)
add2 should add 2 to the proper input
```

```
-----
Traceback (most recent call last):
```

```
  File "tests.py", line 12, in test_right_input
```

```

        self.assertEqual(_output, result)
AssertionError: 3 != None

```

```

-----
Ran 1 test in 0.000s

```

```

FAILED (failures=1)

```

che ci indica il fallimento dei test come preventivato. Se però ridefiniamo `add2` come

```

def add2(x):
    return x + 2

```

allora abbiamo

```

l@740n:~/cs/python/code$ python tests.py -v
test_right_input (__main__.add2Tests)
add2 should add 2 to the proper input ... ok

```

```

-----
Ran 1 test in 0.000s

```

```

OK

```

Che ci indica corretto test.

9.2.2 Test eccezioni

Possiamo evolvere l'esempio considerando che la nostra funzione possa accettare solamente valori numerici e in caso di input che non sia tale si deve comportare in maniera appropriata, ovvero deve fallire sollevando una eccezione. In questo caso `unittest.TestCase` fornisce il metodo `assertRaises` che prende in input:

- l'oggetto eccezione di interesse
- la funzione
- i parametri da dare alla funzione

Ad esempio se in Python si somma 3 ed `'a'` viene restituito `TypeError`, possiamo assicurarci che ciò avvenga nella nostra funzione definendo un test del genere (sempre all'interno della classe `add2Tests`)

```

not_numerics = ('a', 'b')

def test_not_numeric_input(self):
    ''' not numeric input should raise TypeError '''
    for i in self.not_numerics:
        self.assertRaises(TypeError, testme.add2, i)

```

Se volessimo che la funzione gestisse solo interi o simili potremmo sollevare una eccezione custom nel caso contrario dovremmo ridefinire `testme` come segue:

```

class add2NotHandledType(TypeError):
    pass

def add2(x):
    if not isinstance(x, int):
        raise add2NotHandledType('Only int handled')
    return x + 2

```

e il testing complessivamente come

```

import testme
import unittest

class add2Tests(unittest.TestCase):

    known_value = ((1,3), (2,4))

    def test_right_input(self):
        '''add2 should add 2 to the proper input'''
        for _input, _output in self.known_value:
            result = testme.add2(_input)
            self.assertEqual(_output, result)

    not_integers = ('a', 1.1)

    def test_not_numeric_input(self):
        ''' not numeric input should raise TypeError '''
        for i in self.not_integers:
            self.assertRaises(testme.add2NotHandledType, testme.add2, i)

if __name__ == '__main__':
    unittest.main()

```

All'esecuzione abbiamo:

```

l0m740n:~/cs/python/code$ python3 tests.py -v
test_not_numeric_input (__main__.add2Tests)
not numeric input should raise TypeError ... ok
test_right_input (__main__.add2Tests)
add2 should add 2 to the proper input ... ok

```

```

-----
Ran 2 tests in 0.000s

```

OK

9.2.3 Test fixtures

Potremmo essere interessati a impostare delle **test fixtures** ovvero due funzioni appartenenti alla classe del test case, obbligatoriamente di nome **setUp** e **tearDown**, che vengono utilizzate per predisporre (pre) e pulire (post) .

Il funzionamento diviene così: per ogni metodo di ogni test case

- istanzia un oggetto di test
- esegui `setUp`
- esegui il metodo di test
- esegui `tearDown`

Un esempio

```
class Test(unittest.TestCase):
    def setUp(self):
        self.seq = range(0, 10)
        random.shuffle(self.seq)

    def tearDown(self):
        del self.seq

    def test_basic_sort(self):
        self.seq.sort()
        self.assertEqual(self.seq, range(0, 10))
```

Parte II

Scientific Stack

Capitolo 10

Numpy

Contents

10.1	L'ndarray	130
10.1.1	Creazione e copia	130
10.1.2	Tipi (dtype): coercizione e testing	131
10.1.3	Forma, dimensioni e reshape	132
10.2	Indexing	134
10.2.1	Array unidimensionali	134
10.2.2	Array multidimensionali	135
10.2.3	Subarray come viste vs copie	138
10.2.4	Assegnazione e unicità degli indici	139
10.3	Elaborazioni di array	139
10.3.1	Inserimento/rimozione elementi (insert , delete)	139
10.3.2	Aritmetica vettorizzata	139
10.3.3	Operazioni insiemistiche	140
10.3.4	Concatenazione (concatenate , vstack , hstack)	140
10.3.5	Splitting (split , vsplit , hsplit)	141
10.3.6	Ripetizione/binding (repeat , tile)	142
10.3.7	Sorting/ordine (sort , argsort)	143
10.4	Universal functions	144
10.4.1	Introduzione	144
10.4.2	Metodi delle ufunction (reduce , accumulate , outer)	144
10.4.3	Creazione di ufunctions	146
10.5	Broadcasting	147
10.6	Altri argomenti	151
10.6.1	Lavorare con booleani	151
10.6.2	Array di stringhe	152

Il template per l'importazione è:

```
>>> import numpy as np
```

Fornisce:

- l'oggetto `ndarray`, un array multidimensionale efficiente, costruito come puntatore a dati in memoria;
- *universal functions*: funzioni per operare su tutti gli elementi di un array;
- strumenti per integrare codice scritto in C, C++ e Fortran

10.1 L'ndarray

L'ndarray è un array ad n dimensioni di dati omogenei composto da:

- un puntatore a dati in memoria;
- un attributo `dtype` che definisce il tipo di dato;
- un attributo `shape`, tuple che definisce la struttura dell'array.

10.1.1 Creazione e copia

Si usa la funzione `array` passando dati di tipo sequenza (liste, tuple ecc)

```
>>> # creazione a partire da una lista, in varie shape
>>> np.array([ 1, 2, 3]) # vector
array([1, 2, 3])
>>> x = np.array([[1, 2, 3], [4, 5, 6]]) # matrix
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
```

Altre funzioni di convenienza per la generazione rapida di array:

```
>>> # Creazione rapida di dati
>>> np.arange(0, 20, 2) # simile a range(), da 0 a 20 a step di 2
array([ 0, 2, 4, 6, 8, 10, 12, 14, 16, 18])
>>> np.linspace(0, 1, 5) # interpolazione lineare
array([0. , 0.25, 0.5 , 0.75, 1. ])
>>> np.full(5, 2) # vettore riempito di 2
array([2, 2, 2, 2, 2])
>>> np.zeros(10) # array di zero
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> np.ones((3, 5)) # array 3x5 di uno
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
>>> np.eye(3) # matrice identità 3x3
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

>>> # creazione rapida sfruttando altre shape
>>> np.zeros_like(x) # array di 0 della stessa shape di x (np.ones_like)
array([[0, 0, 0],
       [0, 0, 0]])
```

```

>>> # Generazione casuale
>>> np.random.random((3, 3)) # array 3x3 con valori casuali uniformi 0-1
array([[0.44434511, 0.96048937, 0.5112991 ],
       [0.79614811, 0.23223604, 0.67674782],
       [0.46016738, 0.15008389, 0.26099215]])
>>> np.random.randint(0, 10, (3, 3)) # matrice 3x3 con interi nell'intervallo [0,10 )
array([[5, 2, 3],
       [2, 9, 3],
       [9, 7, 3]])
>>> np.random.normal(0, 1, 5) # da normale mu=0, sd=1
array([ 1.95599954,  1.63469539,  1.79310349, -0.41927804, -0.532048  ])

```

Referenza vs copia di array Se si assegna una variabile array ad un'altra variabile si crea un nuovo riferimento, non una copia: ossia non viene effettuata una copia di dati ma si creano due nomi che puntano alla stessa memoria:

```

>>> x = np.arange(10)
>>> y = x
>>> x[2] = -9999
>>> y
array([  0,    1, -9999,    3,    4,    5,    6,    7,    8,
        9])
>>> y[0] = -111
>>> x
array([ -111,    1, -9999,    3,    4,    5,    6,    7,    8,
        9])

```

Se necessario creare una copia per evitare di modificare l'originale utilizzare il metodo `copy`:

```

>>> x = np.arange(3)
>>> x_copy = x.copy()
>>> x_copy[0] = -999
>>> x
array([0, 1, 2])

```

10.1.2 Tipi (dtype): coercizione e testing

I tipi di dato, chiamati `dtype`, sono riportati in tabella 10.1: per utilizzarli in creazione dell'array specificare l'omonimo parametro in uno di questi due modi possibili:

```

>>> np.zeros(10, dtype = 'int16')
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int16)
>>> np.zeros(10, dtype = np.float64)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

```

Coercizione del dtype Si usa il metodo `astype` sull'array da convertire

dtype	Descrizione
bool	Boolean (True or False) stored as a byte
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t ; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

Tabella 10.1: dtypes di numpy

```
>>> a = np.zeros(10, dtype = np.float64)
>>> b = a.astype(np.int8)
>>> b
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int8)
```

Test del tipo I tipi numpy hanno una gerarchia per la quale ad esempio gli interi derivano dalla classe genitrice **np.integer** mentre i numeri a virgola mobile da **np.floating**.

Per il testing si possono usare queste classi in **np.issubdtype**.

```
>>> np.issubdtype(a.dtype, np.integer)
False
>>> np.issubdtype(a.dtype, np.floating)
True
```

10.1.3 Forma, dimensioni e reshape

10.1.3.1 Forma e dimensioni

Ogni array ha attributo:

- **ndim**: numero di dimensioni
- **shape**: numero di elementi per ciascuna dimensione
- **size**: numero di elementi complessivi
- **nbytes**: memoria occupata in bytes (dipendente dal dtype adottato, investigabile con **itemsize**)

```
>>> z1 = np.zeros(shape = 3) # vector
>>> z2i = np.zeros(shape = (3,4), dtype = np.int8) # matrix
```

```

>>> z2f = np.zeros(shape = (3,4), dtype = np.float32)
>>> z3 = np.zeros(shape = (3,4,5)) # geneneral array

>>> z2i.ndim
2
>>> z2i.shape
(3, 4)
>>> z2i.size
12
>>> z2i.nbytes
12
>>> z2f.nbytes
48
>>> z2i.itemsize
1
>>> z2f.itemsize
4

```

10.1.3.2 Reshaping

Per modificarla la forma si può assegnare all'attributo `shape` o utilizzare il metodo `reshape` (ricordandosi di salvare); `reshape` ritorna una vista sui dati e non effettua copie, se non necessario.

```

>>> a = np.arange(4) ## assegnare all'attributo shape (C mode/row major di default)
>>> a
array([0, 1, 2, 3])
>>> a.shape = (2,2)
>>> a
array([[0, 1],
       [2, 3]])

>>> b = a.reshape((4,1)) ## uso di reshape
>>> b
array([[0],
       [1],
       [2],
       [3]])

>>> c = a.reshape((4,)) ## tornare ad un array a una dimensione di 4
>>> c
array([0, 1, 2, 3])

>>> d = a.reshape(-1) ## stessa cosa ma ancora più comodo
>>> d
array([0, 1, 2, 3])

```

10.1.3.3 Ravelling/flattening

Altri due metodi per tornare ad una struttura dati unidimensionale sono:

- `flatten` fa il flattening producendo sempre una copia dell'array di partenza
- `ravel` fa il flattening ma non produce una copia dei dati, bensì una vista dell'array originale; se si modifica l'array ritornato da `ravel` si potrebbero modificare gli elementi dell'array originale.
`ravel` è spesso più veloce ma bisogna essere più attenti con le modifiche.

```
>>> b.flatten()
array([0, 1, 2, 3])
>>> b.ravel()
array([0, 1, 2, 3])
```

10.2 Indexing

Serve per accedere in lettura e scrittura agli elementi di un array.

10.2.1 Array unidimensionali

Si può specificare tra quadre alternativamente:

- lo slicing `start:stop:step` (con default rispettivamente a, 0, dimensione e 1) similmente ai dati builtin

```
>>> x = np.arange(10)
>>> x[1]
np.int64(1)
>>> x[1] = -1
>>> x[3:5] = [5, 4]
>>> x[:4:2] = 0
>>> x[-2] = 1
>>> x
array([ 0, -1,  0,  5,  4,  5,  6,  7,  1,  9])
```

Se `step` è negativo si procede in senso inverso (i default di `start/stop` vengono invertiti: per `start` la dimensione dell'array, per `stop` 0)

```
>>> x[::-1] # reverse an array
array([ 9,  1,  7,  6,  5,  4,  5,  0, -1,  0])
>>> x[5::-2] # reversed from index 5 backward
array([ 5,  5, -1])
```

- **liste e array numerici**; nel secondo caso la struttura ritornata ha lo stesso `shape` dell'indice (non del dato indicizzato):

```
>>> # lista
>>> select = [1, 2, 4, 5]
>>> x[select] # x[np.array(select)] alternativamente
array([-1,  0,  4,  5])

>>> # array con shape (2x2)
>>> ind = np.array([[2, 1],
```

```
... [4, 7]])
>>> x[ind]
array([[ 0, -1],
       [ 4,  7]])
```

- **liste/array logici**

```
>>> x = np.array([1, 2, 3])
>>> sell = [True, False, True]
>>> sela = np.array(sell)
>>> x[sell]
array([1, 3])
>>> x[sela]
array([1, 3])
```

10.2.2 Array multidimensionali

Tra quadre, separate da virgola, porre qualcosa di utilizzabile (slice, liste, array):

- **interi:** permettono di selezionare un singolo elemento

```
>>> # bidimensionale: [righe, colonne]
>>> x = np.arange(12).reshape(3,4)
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> x[0, 0] # singolo elemento
np.int64(0)
```

- **slicing**

```
>>> x[:2, 1:] # slicing: sino riga 2 esclusa, colonna 1 inclusa e sgg
array([[1, 2, 3],
       [5, 6, 7]])
>>> x[::-1, ::-1] # reverse
array([[11, 10,  9,  8],
       [ 7,  6,  5,  4],
       [ 3,  2,  1,  0]])
```

L'accesso a **righe/colonne intere** avviene combinando indici numerici/logici e slicing vuota (:). Se un indice non è specificato, si prendono tutti i dati su quella dimensione:

```
>>> # se si fornisce un solo indice alla struttura multidimensionale
>>> # viene interpretato nel primo asse/dimensione
>>> x[0] # prima riga
array([0, 1, 2, 3])

>>> # prima riga the proper way
>>> # x[0, ] # this works
>>> x[0, :]
```

```
array([0, 1, 2, 3])
>>> x[0, ::]
array([0, 1, 2, 3])
```

```
>>> # prima colonna
>>> # x[, 0] # this gives error
>>> x[:, 0]
array([0, 4, 8])
>>> x[:, 0]
array([0, 4, 8])
```

- **liste e array** in questo caso per l'accoppiamento degli indici funziona il *broadcasting*:

- se utilizziamo due array unidimensionali verrà restituita la selezione in parallelo
- se un indice è un array multidimensionale (vettore riga o colonna) otteniamo una struttura a due dimensioni (ogni valore di riga è matchato con ogni vettore colonna così come avviene nel *broadcasting* delle operazioni aritmetiche)

```
>>> x = np.arange(12).reshape(3,4)
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
>>> x[[0,1], [1,2]] # liste: elemento (0, 1) ed elemento (1, 2) della x
array([1, 6])
```

```
>>> row = np.array([0, 1, 2]) # array: elementi (0, 2), (1,1) e (2,3) (broadcasting)
>>> col = np.array([2, 1, 3])
>>> x[row, col]
array([ 2,  5, 11])
```

```
>>> rowt = row.reshape(3,1) # array2
>>> rowt
array([[0],
       [1],
       [2]])
>>> col
array([2, 1, 3])
>>> (rowt * col).shape # struttura degli indici post broadcast?
(3, 3)
>>> x[rowt, col]
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

Nell'indexing di liste e array la struttura ritornata riflette la shape degli indici post broadcast, non la shape dell'array indicizzato.

- **indexing logico** (*boolean masking*): la selezione di righe/colonne mediante liste/array logici si fa normalmente

```
>>> x = np.arange(12).reshape(3,4)
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> # selezione colonne (prima e seconda)
>>> c1 = [True, True, False, False]
>>> ca = np.array(c1)
>>> x[:, c1]
array([[0, 1],
       [4, 5],
       [8, 9]])
>>> x[:, ca]
array([[0, 1],
       [4, 5],
       [8, 9]])

>>> # selezione righe (prima e terza)
>>> r1 = [True, False, True]
>>> ra = np.array(r1)
>>> x[r1, :]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
>>> x[ra, :]
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

Se il `ndim` dell'oggetto passato come indice coincide con quello indicizzato, viene ritornato un 1-dimensional array riempito con gli elementi corrispondenti a `True`.

Questo può essere usato per masking. Ad esempio:

```
>>> # esempio 1: masking
>>> b = np.arange(9).reshape(3, 3)
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> b > 4
array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]])
>>> b[b > 4]
array([5, 6, 7, 8])

>>> # esempio 2: selezionare intersezione di righe e colonne
>>> rar = ra.reshape(3,1)
>>> rar
```

```

array([[ True],
       [False],
       [ True]])
>>> ca
array([ True,  True, False, False])

>>> # x[rar, ca] # doesnt work
>>> # x[ra, ca] # doesnt work

>>> rar * ca          # nel caso di logici questo è utile nell'indicare la selez
array([[ True,  True, False, False],
       [False, False, False, False],
       [ True,  True, False, False]])
>>> x[rar * ca]       # black magic come in b>4, ma non mantiene la struttura :(
array([0, 1, 8, 9])
>>> x[np.ix_(ra, ca)] # per mantenere la struttura usare quanto ritornato da np.
array([[0, 1],
       [8, 9]])

```

10.2.3 Subarray come viste vs copie

I subarray generati

- con *slicing/tuple* sono **viste**: pertanto eventuali modifiche si ripercuoteranno sull'array/struttura di base indipendentemente da dove sono state effettuate;
- con *array/liste* sono **copie**

```

>>> x = np.arange(10)          # subarray con slice sono una vista
>>> x_slice = x[5:8]
>>> x_slice[2] = 999
>>> x
array([ 0,  1,  2,  3,  4,  5,  6, 999,  8,  9])

>>> x = np.arange(10)          # subarray con lista sono copia
>>> x_slice = x[[5, 6, 7]]
>>> x_slice[2] = 999
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> x = np.arange(4)           # subarray con array sono copia
>>> ind = np.array([True, True, False, False])
>>> x_slice = x[ind]
>>> x_slice[:] = 999
>>> x
array([0, 1, 2, 3])

```

10.2.4 Assegnazione e unicità degli indici

Se non si desiderano comportamenti inattesi, prestare attenzione all'unicità degli indici, in quanto se un indice è ripetuto l'assegnazione sarà effettuata molteplici volte, come i seguenti esempi mostrano

```
>>> x = np.zeros(10, dtype = np.int_)

>>> # primo esempio
>>> x[[0,0]] = [4, 6]
>>> # qui si ha x[0] = 4 e poi x[0] = 6
>>> x
array([6, 0, 0, 0, 0, 0, 0, 0, 0, 0])

>>> # secondo esempio
>>> i = [2,2,4,4,6,6,6,6,6,6]
>>> x[i] += 1
>>> x
array([6, 0, 1, 0, 1, 0, 1, 0, 0, 0])
>>> # anche questo non intuitivo, si pensava la posizione 6 fosse
>>> # aumentato tante volte, ma così non è per cazzi suoi
>>> # (vedi vanderplas fancy indexing nel caso, ma anche chi se ne ciava)
```

10.3 Elaborazioni di array

10.3.1 Inserimento/rimozione elementi (insert, delete)

```
>>> a = np.array([1, 2, 3]) # any change need to be assigned to be saved
>>> np.insert(a, 1, 5)      # insert element in an array
array([1, 5, 2, 3])
>>> np.delete(a, [1])      # remove item from an array
array([1, 3])
```

10.3.2 Aritmetica vettorizzata

Le operazioni aritmetiche sono vettorizzate per cui è possibile fare

```
>>> x = np.arange(4)
>>> x
array([0, 1, 2, 3])

>>> ## Operazioni aritmetiche
>>> -x      # unary negation
array([ 0, -1, -2, -3])
>>> x + 1   # addition
array([1, 2, 3, 4])
>>> x - 5   # subtraction
array([-5, -4, -3, -2])
>>> x * 2   # multiplication
array([0, 2, 4, 6])
>>> x ** 2  # power
```

Funzione	Descrizione
<code>unique(x)</code>	sorted unique elements in x
<code>intersect1d(x, y)</code>	sorted common elements in x and y
<code>union1d(x, y)</code>	sorted union of elements
<code>in1d(x, y)</code>	boolean array if each element of x is contained in y
<code>setdiff1d(x, y)</code>	set difference (elements in x that are not in y)
<code>setxor1d(x, y)</code>	set symmetric differences (elements in one of the arrays, but not both)

Tabella 10.2: Operazioni insiemistiche tra array

```

array([0, 1, 4, 9])
>>> x / 2 # division
array([0. , 0.5, 1. , 1.5])
>>> x // 2 # floor division (e.g., 3 // 2 = 1)
array([0, 0, 1, 1])
>>> x % 2 # modulus remainder
array([0, 1, 0, 1])
>>> -(0.5*x + 1) ** 2 # misc expression
array([-1. , -2.25, -4. , -6.25])

>>> ## Comparazione
>>> x == 2
array([False, False,  True, False])
>>> x != 2
array([  True,  True, False,  True])
>>> x > 2
array([False, False, False,  True])
>>> x >= 2
array([False, False,  True,  True])
>>> x < 2
array([  True,  True, False, False])
>>> x <= 2
array([  True,  True,  True, False])

```

10.3.3 Operazioni insiemistiche

Per effettuare operazioni di tipo insiemistico le funzioni sono quelle riportate in tabella 10.2.

10.3.4 Concatenazione (concatenate, vstack, hstack)

`numpy.concatenate` prende una tupla o una lista di array e li unisce

```

>>> x = np.array([1, 2, 3]) # unidimensionale
>>> y = np.array([3, 2, 1])
>>> z = [99, 99, 99]
>>> np.concatenate([x, y, z])
array([ 1,  2,  3,  3,  2,  1, 99, 99, 99])

>>> x = np.array([[1, 2, 3], [4, 5, 6]]) # bidimensionale
>>> y = np.array([[7, 8, 9], [10, 11, 12]])

```

```
>>> np.concatenate([x, y])           # axis=0 default
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> np.concatenate([x, y], axis=1)
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

Per con array di dimensioni miste `numpy.hstack` e `numpy.vstack` tornano comode (per fare stack sulla prima o seconda dimensione)¹:

```
>>> x = np.array([1, 2, 3])
>>> grid = np.array([[9, 8, 7],
...                  [6, 5, 4]])
>>> y = np.array([[99],
...               [99]])

>>> np.hstack([grid, y])             # horizontally stack the arrays
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
>>> np.vstack([x, grid])             # vertically stack the arrays
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

10.3.5 Splitting (split, vsplit, hsplit)

`numpy.split` suddivide un array in molteplici array. Si usano passando una lista di indici per indicare dove fare i tagli:

```
>>> x = [1, 2, 3, 99, 99, 3, 2, 1]    # split di array
>>> x1, x2, x3 = np.split(x, [3, 5])
>>> x1
array([1, 2, 3])
>>> x2
array([99, 99])
>>> x3
array([3, 2, 1])

>>> x = np.arange(10).reshape(5,2)    # di matrice
>>> x
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> x1, x2, x3 = np.split(x, [1, 3])
>>> x1
array([[0, 1]])
```

¹Analogamente `np.dstack` effettuerà lo stack sulla terza dimensione.

```
>>> x2
array([[2, 3],
       [4, 5]])
>>> x3
array([[6, 7],
       [8, 9]])
```

`hsplit/vsplit` sono funzioni di convenienza per splittare sulla prima o seconda dimensione, specularmente a `vstack/hstack`²

```
>>> x = np.arange(16).reshape((4, 4))
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])

>>> upper, lower = np.vsplit(x, [2])    # vertical split
>>> upper
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> lower
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])

>>> left, right = np.hsplit(grid, [2]) # horizontal split
>>> left
array([[9, 8],
       [6, 5]])
>>> right
array([[7],
       [4]])
```

10.3.6 Ripetizione/binding (repeat, tile)

Per ripetere:

- i singoli argomenti di un array, ciascuno n volte, si usa `repeat`; nel caso di array multidimensionali si può selezionare l'asse della ripetizione
- un array nel suo complesso usare `tile`: se il secondo argomento è un intero viene effettuata una copia per riga, con una tuple si specifica la struttura finale

```
>>> x = np.array([1,2,3])    # unidimensional repeat
>>> x.repeat(2)              # a) each element 2 time
array([1, 1, 2, 2, 3, 3])
>>> x.repeat([3,2,1])       # b) different times
array([1, 1, 1, 2, 2, 3])

>>> np.tile(x, 2)           # unidimensional tile
```

²Analogamente `numpy.dsplit` effettua il taglio sul terzo asse.

```

array([1, 2, 3, 1, 2, 3])

>>> x = np.arange(4).reshape(2,2) # multidimensional repeat
>>> x.repeat(2, axis = 0)          # a) by row, common number
array([[0, 1],
       [0, 1],
       [2, 3],
       [2, 3]])
>>> x.repeat([2,3], axis = 1)      # b) by col, different number
array([[0, 0, 1, 1, 1],
       [2, 2, 3, 3, 3]])

>>> np.tile(x, 2)                  # multidimensional tile (default by col)
array([[0, 1, 0, 1],
       [2, 3, 2, 3]])
>>> np.tile(x, (2, 3))            # specify the building repetition
array([[0, 1, 0, 1, 0, 1],
       [2, 3, 2, 3, 2, 3],
       [0, 1, 0, 1, 0, 1],
       [2, 3, 2, 3, 2, 3]])
>>> np.tile(x, (2, 1))            # by col
array([[0, 1],
       [2, 3],
       [0, 1],
       [2, 3]])

```

10.3.7 Sorting/ordine (sort, argsort)

Per

- ordinare un array usare il metodo `sort`. Nel caso di array multidimensionale, specificare `axis` per la dimensione di sorting:

```

>>> x = np.array([2,1,3,4])        # unidimensional sorting
>>> x.sort()
>>> x
array([1, 2, 3, 4])

>>> rand = np.random.RandomState(42) # bidimensional
>>> X = rand.randint(0, 10, (4, 6))
>>> np.sort(X, axis=0)              # sorting di ogni colonna
array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])
>>> np.sort(X, axis=1)              # sort di ogni riga
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])

```

Da notare che negli ultimi due casi si perdono eventuali relazioni tra righe e colonne.

- ottenere gli indici dell'ordine si usa `argsort`

```
>>> x = np.array([2, 1, 4, 3, 5])
>>> i = np.argsort(x)
>>> i
array([1, 0, 3, 2, 4])
>>> x[i]
array([1, 2, 3, 4, 5])
```

10.4 Universal functions

10.4.1 Introduzione

Le universal functions (reference qui), o *ufunctions*, sono funzioni chiamate come

```
np.ufunc()
```

ed eseguite su tutti gli elementi di array in maniera vettorizzata.

Vi sono funzioni *unarie* (si applicano ad un array separatamente, le più importanti in tabella 10.3) e *binarie* (si applicando a più array, le più importanti in tab 10.4).

Gli operatori aritmetici e di comparazione (es `>=`) funzionano sotto la scocca, come *ufuncs* binarie di tabella (es `a+b` chiama `np.add(a,b)`).

Altre ufuncs si trovano in `scipy.special`

10.4.2 Metodi delle ufunction (reduce, accumulate, outer)

A quanto pare le ufunction in realtà assomigliano a classi aventi propri metodi (`reduce`, `accumulate`, `reduceat`, `outer`, `at`).

La chiamata di questi metodi è utile soprattutto per funzioni che prendono in input due argomenti e ne ritornano uno singolo in output (eg binarie):

- `reduce` applica una ufuncs agli elementi di un array sino a che un singolo risultato rimane
- `accumulate` fa l'operazione cumulata
- `outer` applica una funzione al prodotto cartesiano degli elementi di due array

Un esempio con `np.add`

```
>>> x = np.arange(1, 6)

>>> np.add.reduce(x)          # come np.sum(x)
np.int64(15)
>>> np.add.accumulate(x)     # come np.cumsum(x)
array([ 1,  3,  6, 10, 15])
>>> np.add.outer(x, x)
```


Funzione	Descrizione
<code>abs, fabs</code>	absolute value
<code>sqrt</code>	square root
<code>exp, expm1</code>	esponenziale, $\exp(x) - 1$
<code>log, log10, log2, log1p</code>	Natural log, log base 10, log base 2, and $\log(1 + x)$, respectively
<code>sin, cos, tan</code>	trigonometriche
<code>arcsin, arccos, arctan</code>	trigonometriche inverse
<code>sign</code>	funzione segno: 1 se positivo, 0 se zero, o -1 se negativo
<code>ceil</code>	smallest integer \geq to each element
<code>floor</code>	largest integer \leq to each element
<code>rint</code>	round to nearest integer (preserving the dtype)
<code>isnan</code>	boolean indicating whether each value is NaN (Not a Number)
<code>isfinite, isinf</code>	each element is finite (non-inf, non-NaN) or infinite, respectively
<code>any</code>	per array booleani, testa se alcuni sono veri
<code>all</code>	per array booleani, testa se tutti sono veri
<code>logical_not, bitwise_not (~)</code>	logical not element-wise
<code>sum</code>	Sum of all the elements in the array or along an axis.
<code>prod</code>	Produttoria
<code>mean</code>	Arithmetic mean. Zero-length arrays have NaN mean.
<code>median</code>	Mediana
<code>percentile</code>	Percentile
<code>std, var</code>	Standard deviation and variance.
<code>min, max</code>	Minimum and maximum.
<code>argmin, argmax</code>	Indices of minimum and maximum elements, respectively.
<code>cumsum</code>	Cumulative sum of elements starting from 0
<code>cumprod</code>	Cumulative product of elements starting from 1

Tabella 10.3: *ufuncs* unarie

Funzione	Descrizione
<code>add (+)</code>	addition
<code>subtract (-)</code>	subtraction
<code>negative (-)</code>	unary negation
<code>multiply (*)</code>	multiplication
<code>divide (/)</code>	division
<code>floor_divide (//)</code>	resto divisione
<code>power (**)</code>	power
<code>mod (%)</code>	modulus remainder
<code>less (<)</code>	less than
<code>greater (>)</code>	
<code>less_equal (<=)</code>	
<code>greater_equal (>=)</code>	
<code>not_equal (!=)</code>	
<code>equal (==)</code>	
<code>maximum, fmax</code>	Element-wise maximum. <code>fmax</code> ignores NaN
<code>minimum, fmin</code>	Element-wise minimum. <code>fmin</code> ignores NaN
<code>logical_and, bitwise_and (&)</code>	and
<code>logical_or, bitwise_or ()</code>	or
<code>logical_xor, bitwise_xor (^)</code>	xor
<code>matmul (@)</code>	Matrix product of two arrays
<code>cov</code>	covariance coefficient/matrix
<code>corrcoef</code>	correlation coefficient/matrix

Tabella 10.4: *ufuncs* binarie

```
array([[ 2,  3,  4,  5,  6],
       [ 3,  4,  5,  6,  7],
       [ 4,  5,  6,  7,  8],
       [ 5,  6,  7,  8,  9],
       [ 6,  7,  8,  9, 10]])
```

10.4.3 Creazione di ufunctions

Per la creazione di funzioni vettorizzate possiamo:

- scrivere funzioni in puro python e vettorizzarle con `np.frompyfunc` (o il wrapper `vectorize`).
`numpy.frompyfunc` prende in input una funzione, il numero di inputs presi e il numero di outputs forniti; la funzione creata/ritornata restituisce sempre un array:

```
>>> def add1_worker(x): # 1 input 1 output
...     return x + 1
...
>>> add1 = np.frompyfunc(add1_worker, 1, 1)
>>> add1(np.arange(10))
array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=object)

>>> def add2_worker(x, y): # 2 input, 1 output
```

```

...     return x + y
...
>>> add2 = np.frompyfunc(add2_worker, 2, 1)
>>> add2(np.arange(10), np.arange(10))
array([0, 2, 4, 6, 8, 10, 12, 14, 16, 18], dtype=object)

>>> def max_min_worker(x, y): # 2 input 2 output
...     return max(x, y), min(x, y) # pure python here
...
>>> maxmin = np.frompyfunc(max_min_worker, 2, 2)
>>> maxmin_wrong = np.frompyfunc(max_min_worker, 2, 1) # wrong in returned

>>> x = np.random.random(3)
>>> y = np.random.random(3)
>>> x
array([0.05890559, 0.43265634, 0.99432112])
>>> y
array([0.20090904, 0.49256551, 0.62591334])
>>> maxmin(x, y)
(array([0.20090904444976976, 0.4925655073905716, 0.9943211175192047],
      dtype=object), array([0.05890559045515198, 0.4326563382467735, 0.6259133358329919],
      dtype=object))
>>> maxmin_wrong(x, y)
array([(0.20090904444976976, 0.05890559045515198),
      (0.4925655073905716, 0.4326563382467735),
      (0.9943211175192047, 0.6259133358329919)], dtype=object)

```

`numpy.vectorize` è una alternativa meno generale/più lenta che permette di specificare il tipo ritornato la funzione.

- utilizzare compilatori LLVM con `numba` a partire da funzioni Python. Mediante Numba si creano funzioni veloci mediante il progetto LLVM (che traduce codice python in codice macchina eseguibile da CPU o GPU). Vedere https://wesmckinney.com/book/advanced-numpy.html#numpy_numba
- utilizzare l'interfaccia C (modo più generale)

10.5 Broadcasting

Per

- array della stessa dimensionalità le operazioni aritmetiche vengono effettuate elemento per elemento

```

>>> x = np.array([0, 1, 2])
>>> y = np.array([5, 5, 5])
>>> x + y
array([5, 6, 7])

```

- array di diverse dimensioni opera il *broadcasting*, ossia un set di regole per applicare ufuncs binarie ad array di dimensioni non uguale. Alcuni esempi:

```

>>> # aggiunta di costante ad array: il valore 5 viene espanso a [5 5 5] prima d
>>> x + 5
array([5, 6, 7])

>>> # aggiunta di vettore a matrice il vettore viene stretchato sulla
>>> # seconda dimensione per matchare la forma della matrice
>>> M = np.ones((3, 3))
>>> M
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
>>> M + x
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])

>>> # somma di vettore 1 x 3 con sua trasposta 3 x 1: entrambi gli array
>>> # sono stretchati fino a raggiungere una dimensione comune (due
>>> # matrici 3 x 3) dopodiché viene effettuata la somma.
>>> x = np.arange(3)
>>> y = x.reshape(3,1).copy()
>>> x
array([0, 1, 2])
>>> y
array([[0],
       [1],
       [2]])
>>> x + y
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])

```

La geometria di esempi del genere è visualizzata in figura 10.1; non vi è una vera e propria espansione in memoria per ragioni di efficienza ma è utile tenere a mente come modello.

Regole di funzionamento Informalmente funziona abbastanza similmente al recycling di R: l'array più piccolo viene replicato per matchare quello più grande³. Formalmente le regole applicate in sequenza sono:

1. se gli array differiscono nel numero di dimensioni (il numero di elementi di **shape**), la forma di quello con un numero inferiori di dimensioni è riempita a sinistra con 1;
2. una volta uniformati **numpy** confronta gli **shape**, elemento (dimensione) per elemento, partendo dall'ultimo. Due dimensioni sono *compatibili* quando sono alternativamente uguali o una di esse è 1: se

- tutte le dimensioni sono *compatibili* verrà eseguita l'operazione

³Il broadcasting è lievemente più sicuro/meno flessibile perché funziona solamente se la struttura matcha perfettamente o se si ha un array di un elemento (è questo che di fatto viene riciclato).

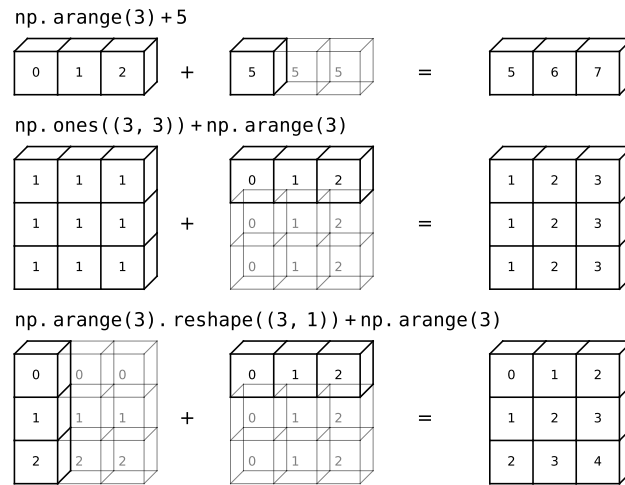


Figura 10.1: Broadcasting geometrics.

- *non tutte sono compatibili* viene sollevata l'eccezione e l'operazione termina

Nel caso di dimensioni compatibili, se due dimensioni non sono uguali, l'array con shape 1 in quella dimensione viene stretchato per matchare l'altra dimensione;

Example 10.5.1 (Somma di un array bidimensionale a monodimensionale). Abbiamo:

```
>>> M = np.ones((2, 3))
>>> a = np.arange(3)
>>> M
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> a
array([0, 1, 2])

>>> ## le shape sono
>>> M.shape
(2, 3)
>>> a.shape
(3,)
```

Applicando le regole del broadcasting:

1. per la regola 1 l'array `a` ha meno dimensioni quindi viene riempito sulla sinistra

```
M.shape -> (2, 3)
a.shape -> (1, 3)
```

2. vediamo quali dimensioni non sono in agreement, queste verranno stretchate per matchare

```
M.shape -> (2, 3)
a.shape -> (2, 3)
```

Ora le shape matchano e la shape del risultato finale sarà (2, 3)

```
>>> M+a
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Example 10.5.2 (Entrambi gli array necessitano di broadcasting). Abbiamo:

```
>>> a = np.arange(3).reshape((3, 1))
>>> b = np.arange(3)
>>> a
array([[0],
       [1],
       [2]])
>>> b
array([0, 1, 2])
>>> a.shape
(3, 1)
>>> b.shape
(3,)
```

Si ha:

1. Per regola 1 b viene riempito sulla sinistra con 1

```
a.shape -> (3, 1)
b.shape -> (1, 3)
```

2. per regola 2 facciamo l'upgrade degli 1 per matchare la dimensione dell'array

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

3. visto che matchano il risultato sarà un array 3×3

Example 10.5.3 (Array incompatibili). Un caso lievemente diverso dal primo dove M è trasposta

```
>>> M = np.ones((3, 2))
>>> a = np.arange(3)
>>> M
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> a
array([0, 1, 2])
```

```
>>> M.shape
(3, 2)
>>> a.shape
(3,)
```

Per regola

1. riempiamo a sinistra a

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

2. la prima dimensione di a è stretchata e si ha

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

3. le dimensioni finali non matchano quindi viene sollevato un errore

```
>>> M + a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

10.6 Altri argomenti

10.6.1 Lavorare con booleani

Vediamo un po' di applicazioni utili

```
>>> ## dati
>>> rng = np.random.RandomState(0)
>>> n = rng.randint(10, size=(3, 4)) # interi uniformi < 10
>>> x = np.array([True, False, True, False])
>>> y = np.array([True, True, False, False])

>>> ## operatori booleani vettorizzati
>>> x & y # np.bitwise_and
array([ True, False, False, False])
>>> x | y # np.bitwise_or
array([ True,  True,  True, False])
>>> ~x # np.bitwise_not
array([False,  True, False,  True])
>>> x ^ y # np.bitwise_xor
array([False,  True,  True, False])

>>> ## any/ all
>>> np.any(n > 8)
np.True_
>>> np.all(n < 10)
np.True_
>>> np.all(n < 8, axis=1)
```

```

array([ True, False,  True])

>>> ## conta di elementi che rispettano un test
>>> np.sum(n < 6) # overall
np.int64(8)
>>> np.sum(n < 6, axis = 1) # per riga
array([4, 2, 2])
>>> np.sum(n > 6, axis = 0) # per colonna
array([1, 1, 1, 0])

>>> ## if then else vettorizzato: np.where(test, iftrue, iffalse)
>>> t = np.arange(6)
>>> f = -t
>>> test = np.array([True, False] * 3)
>>> res = np.where(test, t, f)
>>> res
array([ 0, -1,  2, -3,  4, -5])

>>> # recoding con np.where
>>> c = np.where(t > 3, 3, np.where(t > 1, 2, 1))
>>> c
array([1, 1, 2, 2, 3, 3])

>>> # o con algebra e logica pura
>>> d = 1 + (t > 1) + (t > 3)
>>> d
array([1, 1, 2, 2, 3, 3])

```

10.6.2 Array di stringhe

Gli array possono immagazzinare anche stringhe, ma queste devono essere di dimensione fissata per motivi di efficienza

```

>>> names = ["luca", "bob", "joe"]
>>> x = np.array(names)
>>> x.dtype # stringhe di 4 elementi al massimo
dtype('<U4')
>>> x[0][0:2] # utilizzo di indici
'lu'
>>> # sono comunque normali stringhe eh ...
>>> for i in range(3):
...     x[i] = x[i].capitalize()
...
>>> x
array(['Luca', 'Bob', 'Joe'], dtype='<U4')
>>> # ... ma di lunghezza massima fissata
>>> x[2] = "asdasdasd" # assegnazione, ocio si tronca silentemente
>>> x
array(['Luca', 'Bob', 'asda'], dtype='<U4')

>>> # per allocare più spazio, ad esempio

```



```
>>> y = np.array(names, dtype = '<U16')
>>> y[2] = "asdadasd"
>>> y
array(['luca', 'bob', 'asdadasd'], dtype='<U16')
```

Per creare array di *stringhe di dimensione variabili* non preventivabile a priori si può usare `dtype=object`. Così facendo si crea un array di oggetti generici al quale si può assegnare la qualunque: si perde però l'efficienza di `numpy` (che non lavora più diretto in sequenze contigue di memoria e usare oggetti python aggiunge un sacco di overhead):

```
>>> ## https://stackoverflow.com/questions/14639496
>>> x = np.array(['apples', 'foobar', 'cowboy'], dtype=object)
>>> x
array(['apples', 'foobar', 'cowboy'], dtype=object)
>>> x[2] = "asdadasd"
>>> x
array(['apples', 'foobar', 'asdadasd'], dtype=object)

>>> for i in range(3):
...     x[i] = x[i].capitalize()
...
>>> x
array(['Apples', 'Foobar', 'Asdadasd'], dtype=object)

>>> # A questo array si può assegnare la qualunque ..
>>> x[1] = {1:2, 3:4}
>>> x
array(['Apples', {1: 2, 3: 4}, 'Asdadasd'], dtype=object)
```


Capitolo 11

Pandas

Contents

11.1	Index	156
11.2	Series	156
11.2.1	Creazione e contenuto (<code>array</code> , <code>index</code> , <code>name</code> , <code>dtype</code>)	157
11.2.2	Indexing (<code>[]</code> , <code>.loc</code> , <code>.iloc</code>)	157
11.2.3	Funzionalità per indici (<code>filter</code> , <code>reindex</code> , <code>reset_index</code> , <code>rename</code>)	159
11.2.4	Modifica di valori	160
11.2.5	Rimozione elementi (<code>drop</code> , <code>del</code>)	160
11.2.6	Indici, elaborazione vettorizzata, allineata, reindexing	161
11.2.7	Coercizione di tipo (<code>astype</code>)	162
11.2.8	Valori condizionali (<code>ifelse</code>): <code>np.where</code> , <code>pd.Series.where</code>	163
11.2.9	Applicazione di funzioni (<code>map</code>)	163
11.2.10	Applicazione di più funzioni (<code>.agg</code>)	163
11.2.11	Recode (<code>map</code> e <code>replace</code>)	164
11.2.12	Test di appartenenza (<code>in</code> , <code>isin</code>)	164
11.2.13	Dati mancanti (<code>isna</code> , <code>notna</code> , <code>dropna</code> , <code>fillna</code>)	165
11.2.14	Gestione duplicati (<code>duplicated</code> , <code>unique</code> , <code>drop_duplicates</code>)	165
11.2.15	Sorting (<code>sort_index</code> , <code>sort_values</code>)	166
11.2.16	Discretizzazione/creazione di classi (<code>cut</code> , <code>qcut</code>)	166
11.2.17	Dummy variables (<code>get_dummies</code> , <code>str.get_dummies</code> , <code>idxmax</code>)	168
11.2.18	Stringhe: <code>Series.str</code>	169
11.2.19	Date/ore: <code>Series.dt</code> e funzioni varie	172
11.2.20	Dati categorici: <code>Categorical</code> e <code>Series.cat</code>	174
11.2.21	Indici gerarchici (<code>MultiIndex</code>) nelle serie	176
11.3	DataFrame	178
11.3.1	Creazione e contenuto (<code>info</code> , <code>shape</code> , <code>index</code> , <code>columns</code> , <code>values</code> , <code>name</code>)	178
11.3.2	Indexing (<code>[]</code> , <code>.loc</code> , <code>.iloc</code>)	179
11.3.3	Selezione di righe con <code>query</code>	184
11.3.4	Selezione di colonne sulla base di tipo	184

11.3.5	Accesso a singoli numeri: <code>.at</code> , <code>.iat</code>	185
11.3.6	Aggiunta di colonne (assegnazione, <code>insert</code> , <code>assign</code>)	185
11.3.7	Modifica di valori (indexing e assegnazione: <code>loc</code> , <code>iloc</code> , <code>at</code> , <code>iat</code>)	187
11.3.8	Rimozione righe/colonne (<code>drop</code> , <code>del</code>)	187
11.3.9	Rinominare indici/colonne (<code>rename</code>)	188
11.3.10	Funzionalità per indici, reindexing, <code>MultiIndex</code>	189
11.3.11	Elaborazione allineata	191
11.3.12	Coercizione di tipi (<code>astype</code> , <code>transform</code>)	193
11.3.13	Aggregazione (<code>agg</code>)	193
11.3.14	Applicazione di funzioni	194
11.3.15	Ciclo su righe/colonne (<code>iterrows</code> , <code>items</code>)	196
11.3.16	Merge (<code>merge</code>)	197
11.3.17	Binding (<code>concat</code>)	199
11.3.18	Reshape	199
11.3.19	Test di appartenenza (<code>in</code> , <code>isin</code>)	202
11.3.20	Dati mancanti (<code>count</code> , <code>isna</code> , <code>notna</code> , <code>dropna</code> , <code>fillna</code>)	203
11.3.21	Gestione duplicati (<code>duplicated</code> , <code>drop_duplicates</code>)	204
11.3.22	Sorting di righe/colonne (<code>sort_values</code> , <code>sort_index</code>)	205
11.4	Data I/O	206
11.5	Cookbook	208
11.5.1	Stampare localmente tutto il contenuto di un Data- Frame	208

`pandas` è il pacchetto di Python che fornisce strutture di dati e funzioni di utilità per l'analisi statistica standard. Importazione standard:

```
>>> import pandas as pd
>>> import numpy as np # spesso utile/necessario
```

Le strutture dati fornite sono `Series` (array unidimensionale) e `DataFrame` (array bidimensionale), alle quali si aggiungono altre strutture per gli indici (`Index` `MultiIndex`).

11.1 Index

L'`Index` è l'oggetto che fornisce i metadati necessari per `Series`, nonché per righe e colonne del `DataFrame`. Alcune peculiarità:

- sono oggetti *immutabili* e non possono essere modificati una volta creati;
- possono contenere doppi;
- possono essere condivisi tra strutture di dati;
- si usa `in` per testare la presenza di un dato indice in un oggetto `Index`.

11.2 Series

`Series` è un array unidimensionale, di contenuto omogeneo, dimensione fissa (?), dotato (eventualmente) di etichette/labels (index) (può essere pensato come un `dict`).

11.2.1 Creazione e contenuto (array, index, name, dtype)

Il modo base per creare una Series è

```
x = pd.Series(data)           # senza indici (data: lista, np.array o scalare)
x.index = ["a", "b", "c"]     # aggiungere indici in un secondo momento
y = pd.Series(dict)           # con indici, quelli del dict
z = pd.Series(data, index = idx) # con indici, forniti
```

Gli elementi principali di una serie sono **array** (un oggetto che wrappa un array numpy) ed **index** (oggetto RangeIndex):

```
>>> x = pd.Series({"d":4, "b":1, "a":2, "c":3})
>>> x
d    4
b    1
a    2
c    3
dtype: int64
>>> x.array
<NumpyExtensionArray>
[np.int64(4), np.int64(1), np.int64(2), np.int64(3)]
Length: 4, dtype: int64
>>> x.index
Index(['d', 'b', 'a', 'c'], dtype='object')
```

Infine sia serie che **index** hanno l'attributo **name**, sfruttato da pandas qua e la:

```
>>> y = pd.Series({'luca': 23, 'andrea': 12, 'davide': 15})
>>> y.name = 'età_anni'
>>> y.index.name = 'giocatore'
>>> y
giocatore
luca      23
andrea    12
davide    15
Name: età_anni, dtype: int64
```

Pandas ha un set di tipi propri (tab 11.1) che possono essere specificati in sede di chiamata (parametro **dtype**). Nella definizione, se non viene specificato **dtype** nella chiamata pandas cerca di inferire

```
>>> x = pd.Series([0, 1, 0, 1])
>>> x.dtype
dtype('int64')
>>> x = pd.Series([0, 1, 0, 1], dtype = 'boolean')
>>> x.dtype
BooleanDtype
```

11.2.2 Indexing ([], .loc, .iloc)

```
>>> x = pd.Series([4, 1, 2, 3])           # senza
>>> y = pd.Series({"d":4, "b":1, "a":2, "c":3}) # con indici
```

Stringa	Classe	Descrizione
boolean	BooleanDtype	Nullable Boolean data
category	CategoricalDtype	Categorical data type
?	DatetimeTZDtype	Datetime with time zone
Float32	Float32Dtype	32-bit nullable floating point
Float64	Float64Dtype	64-bit nullable floating point
Int8	Int8Dtype	8-bit nullable signed integer
Int16	Int16Dtype	16-bit nullable signed integer
Int32	Int32Dtype	32-bit nullable signed integer
Int64	Int64Dtype	64-bit nullable signed integer
UInt8	UInt8Dtype	8-bit nullable unsigned integer
UInt16	UInt16Dtype	16-bit nullable unsigned integer
UInt32	UInt32Dtype	32-bit nullable unsigned integer
UInt64	UInt64Dtype	64-bit nullable unsigned integer

Tabella 11.1: Tipi estesi di pandas

```

>>> # serie senza indici: indexing numerico
>>> x[0]          # ammesso, no warning here
np.int64(4)
>>> x[[3, 0, 1]] # ammesso, no warning here
3    3
0    4
1    1
dtype: int64

>>> # serie con indici: indexing numerico
>>> # y[0]          # warning/deprecato: usare x.iloc[0]
>>> # y[[3, 0, 1]]  # warning/deprecato: usare x.iloc[[3,0,1]]
>>> y[:3]          # indici con slicing
d    4
b    1
a    2
dtype: int64
>>> y[y > y.median()] # indici logici
d    4
c    3
dtype: int64

>>> # serie con indici: indexing con stringhe (indici della serie)
>>> y["a"]          # restituito un valore
np.int64(2)
>>> y[["b", "d"]]   # restituita una Serie
b    1
d    4
dtype: int64
>>> y['d':'a']      # slicing con label/index: estremi inclusi
d    4
b    1

```

```
a    2
dtype: int64
```

Se si desidera una selezione sicura utilizzare gli attributi `.loc` e `.iloc`;

- `loc` prende in input labels/stringhe oppure booleani (o una funzione che produca questi); se l'array non ha indici non funziona. Se si vuole essere sicuri che una array sia ritornato fornire una lista a `loc`

```
>>> y = pd.Series({"d":4, "b":1, "a":2, "c":3})
```

```
>>> # singolo valore vs lista
```

```
>>> y.loc["a"]
```

```
np.int64(2)
```

```
>>> y.loc[["a"]]
```

```
a    2
```

```
dtype: int64
```

```
>>> # funzione selettiva
```

```
>>> y.loc[lambda x: x > 2]
```

```
d    4
```

```
c    3
```

```
dtype: int64
```

```
>>> # stessa cosa con array di booleani
```

```
>>> y.loc[y > 2]
```

```
d    4
```

```
c    3
```

```
dtype: int64
```

- `iloc` prende in input interi o booleani (o una funzione).

```
>>> # iloc e interi
```

```
>>> x = pd.Series([4, 1, 2, 3]) # senza
```

```
>>> x.iloc[[1, 3]]
```

```
1    1
```

```
3    3
```

```
dtype: int64
```

```
>>> x.iloc[[True, False, True, False]]
```

```
0    4
```

```
2    2
```

```
dtype: int64
```

11.2.3 Funzionalità per indici (filter, reindex, reset_index, rename)

Il metodo `filter` restituisce i subset i quali indici soddisfano criteri

```
>>> y = pd.Series({"d":4, "b":1, "a":2, "c":3})
```

```
>>> y.filter(items = ["a", "b"])
```

```
a    2
```

```
b    1
```

```
dtype: int64
```

```
>>> # possibili anche regex
```

Il metodo `reindex` estrae in base agli indici, non accetta doppi nelle chiavi ma riempie buchi senza sollevare eccezioni (es indici richiesti non esistenti)

```
>>> y = pd.Series({"d":4, "b":1, "a":2, "c":3})
>>> y.reindex(["d", "x"])
d    4.0
x     NaN
dtype: float64
```

Il metodo `reset_index` sostituisce l'indice con un progressivo numerico e restituisce un dataframe col vecchio indice come variabile e la vecchia variabile

```
>>> y = pd.Series({"d":4, "b":1, "a":2, "c":3}) # con indici
>>> y.reset_index()
   index  0
0      d  4
1      b  1
2      a  2
3      c  3
```

Il metodo `rename` effettua un recode dell'indice

```
>>> y.rename({"a": "x", "b": "y"})
d    4
y    1
x    2
c    3
dtype: int64
```

11.2.4 Modifica di valori

Funzionante mediante indici (e broadcasting)

```
>>> x = pd.Series([4, 1, 2, 3]) # senza
>>> y = pd.Series({"d":4, "b":1, "a":2, "c":3}) # con indici

>>> x[0] = 2 # senza indici
>>> x[:] = 0 # annullare un vettore; x = 0 è sbagliato

>>> y["a"] = 5 # con indici
>>> y[:] = 1 # settare tutti gli elementi a 1
```

11.2.5 Rimozione elementi (drop, del)

Metodo `drop` per una modifica temporanea (ritorna l'oggetto modificato) o `del` per una definitiva


```

>>> x = pd.Series([1, 2, 3], index = ['a', 'b', 'c'])

>>> x.drop('b') # Rimozione temporanea
a    1
c    3
dtype: int64
>>> x          # la modifica non è salvata
a    1
b    2
c    3
dtype: int64
>>> x.drop(['b', 'c']) # più elementi
a    1
dtype: int64

>>> del x['b'] # rimozione definitiva
>>> x
a    1
c    3
dtype: int64

```

11.2.6 Indici, elaborazione vettorizzata, allineata, reindexing

Similmente a numpy le elaborazioni sono vettorizzate (si applicano *ufuncs* e broadcasting); viene preservato l'indice:

```

>>> x = pd.Series([0,1,2])
>>> np.exp(x) + 1
0    2.000000
1    3.718282
2    8.389056
dtype: float64

```

Le elaborazioni aventi per oggetto due serie diverse avvengono sulla base degli indici, effettuando il cosiddetto allineamento automaticamente

```

>>> x = pd.Series({'a': 1, 'c': 2, 'b': 3})
>>> y = pd.Series({'c': 1, 'b': 0, 'd': 0})
>>> x * y
a    NaN
b    0.0
c    2.0
d    NaN
dtype: float64

```

a non può essere calcolato perché manca in y, d perché manca in x.

Il **reindexing** crea un nuovo oggetto, caratterizzato da un set di indici diverso: nelle serie serve per riordinare

```

>>> x = pd.Series([4.5, 7.2, -5.3, 3.6], index = ['d', 'b', 'a', 'c'])
>>> x

```

```

d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
>>> y = x.reindex(['a', 'b', 'c', 'd', 'e'])
>>> y
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64

>>> y['a'] = 999 # viene effettivamente creata una copia
>>> x
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64

```

11.2.7 Coercizione di tipo (astype)

Si usa il metodo `astype` fornendo alternativamente un tipo builtin di Python, una stringa, un `numpy.dtype` o i tipi di pandas

```

>>> a = pd.Series(["1", "2", "3"])

>>> a.astype(int) # tipo builtin
0    1
1    2
2    3
dtype: int64
>>> a.astype("float") # stringa
0    1.0
1    2.0
2    3.0
dtype: float64
>>> a.astype(np.int8) # numpy
0    1
1    2
2    3
dtype: int8
>>> a.astype("category") # pandas type
0    1
1    2
2    3
dtype: category
Categories (3, object): ['1', '2', '3']

```

11.2.8 Valori condizionali (ifelse): np.where, pd.Series.where

Si ha che

- np.where è l'equivalente più diretto di ifelse di R

```
>>> test = np.array([True, False, True, False])
>>> a = np.array(["a", "a", "a", "a"])
>>> b = np.array(["b", "b", "b", "b"])
>>> np.where(test, a, b)
array(['a', 'b', 'a', 'b'], dtype='<U1')
```

- pd.Series.where modifica una serie di base laddove una condizione non è verificata

```
>>> test = pd.Series([True, False, True, False])
>>> a = pd.Series(["a", "a", "a", "a"])
>>> a.where(test, other = "qweqwe")
0      a
1  qweqwe
2      a
3  qweqwe
dtype: object
```

11.2.9 Applicazione di funzioni (map)

Per applicare una funzione per scalare a tutti gli elementi di una Series utilizzare map

```
>>> # esempio di formattazione
>>> rng = np.random.default_rng(6235)
>>> a = pd.Series(rng.random(3))
>>> def formatter(x):
...     return f"{x:.2f}"
...
>>> a.map(formatter)
0    0.55
1    0.67
2    0.73
dtype: object
```

11.2.10 Applicazione di più funzioni (.agg)

agg permette di applicare più funzioni alla stessa serie, per info sui metodi vedere Harrison pag 67. Di base quando gli si passa una stringa (es "mean" pandas mappa al metodo corrispondente per le serie)

```
>>> s = pd.Series([1, 2, 3, 4])

>>> def custom(x):
...     return x[0]
...
>>> s.agg(['min', 'max', np.var, custom])
```

```

min      1.000000
max      4.000000
var      1.666667
custom   1.000000
dtype: float64

```

11.2.11 Recode (map e replace)

Se a `map` di passa un dict effettua dei *recode* (il mapping deve essere completo):

```

>>> a = pd.Series(["1", "2", "3", "4"])
>>> rec = {"1": "1-2", "2": "1-2", "3": "3-4", "4": "3-4"}
>>> a.map(rec)
0    1-2
1    1-2
2    3-4
3    3-4
dtype: object

```

Se si vuole sostituire solo alcuni elementi e lasciare invariati gli altri si usa `replace`:

```

>>> a = pd.Series(["1", "2", "3", "4"])
>>> a.replace({"3": "3+", "4": "3+"})
0     1
1     2
2    3+
3    3+
dtype: object

```

11.2.12 Test di appartenenza (in, isin)

Per testare l'appartenenza si può pensare alla serie come a un `dict` e usare `in` sugli indici o il metodo `isin` per i valori:

```

>>> x = pd.Series([1, 3], index = ['a', 'c'])

>>> 'a' in x          # uso di indici
True
>>> 'b' in x
False

>>> x.isin([1, 2]) # uso di valori
a     True
c     False
dtype: bool

```

11.2.13 Dati mancanti (isna, notna, dropna, fillna)

Usiamo `np.nan` per indicare dati mancanti¹. È considerato NA anche il `None` builtin di Python. Vediamo metodi/funzioni più utili a livello di `Series`: generalmente se non si usa `inplace` i metodi restituiscono una copia.

```
>>> z = pd.Series([1, 2, np.nan, 3, None])

>>> z.isna()      # test, equivalentemente pd.isna(z)
0    False
1    False
2     True
3    False
4     True
dtype: bool
>>> z.notna()     # negazione del test precedente, equivale a pd.notna(z)
0     True
1     True
2    False
3     True
4    False
dtype: bool
>>> z.dropna()    # utility: filtrare
0    1.0
1    2.0
3    3.0
dtype: float64
>>> z.fillna(-9) # utility: riempire
0    1.0
1    2.0
2   -9.0
3    3.0
4   -9.0
dtype: float64
```

11.2.14 Gestione duplicati (duplicated, unique, drop_duplicates)

I metodi più utili:

```
>>> x = pd.Series([1, 2, 2, 3, 3, 3])

>>> x.duplicated()      # marca i doppi
0    False
1    False
2     True
3    False
4     True
5     True
dtype: bool
>>> x.unique()          # rende unica: restituisce ndarray
```

¹Con Pandas 2.0 `pd.NA` è considerato sperimentale. A volte può essere comodo il trick `from numpy import NaN as NA` specialmente quando si devono generare tanti dati a mano.

```
array([1, 2, 3])
>>> x.drop_duplicates() # rende unica: restituisce Series
0    1
1    2
3    3
dtype: int64
```

11.2.15 Sorting (sort_index, sort_values)

Se si vuole ordinare sulla base degli indici si usa il metodo `sort_index`, sulla base dei valori si usa il metodo `sort_values`

```
>>> x = pd.Series([1, 3, 2, 4], index=["d", "a", "b", "c"])
>>> x.sort_index()
a    3
b    2
c    4
d    1
dtype: int64
>>> x.sort_values()
d    1
b    2
a    3
c    4
dtype: int64
```

11.2.16 Discretizzazione/creazione di classi (cut, qcut)

Si usa la funzione `cut` sulla `Series` (che ha la peculiarità di restituire un oggetto `Categorical` che presenta notevoli somiglianze coi `factor`). Si possono fornire i `breaks` o il numero per `breaks` equispaziati. `qcut` invece effettua un `cut` sulla base dei quantili.

```
>>> rng = np.random.default_rng(12093)
>>> age = pd.Series(rng.integers(1, 100, size = 10))
>>> age
0    32
1     2
2    79
3    26
4    93
5    41
6    49
7    83
8    71
9    23
dtype: int64

>>> # cuts specificati
>>> cuts = [0, 25, 50, 75, 100, 125]
>>> labs = ["giovane", "medio", "anziano", "vecchio", "vetusto"]
```

```
>>> agecl1 = pd.cut(age, bins = cuts, labels = labs)
>>> agecl1
0      medio
1    giovane
2    vecchio
3      medio
4    vecchio
5      medio
6      medio
7    vecchio
8    anziano
9    giovane
dtype: category
Categories (5, object): ['giovane' < 'medio' < 'anziano' < 'vecchio' < 'vetusto']
```

```
>>> # 3 cut equispaziati tra minimo e massimo
```

```
>>> agecl2 = pd.cut(age, bins = 3)
```

```
>>> agecl2
```

```
0    (1.909, 32.333]
1    (1.909, 32.333]
2    (62.667, 93.0]
3    (1.909, 32.333]
4    (62.667, 93.0]
5    (32.333, 62.667]
6    (32.333, 62.667]
7    (62.667, 93.0]
8    (62.667, 93.0]
9    (1.909, 32.333]
```

```
dtype: category
```

```
Categories (3, interval[float64, right]): [(1.909, 32.333] < (32.333, 62.667] < (62.667, 93.0]]
```

```
>>> # quantiles based
```

```
>>> agecl3 = pd.qcut(age, q = 5) # quintile of age
```

```
>>> agecl3
```

```
0    (25.4, 37.4]
1    (1.999, 25.4]
2    (57.8, 79.8]
3    (25.4, 37.4]
4    (79.8, 93.0]
5    (37.4, 57.8]
6    (37.4, 57.8]
7    (79.8, 93.0]
8    (57.8, 79.8]
9    (1.999, 25.4]
```

```
dtype: category
```

```
Categories (5, interval[float64, right]): [(1.999, 25.4] < (25.4, 37.4] < (37.4, 57.8] < (57.8, 79.8] < (79.8, 93.0]]
```

11.2.17 Dummy variables (`get_dummies`, `str.get_dummies`, `idxmax`)

Per:

- creare un `DataFrame` di dummy a partire da una serie si usa `get_dummies`

```
>>> x = pd.Series(["a", "a", "b", "b", "c"])
>>> pd.get_dummies(x)
   a      b      c
0  True False False
1  True False False
2 False  True False
3 False  True False
4 False False  True
>>> pd.get_dummies(x, dtype = int)
   a  b  c
0  1  0  0
1  1  0  0
2  0  1  0
3  0  1  0
4  0  0  1
```

Interessante anche `str.getdummies` utile per specificare i separatori (es per risposte multiple)

```
>>> x = pd.Series(["a|b", "b|c", "a", "a|b|c"])
>>> x.str.get_dummies(sep = "|")
   a  b  c
0  1  1  0
1  0  1  1
2  1  0  0
3  1  1  1
```

In entrambi i casi si può aggiungere un prefisso ai nomi di colonna creati mediante specificando `prefix`.

- per ritrasformare un dataframe di dummies in una unica variable usare `idxmax`:

```
>>> x = pd.Series(["a", "a", "b", "b", "c"])
>>> dum = pd.get_dummies(x)

>>> dum.idxmax(axis = "columns")
0    a
1    a
2    b
3    b
4    c
dtype: object
```


11.2.18 Stringhe: Series.str

Vediamo alcune funzionalità utili delle Series di stringhe.

- Vi sono metodi direttamente accedibili dalla Series
- altri metodi sono accedibili da Series.str

Ad ogni modo vi è overlap con i metodi delle str Python di base.

```
>>> suits = pd.Series(["clubs", "Diamonds", "hearts", "Spades"])
>>> rps = pd.Series(["rock ", " paper", "scissors"])

>>> # lunghezza e sottostringhe
>>> # -----
>>> suits.str.len()          # lunghezza
0    5
1    8
2    6
3    6
dtype: int64
>>> suits.str[2:5]          # subsetting stringa (es)
0    ubs
1    amo
2    art
3    ade
dtype: object
>>> rps.str.strip()         # elimina bianchi
0    rock
1    paper
2    scissors
dtype: object
>>> suits.str.pad(8, fillchar="_") # aggiungi caratteri per uniformare lunghezza
0    __clubs
1    Diamonds
2    __hearts
3    __Spades
dtype: object

>>> # case (vi è anche title e capitalize, meno interessanti)
>>> # -----
>>> suits.str.lower()
0    clubs
1    diamonds
2    hearts
3    spades
dtype: object
>>> suits.str.upper()
0    CLUBS
1    DIAMONDS
2    HEARTS
3    SPADES
dtype: object
```

```

>>> # splitting
>>> # -----
>>> agecl = pd.Series(["0-10", "11-15", "11-15", "61-65", "46-50"])
>>> agecl.str.split("-") # splitta in liste di caratteri
0      [0, 10]
1      [11, 15]
2      [11, 15]
3      [61, 65]
4      [46, 50]
dtype: object
>>> agecl.str.split("-", expand=True) # come sopra ma ritorna un df
   0  1
0  0  10
1  11  15
2  11  15
3  61  65
4  46  50

>>> # paste two
>>> # -----
>>> suits + "5" # aggiungi in coda
0      clubs5
1  Diamonds5
2    hearts5
3    Spades5
dtype: object
>>> suits + suits # aggiungi in coda
0      clubsclubs
1  DiamondsDiamonds
2    heartshearts
3    SpadesSpades
dtype: object
>>> suits.str.cat(suits) # aggiungi in coda
0      clubsclubs
1  DiamondsDiamonds
2    heartshearts
3    SpadesSpades
dtype: object

>>> # collassa a stringa unica
>>> # -----
>>> suits.str.cat(sep=", ")
'clubs, Diamonds, hearts, Spades'

>>> # trova match
>>> # -----
>>> suits.str.contains("[ae]") # ne ha?
0    False
1     True
2     True

```

```

3      True
dtype: bool
>>> suits.str.count("[ae]")      # conta quanti
0      0
1      1
2      2
3      2
dtype: int64
>>> suits.str.find("e")          # trova posizione match
0     -1
1     -1
2      1
3      4
dtype: int64

>>> # replace: utilizzare str.replace per rimpiazzare singola lettera, .replace per
>>> # parole intere
>>> # -----
>>> suits.str.replace("a", "4")
0      clubs
1    Di4monds
2      he4rts
3      Sp4des
dtype: object
>>> suits.replace("Diamonds", "foobar")
0      clubs
1    foobar
2    hearts
3    Spades
dtype: object
>>> suits.replace({"Diamonds": "foobar", "Spades": "asdasd"})
0      clubs
1    foobar
2    hearts
3    asdasd
dtype: object

>>> # estrai i match di una espressione regolare
>>> # -----
>>> suits.str.extractall("[ae](.)") # restituisce df
      0  1
      match
1 0      a  m
2 0      e  a
3 0      a  d
   1      e  s
>>> suits.str.findall("[ae](.)")    # restituisce series
0      []
1      [(a, m)]
2      [(e, a)]
3      [(a, d), (e, s)]

```

```

dtype: object
>>> suits.str.findall("[ae]")          # altro esempio (senza parentesi)
0      []
1      [ia]
2      [he]
3      [pa, de]
dtype: object

```

11.2.19 Date/ore: Series.dt e funzioni varie

Vediamo alcune funzionalità utili delle `Series` di date/ore. Qui invece vi è legame col modulo `datetime`.

```

>>> # alcune stringhe in vari formati di data/or
>>> iso = pd.Series(["1969-07-20 20:12:40",
...                  "1969-11-19 06:54:35",
...                  "1971-02-05 09:18:11"])

>>> eu = pd.Series(["20/07/1969 20:12:40",
...                  "19/11/1969 06:54:35",
...                  "05/02/1971 09:18:11"])

>>> us = pd.Series(["07/20/1969 20:12:40",
...                  "11/19/1969 06:54:35",
...                  "02/05/1971 09:18:11"])

>>> # parsing
>>> pd.to_datetime(iso) # iso works out of the box (anche se solo data es 2020-10-01)
0    1969-07-20 20:12:40
1    1969-11-19 06:54:35
2    1971-02-05 09:18:11
dtype: datetime64[ns]
>>> pd.to_datetime(eu, dayfirst = True) # opzione per eu
0    1969-07-20 20:12:40
1    1969-11-19 06:54:35
2    1971-02-05 09:18:11
dtype: datetime64[ns]
>>> pd.to_datetime(us, dayfirst = False) # opzione per us
0    1969-07-20 20:12:40
1    1969-11-19 06:54:35
2    1971-02-05 09:18:11
dtype: datetime64[ns]
>>> pd.to_datetime(eu, format="%d/%m/%Y %H:%M:%S") # formato custom (not needed here)
0    1969-07-20 20:12:40
1    1969-11-19 06:54:35
2    1971-02-05 09:18:11
dtype: datetime64[ns]

>>> # creazione data da componenti
>>> componenti = pd.DataFrame({
...     "year" : [1969, 1969, 1971],

```

```

...     "month" : [7, 11, 2],
...     "day" : [20, 19, 5]
... })
>>> pd.to_datetime(componenti)
0    1969-07-20
1    1969-11-19
2    1971-02-05
dtype: datetime64[ns]

>>> # Estrazione componenti
>>> isod = pd.to_datetime(iso)

>>> isod.dt.year
0    1969
1    1969
2    1971
dtype: int32
>>> isod.dt.month
0     7
1    11
2     2
dtype: int32
>>> isod.dt.month_name("it_IT.UTF-8") # see shell `locale`
0    Luglio
1    Novembre
2    Febbraio
dtype: object
>>> isod.dt.day
0    20
1    19
2     5
dtype: int32
>>> isod.dt.day_name()
0    Sunday
1    Wednesday
2    Friday
dtype: object

>>> # da datetime a data (ocio che è una stringa)
>>> isod.dt.date
0    1969-07-20
1    1969-11-19
2    1971-02-05
dtype: object

>>> # arrotondare datetime alla data
>>> isod.dt.round("D")
0    1969-07-21
1    1969-11-19
2    1971-02-05
dtype: datetime64[ns]

```

```

>>> isod.dt.floor("D")
0    1969-07-20
1    1969-11-19
2    1971-02-05
dtype: datetime64[ns]
>>> isod.dt.ceil("D")
0    1969-07-21
1    1969-11-20
2    1971-02-06
dtype: datetime64[ns]
>>> # specificando il formato sono disponibili altri arrotondamenti, es H,
>>> # M, S per ore minuti secondi, poi possibile arrotondare a 2H 3M etc

>>> # differenza date
>>> from datetime import datetime
>>> date1 = isod
>>> date2 = pd.to_datetime([datetime.now()] * 3)
>>> datediff = date2 - date1
>>> datediff
0    20383 days 14:34:00.390579
1    20262 days 03:52:05.390579
2    19819 days 01:28:29.390579
dtype: timedelta64[ns]
>>> datediff.dt.days / 365.25
0    55.805613
1    55.474333
2    54.261465
dtype: float64

>>> # aggiungere a una data
>>> td = pd.to_timedelta(pd.Series([1,2,3]), "d")
>>> date2 + td
0    2025-05-12 10:46:40.390579
1    2025-05-13 10:46:40.390579
2    2025-05-14 10:46:40.390579
dtype: datetime64[ns]

>>> # esportare a stringa
>>> isod.dt.strftime("%d/%m/%Y")
0    20/07/1969
1    19/11/1969
2    05/02/1971
dtype: object

```

11.2.20 Dati categorici: Categorical e Series.cat

È una rappresentazione a-la factor con interi linkati a label per risparmiare spazio rispetto alle stringhe secche. I dati categorici si possono creare mediante `pd.Categorical` oppure attraverso series specificando `category` come dtype.

```

>>> # metodi di creazione standard

```

```
>>> c = pd.Series(list("abbccc"), dtype = 'category')
>>> c = pd.Categorical(["asd", "foo", "asd", "bar"])

>>> # wrapper comodo che imita factor
>>> import pylibmisc as lb
>>> c = lb.dm.to_categorical([1, 2, 3] * 2,
...                          levels = [1,2,3],
...                          labels = ["a","b","c"])
```

Ad ora, a quanto pare, se i categorici:

- sono una serie a parte si può accedere ai metodi di sotto direttamente
- fanno parte di un dataframe:
 - i metodi di sotto sono accessibili mediante `.cat.metodo`
 - hanno a disposizione anche le funzionalità per stringa sotto `.str`

Alcuni attributi e metodi caratteristici:

```
>>> c.codes          # tipo as.integer di un factor
array([0, 1, 2, 0, 1, 2], dtype=int8)
>>> c.categories     # lista i livelli del factor
Index(['a', 'b', 'c'], dtype='object')
>>> c.ordered        # booleana test ordinamento
False

>>> c.as_ordered()   # trasforma in ordered
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['a' < 'b' < 'c']
>>> c.as_unordered() # trasforma in unordered
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']

>>> c.set_categories(["a", "b"])          # metodo imposta le categorie ad una nuova lista
['a', 'b', NaN, 'a', 'b', NaN]
Categories (2, object): ['a', 'b']
>>> c.rename_categories(["x", "y", "z"])  # rinomina le categorie
['x', 'y', 'z', 'x', 'y', 'z']
Categories (3, object): ['x', 'y', 'z']
>>> c.reorder_categories(["c", "b", "a"]) # cambia ordinamento
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['c', 'b', 'a']

>>> c.add_categories("asd")               # aggiunge categorie alla fine della lista
['a', 'b', 'c', 'a', 'b', 'c']
Categories (4, object): ['a', 'b', 'c', 'asd']
>>> c.remove_categories("b")              # toglie categorie creando mancanti
['a', NaN, 'c', 'a', NaN, 'c']
Categories (2, object): ['a', 'c']
>>> c.remove_unused_categories()          # toglie categorie con zero frequenze (qui non applicabile)
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```

11.2.21 Indici gerarchici (MultiIndex) nelle serie

L'indexing gerarchico (implementato mediante la classe `MultiIndex`) permette di gestire dati multidimensionali, riconducendoli ad una tabella a due dimensioni.

Serve nel *reshape* dei dati (funzioni `stack/unstack`) e nelle operazioni basate su gruppo (formare pivot table, eg statistiche stratificate).

Qui vediamo il multiindexing nelle `Series`, lasciamo quello dei `DataFrame` per più tardi

11.2.21.1 Definizione

```
>>> df = pd.Series(rng.random(10),
...                 index = [list("aaabbbccdd"), [1, 2, 3] * 3 + [3]])
>>> df
a  1    0.779332
   2    0.180644
   3    0.333745
b  1    0.997920
   2    0.741332
   3    0.166895
c  1    0.350324
   2    0.114726
d  3    0.327335
   3    0.368321
dtype: float64
>>> df.index
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 2),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 3),
            ('d', 3)],
           )
```

11.2.21.2 Indexing

L'indexing sulla base di un singolo indice è possibile, ad esempio

```
>>> df['b']
1    0.997920
2    0.741332
3    0.166895
dtype: float64
>>> df['b':'c']
b  1    0.997920
   2    0.741332
   3    0.166895
```



```

c 1    0.350324
  2    0.114726
dtype: float64
>>> df.loc[["b", "d"]]
b 1    0.997920
  2    0.741332
  3    0.166895
d 3    0.327335
  3    0.368321
dtype: float64

```

Ed è possibile la selezione anche da un livello di index più interno:

```

>>> df.loc[:, 2 ]
a    0.180644
b    0.741332
c    0.114726
dtype: float64

```

11.2.21.3 Reshape

Un esempio di reshape di una `Series` con `MultiIndex`, che diviene un `DataFrame`

```

>>> # Serie
>>> df = pd.Series(np.random.uniform(size=9),
...                 index=[["a", "a", "a", "b", "b", "c", "c", "d", "d"],
...                 [1, 2, 3, 1, 3, 1, 2, 2, 3]])
>>> df.index.names = ["id", "time"]
>>> df
id time
a 1    0.829821
  2    0.326755
  3    0.244986
b 1    0.030293
  3    0.052049
c 1    0.070079
  2    0.229958
d 2    0.399490
  3    0.480200
dtype: float64

>>> # DataFrame wide
>>> df2 = df.unstack()
>>> df2
time      1      2      3
id
a    0.829821  0.326755  0.244986
b    0.030293      NaN  0.052049
c    0.070079  0.229958      NaN
d      NaN  0.399490  0.480200
>>> type(df2)
<class 'pandas.core.frame.DataFrame'>

```

```

>>> # Tornare in versione long (serie)
>>> df3 = df2.stack()
>>> df3
id  time
a   1    0.829821
    2    0.326755
    3    0.244986
b   1    0.030293
    3    0.052049
c   1    0.070079
    2    0.229958
d   2    0.399490
    3    0.480200
dtype: float64
>>> type(df3)
<class 'pandas.core.series.Series'>

```

11.3 DataFrame

Struttura a due dimensioni (con indici di riga e colonna) con colonne che possono assumere tipi differenti. Può essere pensato come un dict di Series caratterizzate dagli stessi indici (di riga).

11.3.1 Creazione e contenuto (info, shape, index, columns, values, name)

Si crea mediante:

```
df = pd.DataFrame(data, index, columns) # index, columns opzionali
```

dove `data` può essere un dict (contenente list, dicts, Series o array numpy) un array numpy 2d, un altro DataFrame o Series. `index` e `column`, opzionali, servono per specificare indici di riga e nomi colonna.

```

>>> # modo più comune di creare un DataFrame per colonne: dict di liste
>>> data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...         "year": [2000, 2001, 2002, 2001, 2002, 2003],
...         "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> states = pd.DataFrame(data, index = list("abcdef"))

>>> # creazione di un DataFrame per riga (lista di dict)
>>> data2 = pd.DataFrame([
...     {"state": "Ohio", "year": 2000, "pop": 1.5},
...     {"state": "Ohio", "year": 2001, "pop": 1.7} # ...
... ])

>>> # Nella creazione vengono rispettati indici comuni
>>> a = pd.Series(range(3), index=['a', 'b', 'c'])
>>> b = pd.Series(range(4), index=['a', 'b', 'c', 'd'])

```

```
>>> df = pd.DataFrame({'one' : a, 'two' : b})
>>> df
   one  two
a  0.0    0
b  1.0    1
c  2.0    2
d  NaN    3
```

Per avere una idea sintetica si usa il metodo `info`; gli attributi principali sono `shape`, `index`, `columns`, `values` e i loro `name`.

```
>>> df.info() # idea sintetica
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, a to d
Data columns (total 2 columns):
#   Column  Non-Null Count  Dtype
---  -
0    one      3 non-null      float64
1    two      4 non-null      int64
dtypes: float64(1), int64(1)
memory usage: 96.0+ bytes

>>> df.shape # come numpy
(4, 2)
>>> df.index # indici di riga
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> df.columns # nomi di colonna
Index(['one', 'two'], dtype='object')
>>> df.values # valori grezzi (array numpy)
array([[ 0.,  0.],
       [ 1.,  1.],
       [ 2.,  2.],
       [nan,  3.]])

>>> # attributi name di index e columns
>>> df.index.name = "soggetto"
>>> df.columns.name = "rilevazione"
>>> df
rilevazione  one  two
soggetto
a           0.0    0
b           1.0    1
c           2.0    2
d           NaN    3
```

11.3.2 Indexing ([], .loc, .iloc)

Il subsetting avviene mediante le quadre `[]` (più limitata) e gli attributi `.loc` e `.iloc`.

Selezione di righe o colonne mediante `[]`

```

>>> data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...         "year": [2000, 2001, 2002, 2001, 2002, 2003],
...         "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> df = pd.DataFrame(data)      # senza indice di riga

>>> # SELEZIONE DI COLONNE: stringhe e liste interpretate come nome di col
>>> df.state      # singola colonna, meglio per l'uso interattivo
0      Ohio
1      Ohio
2      Ohio
3  Nevada
4  Nevada
5  Nevada
Name: state, dtype: object
>>> df["state"] # singola colonna, meglio per la programmazione
0      Ohio
1      Ohio
2      Ohio
3  Nevada
4  Nevada
5  Nevada
Name: state, dtype: object
>>> df[["state", "pop"]] # due colonne
   state  pop
0   Ohio  1.5
1   Ohio  1.7
2   Ohio  3.6
3  Nevada  2.4
4  Nevada  2.9
5  Nevada  3.2

>>> # SELEZIONE DI RIGHE: array logici e numerici interpretati come id di riga
>>> df[df.state == 'Ohio']      # logici
   state  year  pop
0   Ohio  2000  1.5
1   Ohio  2001  1.7
2   Ohio  2002  3.6
>>> df[:2]                      # slice
   state  year  pop
0   Ohio  2000  1.5
1   Ohio  2001  1.7

>>> # selezione di RIGHE E COLONNE: usare .loc/.iloc
>>> # df[df.state == 'Ohio', "pop"] # questo R style non funziona ma

```

Selezione più generale (.loc e .iloc) Direi che il meglio sia:

- abituarsi ad utilizzare loc/iloc: il primo richiede indici (numerici o stringa) e funziona selezionandoli, il secondo interi e funziona anche dove indici non sono presenti.
loc permette anche l'utilizzo di funzioni (che restituiscono indice, di riga o

colonna) che ricevono la versione più aggiornata del dataframe e si presta bene per il concatenamento di metodi.

Harrison preferisce `loc` per il codice di produzione

- specificare i criteri (per riga e colonne) separati da virgole tra parentesi quadre;
- se su riga o colonna si prende tutto, specificare una slice vuota `:`;

Da notare che la slice in `loc` vs `iloc`:

- funziona diversamente in `loc` ed `iloc`; con `loc` include l'elemento di arrivo, con `iloc` no
- meglio sortare l'indice prima di effettuare una selezione basata su slice; un template è `sort_index` e poi `loc/iloc` (eventualmente preceduti da `set_index`)
- la slice in `loc` con stringhe può essere basata su matching parziale

Alcuni esempi a seguire:

```
>>> data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...         "year": [2000, 2001, 2002, 2001, 2002, 2003],
...         "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> df = pd.DataFrame(data)      # senza indice di riga
>>> df_id = pd.DataFrame(data, index = list("abcdef")) # con indice di riga

>>> # =====
>>> # SELEZIONE RIGHE: con loc/iloc, UN SOLO INDICE interpretato di RIGA
>>> # =====
>>> df.loc[df.year==2001] # LOC: array booleani
   state  year  pop
1  Ohio   2001  1.7
3 Nevada  2001  2.4
>>> df.loc[[1,2,3]]      # LOC: se gli indici sono numerici possiamo usare numeri con loc
   state  year  pop
1  Ohio   2001  1.7
2  Ohio   2002  3.6
3 Nevada  2001  2.4
>>> df.loc[1:5:2]        # slice include elemento finale
   state  year  pop
1  Ohio   2001  1.7
3 Nevada  2001  2.4
5 Nevada  2003  3.2

>>> df_id.loc['a']        # LOC con indici stringa: singola stringa da serie
state    Ohio
year      2000
pop        1.5
Name: a, dtype: object
>>> df_id.loc[['a']]      # LOC con indici stringa: lista di stringhe da df
   state  year  pop
```

```
a Ohio 2000 1.5
```

```
>>> df.iloc[0]          # ILOC: Singola riga mediante indice come serie
state    Ohio
year     2000
pop       1.5
Name: 0, dtype: object
>>> df.iloc[[0]]        # Lista ritorna dataframe (anche di una sola riga)
   state year pop
0  Ohio  2000  1.5
>>> df.iloc[[2, 3]]
   state year pop
2  Ohio  2002  3.6
3  Nevada 2001  2.4
>>> df.iloc[1:5:2]      # slice esclude elemento finale
   state year pop
1  Ohio  2001  1.7
3  Nevada 2001  2.4

>>> # =====
>>> # SELEZIONE COLONNE: con loc/iloc separare con , e usare slice vuota
>>> # =====
>>> df.loc[:, 'pop']      # LOC: colonna singola come serie
0    1.5
1    1.7
2    3.6
3    2.4
4    2.9
5    3.2
Name: pop, dtype: float64
>>> df.loc[:, ['pop']]    # LOC: colonna singola come dataframe
   pop
0  1.5
1  1.7
2  3.6
3  2.4
4  2.9
5  3.2
>>> df.loc[:, ['year', 'pop']] # LOC: multiple via lista
   year pop
0  2000  1.5
1  2001  1.7
2  2002  3.6
3  2001  2.4
4  2002  2.9
5  2003  3.2
>>> df.loc[:, 'year': 'pop'] # LOC: multiple via slice
   year pop
0  2000  1.5
1  2001  1.7
2  2002  3.6
```

```
3 2001 2.4
4 2002 2.9
5 2003 3.2
```

```
>>> df.iloc[:, 2]      # ILOC: colonna singola (come serie)
```

```
0    1.5
1    1.7
2    3.6
3    2.4
4    2.9
5    3.2
```

```
Name: pop, dtype: float64
```

```
>>> df.iloc[:, [2]]    # ILOC: colonna singola (come dataframe)
```

```
   pop
0  1.5
1  1.7
2  3.6
3  2.4
4  2.9
5  3.2
```

```
>>> df.iloc[:, [1, 2]] # ILOC: colonne multiple mediante lista
```

```
   year  pop
0  2000  1.5
1  2001  1.7
2  2002  3.6
3  2001  2.4
4  2002  2.9
5  2003  3.2
```

```
>>> df.iloc[:, 1:3]    # colonne multiple mediante slice
```

```
   year  pop
0  2000  1.5
1  2001  1.7
2  2002  3.6
3  2001  2.4
4  2002  2.9
5  2003  3.2
```

```
>>> # =====
```

```
>>> # selezione RIGHE e COLONNE
```

```
>>> # =====
```

```
>>> df.loc[df.state == 'Ohio', "pop"]  # righe su logico e colonna (ritornata serie)
```

```
0    1.5
1    1.7
2    3.6
```

```
Name: pop, dtype: float64
```

```
>>> df.loc[df.state == 'Ohio', ["pop"]] # stessa cosa ma dataframe di ritorno
```

```
   pop
0  1.5
1  1.7
2  3.6
```

```
>>> df_id.loc[ 'a', ['pop', 'year']]  # singola riga, più colonne
```

```

pop      1.5
year     2000
Name: a, dtype: object
>>> df_id.loc[:"d", "year":"pop"]      # esempio con slice
      year  pop
a  2000   1.5
b  2001   1.7
c  2002   3.6
d  2001   2.4

>>> df.iloc[3, 1]                      # solo numerici in iloc
np.int64(2001)
>>> df.iloc[3, [1,2]]                  # selezionare piu colonne con iloc
      year     2001
pop      2.4
Name: 3, dtype: object
>>> df.iloc[:, :3]                     # uso slice
      state  year  pop
0    Ohio   2000   1.5
1    Ohio   2001   1.7
2    Ohio   2002   3.6
3  Nevada   2001   2.4
4  Nevada   2002   2.9
5  Nevada   2003   3.2

```

11.3.3 Selezione di righe con query

Il metodo `query` dei dataframe permette di selezionare righe usando una sintassi SQL-like; funziona bene nel chaining (a differenza degli indici, a meno che non si usino funzioni) essendo performata sulla versione più aggiornata del dataframe. Si può

- usare i metodi delle series nella query
- riferirsi a variabili dell'ambiente prefixandole con chiocciola

```

>>> df.query("(year >= 2002) and (~state.isna()) and (pop < 3.5)")
      state  year  pop
4  Nevada   2002   2.9
5  Nevada   2003   3.2

```

11.3.4 Selezione di colonne sulla base di tipo

si usi `.select_dtypes`

```

>>> df = pd.DataFrame({"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...                     "year": [2000, 2001, 2002, 2001, 2002, 2003],
...                     "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]})

>>> df.select_dtypes("number")
      year  pop
0  2000   1.5

```



```

1  2001  1.7
2  2002  3.6
3  2001  2.4
4  2002  2.9
5  2003  3.2
>>> df.select_dtypes("int")
      year
0  2000
1  2001
2  2002
3  2001
4  2002
5  2003
>>> df.select_dtypes("object")
      state
0    Ohio
1    Ohio
2    Ohio
3  Nevada
4  Nevada
5  Nevada

```

11.3.5 Accesso a singoli numeri: `.at`, `.iat`

Per estrarre *singoli numeri* appartenenti ad un dataframe (senza portarsi a dietro tutta la sua struttura) si può utilizzare gli accessori `.at` o `.iat` (fornendo una coppia indice di riga e colonna), Sono equivalenti di `.loc` e `.iloc`: `at` richiede labels, `iat` richiede interi

```

>>> df = pd.DataFrame(np.arange(6).reshape(3, 2), columns=['A', 'B'])

>>> df
   A  B
0  0  1
1  2  3
2  4  5
>>> df.at[2, "B"]
np.int64(5)
>>> df.iat[2, 1]
np.int64(5)

```

11.3.6 Aggiunta di colonne (assegnazione, insert, assign)

Le colonne possono essere create

- usare il metodo `assign` (*dovrebbe essere preferito*) per risparmiare digitazione, usare funzioni (e anche concatenarle). Con `assign` se i nomi variabili sono già esistenti le corrispondenti variabili verranno sovrascritte mentre se inesistenti create. `assign` crea sempre una copia (non modifica) il dataset di partenza quindi il risultato dovrà essere salvato. `assign` dovrebbe essere preferito perché

- utilizzando funzioni si prende in input il dataframe allo stato attuale/aggiornato
- meno spaghetti code rispetto a selezioni con variabili temporanee e permette concatenazione di codice rendendolo più leggibile

Tipicamente

- ogni riga terminerà con `astype` per specificare/coercire il tipo della variabile create
- alla chiusura del metodo si postporrà un `.loc` per selezionare esplicitamente lista le variabili effettivamente da tenere (meglio di `.drop` per cancellare)
- assegnando a nomi di colonna inesistenti (le relative colonne verranno create alla fine); torna comoda la sintassi `df['variabile']=...`
- usando il metodo `insert` per inserire in un determinato punto

```
>>> data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...         "year": [2000, 2001, 2002, 2001, 2002, 2003],
...         "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> df = pd.DataFrame(data, index = list("abcdef")) # con indice di riga

>>> # best method evah
>>> sel = df.columns.to_list() + ["test", "yearp1"]
>>> df2 = df.assign(# uso di assign: deve essere salvato
...               test = range(5, 11),
...               # le funzioni prendono il df attuale come input
...               yearp1 = lambda df_: (df_.year + 1).astype("int64[pyarrow]"),
...               # qui si può già usare yearp1
...               yearp2 = lambda df_: (df_.yearp1 + 1).astype("float")
...               ).loc[:, sel]

>>> # other classic spaghetti/shitty methods
>>> df['list'] = range(6) # deve rispettare lunghezza
>>> df['nparray'] = rng.random(6) # deve rispettare lunghezza
>>> df['series'] = pd.Series([10, 20, 30, 40], # con Series rispettati gli indici
...                        index = ['a', 'b', 'y', 'z'])
>>> df['constant'] = 3 # valore puntuale: funziona broadcasting
>>> df[['foo', 'bar']] = [0, 1] # broadcasting two columns magic

>>> df.insert(0, # insert in posizione specifica (prima) (non necessario df =
...           "nomevar", # nome variabile
...           3) # valore (es scalar, pd.Series, o array)

>>> df
   nomevar  state  year  pop  list  nparray  series  constant  foo  bar
a         3   Ohio  2000  1.5     0  0.949734     10.0         3    0    1
```

b	3	Ohio	2001	1.7	1	0.387793	20.0	3	0	1
c	3	Ohio	2002	3.6	2	0.596787	NaN	3	0	1
d	3	Nevada	2001	2.4	3	0.603423	NaN	3	0	1
e	3	Nevada	2002	2.9	4	0.320868	NaN	3	0	1
f	3	Nevada	2003	3.2	5	0.537778	NaN	3	0	1

11.3.7 Modifica di valori (indexing e assegnazione: loc, iloc, at, iat)

La modifica di colonne/righe/dati avviene mediante assegnazione a nomi esistenti; meglio utilizzare loc o iloc

```
>>> df.loc["a", "constant"] = 5 # singolo valore
>>> df.loc[:, "series"] = 1     # intera colonna
>>> df.iloc[1, :] = 2          # intera riga, volendo
>>> df
```

	nomevar	state	year	pop	list	nparray	series	constant	foo	bar
a	3	Ohio	2000	1.5	0	0.949734	1.0	5	0	1
b	2	2	2	2.0	2	2.000000	2.0	2	2	2
c	3	Ohio	2002	3.6	2	0.596787	1.0	3	0	1
d	3	Nevada	2001	2.4	3	0.603423	1.0	3	0	1
e	3	Nevada	2002	2.9	4	0.320868	1.0	3	0	1
f	3	Nevada	2003	3.2	5	0.537778	1.0	3	0	1

at e iat possono essere usati anche come lvalue per assegnazioni, similmente a loc/iloc.

```
>>> df = pd.DataFrame(np.arange(6).reshape(3, 2), columns=['A', 'B'])
>>> df.at[2, "B"] = 15321
>>> df
```

	A	B
0	0	1
1	2	3
2	4	15321

11.3.8 Rimozione righe/colonne (drop, del)

Per:

- righe o colonne si usa il metodo `drop`, specificando tra parentesi gli assi e assegnando il risultato (o specificando `inplace`)
- le colonne si può usare `del`

```
>>> data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...         "year": [2000, 2001, 2002, 2001, 2002, 2003],
...         "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> df = pd.DataFrame(data) # senza indice di riga
>>> df_id = pd.DataFrame(data, index = list("abcdef")) # con indice di riga

>>> df.drop([0, 1]) # rimozione prime due righe
      state  year  pop
```

```

2   Ohio  2002  3.6
3   Nevada 2001  2.4
4   Nevada 2002  2.9
5   Nevada 2003  3.2
>>> df_id.drop(index = "d") # rimozione riga usando indice
      state year pop
a   Ohio  2000  1.5
b   Ohio  2001  1.7
c   Ohio  2002  3.6
e  Nevada 2002  2.9
f  Nevada 2003  3.2
>>> df.drop(columns = ['year', 'pop']) # rimozione colonne
      state
0   Ohio
1   Ohio
2   Ohio
3  Nevada
4  Nevada
5  Nevada

>>> del df['state']
>>> df
      year  pop
0  2000  1.5
1  2001  1.7
2  2002  3.6
3  2001  2.4
4  2002  2.9
5  2003  3.2

```

11.3.9 Rinominare indici/colonne (rename)

Si usa il metodo `rename`, che può prendere dict e funzioni e applicarle a righe o colonne

```

>>> data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...         "year": [2000, 2001, 2002, 2001, 2002, 2003],
...         "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
>>> df = pd.DataFrame(data, index = list("abcdef")) # senza indice di riga

>>> # utilizzo di un dict sulle colonne, str.upper sugli index/righe
>>> ft = {'state': 'stato', 'pop': 'popolazione'}
>>> df = df.rename(index = str.upper, columns = ft)
>>> df
      stato year popolazione
A   Ohio  2000          1.5
B   Ohio  2001          1.7
C   Ohio  2002          3.6
D  Nevada 2001          2.4
E  Nevada 2002          2.9
F  Nevada 2003          3.2

```

11.3.10 Funzionalità per indici, reindexing, MultiIndex

11.3.10.1 Creare indici da colonne e viceversa (set_index, reset_index)

Si usa:

- `set_index` se si vuol creare indici di riga a partire da una/più colonne (specificando tra parentesi la variabile/lista di variabili). Per non eliminare le colonne specificate come index si usa il parametro `drop = False`.
- `reset_index` salva gli indici in colonne del dataframe e li sostituisce con un progressivo numerico

```
>>> df = pd.DataFrame({'a': range(7), 'b': range(7, 0, -1),
...                    'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
...                    'd': [0, 1, 2, 0, 1, 2, 3]})
>>> df
```

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

```
>>> # Setting di indici sulla base di colonne
```

```
>>> df2 = df.set_index(['c', 'd'])
```

```
>>> df2
```

		a	b
one	d		
	0	0	7
	1	1	6
two	2	2	5
	0	3	4
	1	4	3
	2	5	2
	3	6	1

```
>>> # Da indice a colonna e progressivo numerico come indice
```

```
>>> df2.reset_index()
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

11.3.10.2 Reindexing (reindex, loc)

Serve per riordinare e/o selezionare righe e colonne:

- utilizzare `reindex`: di default lavora sulle righe a meno che non si specifichi `columns`
- se si desidera un modo safe per effettuare reindexing utilizzare `loc`: funziona solo se gli indici forniti esistono già nel `DataFrame` (e non crea valori missing)

```
>>> data = np.arange(9).reshape((3,3))
>>> id = ['a', 'c', 'd']
>>> col = ['Ohio', 'Texas', 'California']
>>> df = pd.DataFrame(data, index = id, columns = col)
>>> df
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
>>> df2 = df.reindex(['a', 'b', 'c', 'd']) # reindexing di riga
>>> df2 # b missing perché non disponibile nei dati di partenza
```

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

```
>>> states = ['Texas', 'Utah', 'California'] # reindexing di colonna
>>> df.reindex(columns = states) # si cancella Ohio, non richiesto
```

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

```
>>> df.loc[["a", "d", "c"], ["California", "Texas"]] # safe reindexing
```

	California	Texas
a	2	1
d	8	7
c	5	4

11.3.10.3 MultiIndex

Sia righe che colonne possono avere indexing multiplo e con nomi

```
>>> df = pd.DataFrame(np.arange(12).reshape((4, 3)),
...                    index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
...                    columns=[['Ohio', 'Ohio', 'Colorado'],
...                              ['Green', 'Red', 'Green']])
>>> df.index.names = ["key1", "key2"]
>>> df.columns.names = ["state", "color"]
>>> df
```

state		Ohio		Colorado
color		Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Anche sui nomi di indici di colonna è possibile effettuare selezioni

```
>>> df["Ohio"]
color      Green  Red
key1 key2
a      1      0    1
      2      3    4
b      1      6    7
      2      9   10
```

Per conoscere il numero di livelli di indici

```
>>> df.index.nlevels
2
```

Invertire gli indici Può essere necessario a volte cambiare l'ordine degli indici di un'asse (ad esempio indice più interno portarlo fuori). Questo si fa mediante `swaplevel`

```
>>> df.swaplevel("key1", "key2")
state      Ohio      Colorado
color      Green Red      Green
key2 key1
1      a      0    1      2
2      a      3    4      5
1      b      6    7      8
2      b      9   10     11
```

11.3.11 Elaborazione allineata

Si possono effettuare operazioni algebriche tra dataframe, ma nel caso sono gli indici a determinare cosa viene messo in relazione con cosa e se una posizione non è disponibile in entrambi gli operandi viene restituito dato mancante

```
>>> df = pd.DataFrame({"x" : [1,2,3,4], "y": [5,6,7,8], "z": [9,10,11,12]},
...                    index = list("abcd"))
>>> df1 = df.loc["a":"c", "x":"y"]
>>> df2 = df.loc["b":"d", "y":"z"]
>>> df1
   x  y
a  1  5
b  2  6
c  3  7
>>> df2
   y  z
b  6  7
c  7  8
d  8  9
```

```

b 6 10
c 7 11
d 8 12
>>> df1 + df2
      x    y    z
a NaN  NaN NaN
b NaN  12.0 NaN
c NaN  14.0 NaN
d NaN   NaN NaN

```

L'allineamento per indice avviene anche per operazioni tra dataframe e series,

```

>>> df
      x  y  z
a  1  5  9
b  2  6 10
c  3  7 11
d  4  8 12
>>> df.mean()
x      2.5
y      6.5
z     10.5
dtype: float64
>>> df.std()
x      1.290994
y      1.290994
z      1.290994
dtype: float64

>>> (df - df.mean()) / df.std()
      x      y      z
a -1.161895 -1.161895 -1.161895
b -0.387298 -0.387298 -0.387298
c  0.387298  0.387298  0.387298
d  1.161895  1.161895  1.161895

>>> # verifica che cio sia cosi

>>> mu = df.mean()[1:] # media di y e z solamente
>>> std = df.std()[1:] # sd di x e y solamente
>>> (df - mu) / std
      x      y      z
a NaN -1.161895 NaN
b NaN -0.387298 NaN
c NaN  0.387298 NaN
d NaN  1.161895 NaN

```

Nel caso vi siano **indici ripetuti** in un dataframe questo si ripercuoterà sui risultati (avendo indici ripetuti anche li, essendo l'elaborazione allineata per indice)

11.3.12 Coercizione di tipi (astype, transform)

Possiamo usare `astype` specificando il mapping di formato in un dict.

```
>>> df = pd.DataFrame({"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...                     "year": [str(y) for y in [2000, 2001, 2002, 2001, 2002, 2003]],
...                     "pop": [str(p) for p in [1.5, 1.7, 3.6, np.nan, 2.9, 3.2]],
...                     "adate": ["2020-01-02", "2021-01-01", "2022-01-02"] * 2
...                     }) # all strings/"object"
>>> ft = {
...     "state": "category",
...     "year": "Int16",
...     "pop": "Float64",
...     "adate": "datetime64[ns]" # soluzione provvisoria, credo
... }                                     # https://wesmckinney.com/book/data-cleaning.html#pandas-ext-

>>> df2 = df.astype(ft)
>>> df2.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
#   Column   Non-Null Count  Dtype
---  -
0   state     6 non-null      category
1   year      6 non-null      Int16
2   pop       6 non-null      Float64
3   adate     6 non-null      datetime64[ns]
dtypes: Float64(1), Int16(1), category(1), datetime64[ns](1)
memory usage: 382.0 bytes
```

Altrimenti si veda `transform` per applicare funzioni custom (di coercizione) in blocco nella sezione 11.3.14.1

11.3.13 Aggregazione (agg)

Possiamo passare al metodo `.agg`

- una lista con cose/aggregazioni per farle eseguire (su tutte le colonne di default). Ad esempio per la conta di non missing, numero di valori unici (missing inclusi), somma valori e primo valore

```
>>> analysis = ["count", "size", "sum", lambda col: col.loc[0]]
>>> df = pd.DataFrame({"a": [1,2,3], "b": [4,5,6], "c": [7,8,9]})
>>> df.agg(analysis)
           a    b    c
count      3    3    3
size       3    3    3
sum        6   15   24
<lambda>   1    4    7
```

- un dict che specifichi per variabile quali analisi fare

```
>>> analysis_dict = {
```

```

...     "a": ["count", "size"],
...     "b": ["sum", lambda col: col.loc[0]]}
>>> df.agg(analysis_dict)
      a      b
count  3.0  NaN
size   3.0  NaN
sum    NaN  15.0
<lambda> NaN   4.0

```

11.3.14 Applicazione di funzioni

11.3.14.1 A righe e colonne (apply, transform)

Per applicare

- diverse funzioni a diverse colonne utilizzare **transform**: si genera il nuovo in un dataframe (che deve esser bindato al vecchio)

```

>>> def two(x):
...     return x*2
...
>>> def three(x):
...     return x*3
...
>>> df = pd.DataFrame({"a": [1,2,3], "b": [4,5,6], "c": [7,8,9]})

>>> t = {"a": two, "b": three}
>>> df.transform(t)
      a      b
0  2  12
1  4  15
2  6  18

```

- una (singola) funzione ad ogni riga o colonna usare **apply**:

```

>>> df = pd.DataFrame({"a" : [1, 2, 3], "b" : [4, 5, 6]})

>>> # ----- funzione che restituisce uno scalare
>>> def f(x):
...     return x.mean()
...
>>> df.apply(f) # default: applica a tutte colonne
a      2.0
b      5.0
dtype: float64
>>> df.apply(f, axis = 1) # applica a righe (cicla su 1, le colonne)
0      2.5
1      3.5
2      4.5
dtype: float64

>>> # ----- funzione che restituisce una serie

```

```

>>> def desc(x):
...     return pd.Series([x.min(), x.max(), x.mean(), x.median()],
...                       index=["min", "max", "mean", "median"])
...
>>> df.apply(desc)           # colonna
      a    b
min  1.0  4.0
max  3.0  6.0
mean  2.0  5.0
median 2.0  5.0
>>> df.apply(desc, axis = 1) # riga
      min max mean median
0  1.0  4.0  2.5   2.5
1  2.0  5.0  3.5   3.5
2  3.0  6.0  4.5   4.5

```

11.3.14.2 A tutto il DataFrame (map e pipe)

map applica una funzione a tutti gli elementi del dataframe

```

>>> rng = np.random.default_rng(665)
>>> df = pd.DataFrame(rng.standard_normal((2, 2)))
>>> df
      0      1
0  0.259614 -0.072077
1 -0.093439 -1.180275
>>> df.map(lambda x: -1 if x < 0 else 1) # sign function
      0  1
0  1 -1
1 -1 -1

```

pipe passa l'intero dataframe a una funzione che si attende di lavorare con dataframe

```

>>> #effective pandas 2 pag 127
>>> agecl = pd.Series(["0-10", "11-15", "11-15", "61-65", "46-50"])

>>> ( agecl
...   .str.split('-', expand = True) # splits in a two cols dataframe
...   .astype(int)
...   .pipe( # passa il dataframe a un generatore di numeri casuali tra le
...         # colonne
...         lambda df_ : pd.Series(
...             # qui possiamo usare df_ per riferirci al dataframe
...             np.random.randint(df_.iloc[:, 0], df_.iloc[:, 1]),
...             index=df_.index
...         ))
... )
0      4
1     13
2     14
3     64

```

```
4      49
dtype: int64
```

11.3.14.3 Sfruttando metodi delle colonne (Series.map)

Dato che le colonne sono `Series` possiamo applicare `map` o `replace` come già visto per queste:

```
>>> df = pd.DataFrame({"a" : list("abcd"),
...                    "b" : [1, 2, 3, 4]})

>>> # mapping di funzione
>>> df["bgt2"] = df["b"].map(lambda x: x > 2)
>>> df
   a  b  bgt2
0  a  1  False
1  b  2  False
2  c  3   True
3  d  4   True
```

Non di funzioni ma si possono applicare `dict` con `map` (recode completi) o `transform` (recode incompleti)

```
>>> # mapping di dict per recode completi
>>> ft = {'a' : "a-b", "b" : "a-b", "c": "c-d", "d": "c-d"}
>>> df["agroup"] = df["a"].map(ft)

>>> # replace per recode incompleti
>>> ft = {4: 3}
>>> df["bre-coded"] = df.b.replace(ft)
```

11.3.15 Ciclo su righe/colonne (iterrows, items)

11.3.15.1 Righe (itertuples)

Si usa:

- `itertuples`: ritorna una tuple con nome, che preserva i tipi ed è più veloce (indice in posizione 0)

```
>>> df = pd.DataFrame({'num_legs': [4, 2],
...                    'num_wings': [0, 2]},
...                    index=['dog', 'hawk'])

>>> for row in df.itertuples():
...     print("id =", row[0],
...           "first_elem =", row[1],
...           "second_elem =", row[2],
...           "first_named =", row.num_legs,
...           "second_named =", row.num_wings)
...
id = dog first_elem = 4 second_elem = 0 first_named = 4 second_named = 0
id = hawk first_elem = 2 second_elem = 2 first_named = 2 second_named = 2
```

- `iterrows`: ritorna una coppia label e `Series` (di minor interesse)

11.3.15.2 Colonne (for secco, items)

Si usa

- un ciclo for secco restituisce il nome della variabile

```
>>> df = pd.DataFrame({'species': ['bear', 'bear', 'marsupial'],
...                     'population': [1864, 22000, 80000]},
...                     index=['panda', 'polar', 'koala'])

>>> for var in df:
...     print(var)
...
species
population
```

- il metodo items che restituisce la tuple con nome variabile e contenuto

```
>>> for varname, content in df.items():
...     print(f'varname: {varname}')
...     print(f'type: {type(varname)}')
...     print(f'content:\n{content}', sep='\n')
...
varname: species
type: <class 'str'>
content:
panda      bear
polar      bear
koala      marsupial
Name: species, dtype: object
varname: population
type: <class 'str'>
content:
panda      1864
polar      22000
koala      80000
Name: population, dtype: int64
```

11.3.16 Merge (merge)

Il merge:

- funziona di default sulla base delle colonne comune presenti in entrambi i dataset; altrimenti specificare i nomi delle variabili comuni mediante `on` (se i due dataframe hanno variabili di merge con lo stesso nome) oppure ordinatamente `left_on` e `right_on`;
- di default ritorna le righe dove la chiave è presente in entrambi i dataset (*inner join*): per altri tipi si possono specificare nel parametro `how` (es `left`, `right` ed `outer`)
- se si desidera che la chiave di merge siano gli indici (di riga) dei dataframe, invece di variabili, bisogna passare `left_index = True` o `right_index = True`

```

>>> df1 = pd.DataFrame({'key': ['a', 'b', 'b', 'c', 'c', 'c'],
...                      'data1': range(6)})
>>> df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
...                      'data2': ['x', 'y', 'z']})
>>> df1
   key  data1
0    a      0
1    b      1
2    b      2
3    c      3
4    c      4
5    c      5
>>> df2
   key data2
0    a     x
1    b     y
2    d     z

>>> pd.merge(df1, df2)                                     # inner with defaults ("key" is common, used)
   key  data1 data2
0    a      0     x
1    b      1     y
2    b      2     y
>>> pd.merge(df1, df2, how = 'left') # tiene dataset di sx, integra con quello di dx
   key  data1 data2
0    a      0     x
1    b      1     y
2    b      2     y
3    c      3  NaN
4    c      4  NaN
5    c      5  NaN
>>> pd.merge(df1, df2, how = 'right') # tiene dataset di dx, integra con quello di sx
   key  data1 data2
0    a    0.0     x
1    b    1.0     y
2    b    2.0     y
3    d    NaN     z
>>> pd.merge(df1, df2, how = 'outer') # tieni tutto
   key  data1 data2
0    a    0.0     x
1    b    1.0     y
2    b    2.0     y
3    c    3.0  NaN
4    c    4.0  NaN
5    c    5.0  NaN
6    d    NaN     z

```

Esempio con chiavi su variabili differenti:

```

db = pd.merge(db, regions, how = 'left',
               left_on = 'st', right_on = 'state code', # nomi delle variabili
               suffixes = (None, None)) # non aggiungere i suffissi _x, _y

```

Per un esempio di merging di diversi dataset aventi le stesse chiavi, al fine di ottenerne solamente uno si può usare la programmazione funzionale e `reduce`

```
from functools import reduce

def merger(x, y):
    return pd.merge(x, y,
                     left_on=['id', 'time'],
                     right_on=['id', 'time'],
                     suffixes=(None, None))
```

```
df = reduce(merger, [demo, lav, sal, bis])
```

11.3.17 Binding (concat)

Si effettua mediante la funzione `concat`

```
>>> df = pd.DataFrame({'key': ['a', 'b', 'd'],
...                     'data2': ['x', 'y', 'z']})

>>> pd.concat([df, df])                                     # di default di riga
   key data2
0    a     x
1    b     y
2    d     z
0    a     x
1    b     y
2    d     z
>>> pd.concat([df, df], axis = 'columns')                  # se no di colonna
   key data2 key data2
0    a     x    a     x
1    b     y    b     y
2    d     z    d     z
```

11.3.18 Reshape

11.3.18.1 Sulla base di index (stack,unstack)

L'indexing gerarchico torna necessario nel reshape. Vi sono due metodi principali:

- `stack` ruota le colonne a righe (va verso long)
- `unstack` ruota da righe a colonne (va verso wide)

In questi casi utile un `name` agli indici

```
>>> df = pd.DataFrame({"one"   : [0, 3],
...                     "two"   : [1, np.nan],
...                     "three" : [2, 5],
...                     "four"  : [np.nan, 6]},
...                     index = pd.Index(["Ohio", "Colorado"], name = "state"))
```

```

>>> df
      one  two  three  four
state
Ohio      0  1.0      2  NaN
Colorado  3  NaN      5  6.0

>>> df.stack()                                # stack: porre in formato long
state
Ohio      one      0.0
          two      1.0
          three     2.0
Colorado  one      3.0
          three     5.0
          four      6.0
dtype: float64
>>> df.stack(future_stack = True) # versione futura da struttura simmetrica
state
Ohio      one      0.0
          two      1.0
          three     2.0
          four     NaN
Colorado  one      3.0
          two      NaN
          three     5.0
          four      6.0
dtype: float64
>>> df_long = df.stack()                    # saving results

>>> df_long.unstack()                        # unstack verso wide (default level = 1: usa sec
      one  two  three  four
state
Ohio    0.0  1.0    2.0  NaN
Colorado 3.0  NaN    5.0  6.0

>>> df_long.unstack(level = 0)                # unstack usando la prima colonna di indici
state Ohio Colorado
one    0.0      3.0
two    1.0      NaN
three  2.0      5.0
four   NaN      6.0

>>> df_long.unstack(level = 'state') # unstack usando i name
state Ohio Colorado
one    0.0      3.0
two    1.0      NaN
three  2.0      5.0
four   NaN      6.0

```

11.3.18.2 Senza index (pivot, melt)

Porre in wide abbiamo un dataset in formato long con tre colonne, id/tempo, variabile e valore lo possiamo mettere in formato wide con il metodo `pivot` dei

`DataFrame`. È equivalente a creare un indice gerarchico con `set_index` e poi chiamare `unstack`

```
>>> df = pd.DataFrame({'id' : ['one', 'one', 'one', 'two', 'two', 'two'],
...                     'var' : ['A', 'B', 'C', 'A', 'B', 'C'],
...                     'val1': [1, 2, 3, 4, 5, 6],
...                     'val2': ['x', 'y', 'z', 'q', 'w', 't']})

>>> # specificando la singola variabile di interesse
>>> df.pivot(index = 'id', columns = 'var', values = 'val1')
var  A  B  C
id
one  1  2  3
two  4  5  6

>>> # non specificando prende tutto
>>> df.pivot(index = 'id', columns = 'var')
      val1      val2
var  A  B  C      A  B  C
id
one   1  2  3      x  y  z
two   4  5  6      q  w  t
```

Porre in long L'operazione inversa si ha con `pd.melt`

```
>>> df = pd.DataFrame({"key": ["foo", "bar", "baz"],
...                     "A": [1, 2, 3],
...                     "B": [4, 5, 6],
...                     "C": [7, 8, 9]})

>>> df
   key  A  B  C
0  foo  1  4  7
1  bar  2  5  8
2  baz  3  6  9

>>> # poni in long tutto
>>> long = pd.melt(df, id_vars = 'key')
>>> long
   key variable  value
0  foo         A      1
1  bar         A      2
2  baz         A      3
3  foo         B      4
4  bar         B      5
5  baz         B      6
6  foo         C      7
7  bar         C      8
8  baz         C      9

>>> # poni una selezione
```

```
>>> long2 = pd.melt(df, id_vars = 'key', value_vars = ["A", "B"])
>>> long2
   key variable  value
0  foo         A      1
1  bar         A      2
2  baz         A      3
3  foo         B      4
4  bar         B      5
5  baz         B      6
```

11.3.19 Test di appartenenza (in, isin)

Si ha:

- `in` applicato ad un DataFrame testa la presenza di una colonna avente un determinato nome
- `isin` cerca nel contenuto del dataframe

```
>>> df = pd.DataFrame({"a": [1, 2, 3],
...                    "b": list("xyz"),
...                    "c": [2,3,4]})
>>> df
   a  b  c
0  1  x  2
1  2  y  3
2  3  z  4

>>> # in per check nomi colonna
>>> 'a' in df
True
>>> '1' in df
False

>>> # ==, != o isin per check valori
>>> df == 2
   a  b  c
0 False False True
1  True False False
2 False False False
>>> df == "z"
   a  b  c
0 False False False
1 False False False
2 False  True False
>>> df.isin([2]) # lista: cerca in tutte le colonne
   a  b  c
0 False False True
1  True False False
2 False False False
>>> df.isin([2, "z"]) # lista 2
   a  b  c
```

```

0 False False True
1 True False False
2 False True False
>>> df.isin({"c" : [3]})# dict: cerca in alcune colonne specificate
      a      b      c
0 False False False
1 False False  True
2 False False False
>>> ~df.isin(["z"])      # negare una ricerca
      a      b      c
0 True  True  True
1 True  True  True
2 True False  True

```

11.3.20 Dati mancanti (count, isna, notna, dropna, fillna)

Le funzioni ritornano una copia modificata che va eventualmente salvata:

```

>>> df = pd.DataFrame([[1., 6.5, 3.],
...                     [1., np.nan, np.nan],
...                     [np.nan, np.nan, np.nan],
...                     [np.nan, 6.5, 3.]])
>>> df
      0      1      2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

>>> # testing e statistiche descrittive
>>> df.isna() # testing 1
      0      1      2
0 False False False
1 False  True  True
2  True  True  True
3  True False False
>>> df.notna() # testing 2
      0      1      2
0  True  True  True
1  True False False
2 False False False
3 False  True  True
>>> df.isna().sum()
0      2
1      2
2      2
dtype: int64
>>> df.isna().mean().sort_values()
0      0.5
1      0.5
2      0.5

```

```
dtype: float64
```

```
>>> # rimozione di dati mancanti: righe
>>> df.dropna() # eliminazione righe con anche un solo NA
      0      1      2
0  1.0  6.5  3.0
>>> df.dropna(how = 'all') # eliminazione righe completamente NA
      0      1      2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0

>>> # rimozione di dati mancanti: colonne
>>> df[4] = np.nan
>>> df.dropna(axis = "columns", how = "all") # eliminazione colonne completamente NA
      0      1      2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

>>> # riempimento di dati mancanti
>>> df.fillna(99)
      0      1      2      4
0  1.0  6.5  3.0  99.0
1  1.0  99.0  99.0  99.0
2  99.0  99.0  99.0  99.0
3  99.0  6.5  3.0  99.0
```

11.3.21 Gestione duplicati (duplicated, drop_duplicates)

`duplicated` ritorna un vettore di booleani per indicare se una riga è duplicata; `drop_duplicates` ritorna un data frame senza duplicati. Alcune opzioni:

- di default tutte le colonne sono considerate, alternativamente specificare `subset`;
- se si usa `keep` si può specificare se tenere i primi elementi duplicati ("`first`", di default), gli ultimi ("`last`") oppure eliminare tutto (`False`).

```
>>> df = pd.DataFrame({"k1": ["one", "two"] * 3 + ["two"],
...                     "k2": [1, 1, 2, 3, 3, 4, 4]},
...                     index = list("abcdef"))
>>> df
      k1  k2
a  one   1
a  two   1
b  one   2
c  two   3
d  one   3
e  two   4
f  two   4
```

```

>>> # Ricerca
>>> df.duplicated()          # ricerca complessiva
a    False
a    False
b    False
c    False
d    False
e    False
f     True
dtype: bool
>>> df.duplicated('k1')     # ricerca in singola colonna
a    False
a    False
b     True
c     True
d     True
e     True
f     True
dtype: bool

>>> # Eliminazione
>>> df.drop_duplicates()    # complessiva
   k1  k2
a  one  1
a  two  1
b  one  2
c  two  3
d  one  3
e  two  4
>>> df.drop_duplicates(subset = ["k1"]) # su singola colonna
   k1  k2
a  one  1
a  two  1
>>> df.drop_duplicates(subset = ["k1"], keep = 'last') # tenere ultimi elementi
   k1  k2
d  one  3
f  two  4

>>> # duplicati a livello di indici
>>> df.index.duplicated()
array([False,  True, False, False, False, False, False])

```

11.3.22 Sorting di righe/colonne (sort_values, sort_index)

Si usa:

- `sort_values` per ordinare sulla base di valori di colonna (è anche possibile fornire una funzione in `key` che elabora cose e restituisce qualcosa sulla base del quale fare ordinamento);
- `sort_index` per ordinare sulla base degli indici di riga o nomi colonna

```

>>> rng = np.random.default_rng(23)
>>> df = pd.DataFrame({'g': ["x", "y", "x", "y"],
...                    'y' : rng.standard_normal((4)),
...                    'z' : rng.standard_normal((4))},
...                    index = list("bacd"))

>>> # ordinare righe per valori contenuti
>>> df.sort_values(by = 'y')
   g      y      z
d  y -2.318936  0.605966
c  x -0.057990  0.909921
a  y  0.217601 -2.126280
b  x  0.553261  0.431494
>>> df.sort_values(by = ['g', 'y'])
   g      y      z
c  x -0.057990  0.909921
b  x  0.553261  0.431494
d  y -2.318936  0.605966
a  y  0.217601 -2.126280
>>> df.sort_values(by = ['g', 'y'], ascending = [True, False])
   g      y      z
b  x  0.553261  0.431494
c  x -0.057990  0.909921
a  y  0.217601 -2.126280
d  y -2.318936  0.605966

>>> # ordinare righe sulla base di indici
>>> # df.set_index("some_variable").sort_index()
>>> df.sort_index()
   g      y      z
a  y  0.217601 -2.126280
b  x  0.553261  0.431494
c  x -0.057990  0.909921
d  y -2.318936  0.605966

>>> # ordinare colonne alfabeticamente per nome variabile
>>> df.sort_index(axis="columns")
   g      y      z
b  x  0.553261  0.431494
a  y  0.217601 -2.126280
c  x -0.057990  0.909921
d  y -2.318936  0.605966

```

11.4 Data I/O

L'importazione avviene con le funzioni `pd.read_*`, l'esportazione usa i metodi dei `DataFrame` `to_*`. Nel seguito i casi più notevoli.

Importazione

```

# Lettura di file testuali
df = pd.read_csv('path.csv') # separatore virgola di default
df = pd.read_csv('path.csv', sep = ';') # separatore punto e virgola
df = pd.read_csv('path.csv', sep = '\t') # separatore tab

# Alcuni binari notevoli
df = pd.read_excel('path.xlsx', sheet_name = 'asd') # file xls
df = pd.read_pickle('path.pkl') # formato binario Python
df = pd.read_feather('path.feather') # formato interscambio R/Python
df = pd.read_sas('path.sas7bdat') # un dataset SAS (in uno dei formati custom di SAS)
df = pd.read_spss('path.sav') # Read a data file created by SPSS
df = pd.read_stata('path.dta') # formato stata

```

Esportazione su file

```

# Scrittura di file testuali
df.to_csv('path.csv')
df.to_csv('path.csv', index = False) # non scrivere l'indice
df.to_csv('path.csv', sep = ";", quoting = csv.QUOTE_NONNUMERIC) # read.csv2

# Alcuni binari notevoli
df.to_excel('path.xlsx')
df.to_feather('path.feather') # arrow::read_feather(path.feather) per leggere
df.to_pickle('path.pkl')
df.to_stata('path.dta')

```

Utilities per il reporting Le seguenti restituiscono stringhe che debbono essere stampate

```

>>> df = pd.DataFrame({"x": list("abc"), "y" : [1,2,3]})

>>> # Markdown: serve il pacchetto tabulate
>>> # print(df.to_markdown())

>>> # Latex
>>> print(df.to_latex())
\begin{tabular}{llr}
\toprule
& x & y \\
\midrule
0 & a & 1 \\
1 & b & 2 \\
2 & c & 3 \\
\bottomrule
\end{tabular}

```

11.5 Cookbook

11.5.1 Stampare localmente tutto il contenuto di un DataFrame

Usare il context manager `pd.option_context` con le seguenti opzioni

```
with pd.option_context('display.max_rows', None, 'display.max_columns', None):  
    stampa_df()
```


Capitolo 12

Matplotlib

Contents

12.1	Introduzione	209
12.2	Salvataggio figura	210
12.3	Impostazione layout figura ed esempi	212
12.3.1	Layout standard	212
12.3.2	Layout custom	214
12.4	Configurazioni	217
12.4.1	Ottenimento e modifica	217
12.4.2	Ripristino impostazioni default	218
12.4.3	Cambiare stile	218
12.5	Grafici utili	218
12.5.1	Linee	219
12.5.2	Diagramma a barre	219
12.5.3	Istogramma	220
12.5.4	Scatterplot	220
12.5.5	Matrice di scatterplot	223
12.5.6	Boxplot	223
12.5.7	Correlation matrix	226

```
import numpy as np
import pylabmisc as lb
import rdatasets
rng = np.random.default_rng(123)

# template importazione matplotlib
import matplotlib.pyplot as plt
```

12.1 Introduzione

Alcuni punti di base:

- vi sono due stili di plotting in `matplotlib`: uno classico state-based che imita matlab e uno object oriented. Preferiamo questo secondo.

Metodo	Descrizione
<code>plot</code>	linee/scatterplot
<code>hist</code>	istogramma
<code>bar</code>	diagramma a barre
<code>scatter</code>	scatterplot più flessibile/lento
<code>errorbar</code>	intervalli di confidenza
<code>set</code>	imposta tutti i seguenti in una unica chiamata
<code>set_title</code>	imposta il titolo
<code>set_xlabel, set_ylabel</code>	imposta il titolo degli assi
<code>set_xticks, set_yticks</code>	imposta la posizione dei ticks sugli assi
<code>set_xticklabels, set_yticklabels</code>	imposta l'etichetta dei ticks
<code>set_xlim, set_ylim</code>	impostare i limiti degli assi/zoom figura
<code>legend</code>	aggiunta di legenda
<code>text</code>	aggiunta di testo

Tabella 12.1: Metodi utili di `ax`

- una sequenza template per creare gli oggetti necessari sino a salvare la mostrare e salvare la ha il seguente template

```
fix, ax = plt.subplots() # setup fig and axis
# ...                  # do the actual plotting
plt.show()              # show the plot
fig.savefig("/tmp/first_plot.png", dpi=600) # save as png
```

- concetti fondanti:
 - una *figure* è il container di una immagine;
 - una figure può contenere al suo interno uno o più *axes*: questi sono gli effettivi plot dell'immagine

```
# Figura 1 x 1: 1 figure e 1 axes
fig, ax = plt.subplots()

# Figura 2 x 3: 1 figure e 6 axes
# ax è un array: (es ax[0,2] è ultimo grafico della prima riga)
fig, ax = plt.subplots(2, 3)
```

Il maggior lavoro si farà con i metodi messi a disposizione da `ax/axes`; alcuni metodi utili sono riportati in tabella 12.1, mentre l'anatomia di un grafico matplotlib in figura 12.1.

- gli *axis* invece sono i veri e propri assi di ciascun plot/axes

12.2 Salvataggio figura

Per il salvataggio figura, in `fig.savefig`:

- specificare i `dpi` per i png
- se `transparent=True` la figura sarà transparent e meglio si adatta all'inclusione in documenti con sfondo di diverso colore

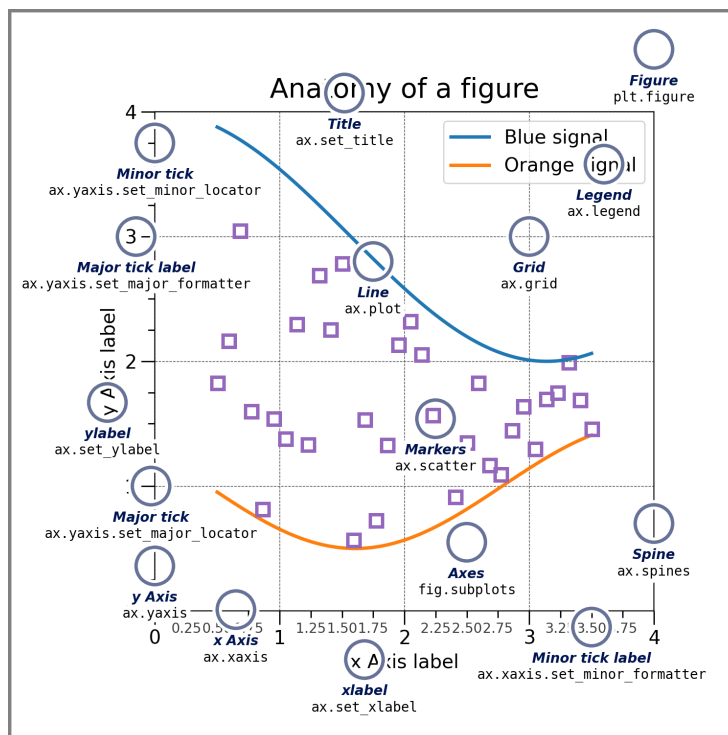


Figura 12.1: Anatomia di una figura

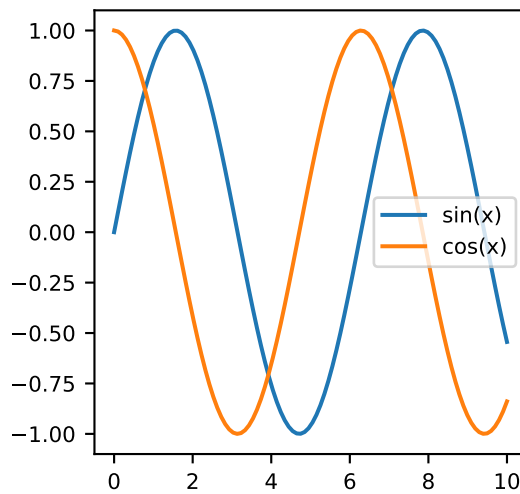


Figura 12.2: First plot

- se si passa `bbox_inches='tight'` si shrinka la fig alle dimensioni del grafico interno, eliminando spazio bianco attorno ad esso ad eccezione di un piccolo ammontare di padding (aggiustabile con `pad_inches`)

Nel seguito si userà `lb.io.export_figure` per comodità (è un wrapper attorno a `savefig`)

12.3 Impostazione layout figura ed esempi

12.3.1 Layout standard

Grafico singolo Esempio minimale figura 12.2:

- si aggiungono le funzioni plottate con successive chiamate usando lo stesso `ax`
- ciascuna funzione ha una `label` che verrà utilizzata nella creazione della `legend`

```
fig, ax = plt.subplots(figsize=(3,3))
x = np.linspace(0, 10, 100)
ax.plot(x, np.sin(x), label = 'sin(x)')
ax.plot(x, np.cos(x), label = 'cos(x)')
ax.legend()
lb.io.export_figure(fig, label = 'first_plot')
```

Vettore di grafici Un esempio con layout grafici separati, in figura 12.3

- aumentiamo la dimensione di `figsize`
- ci riferiamo agli `ax` come ad un vettore

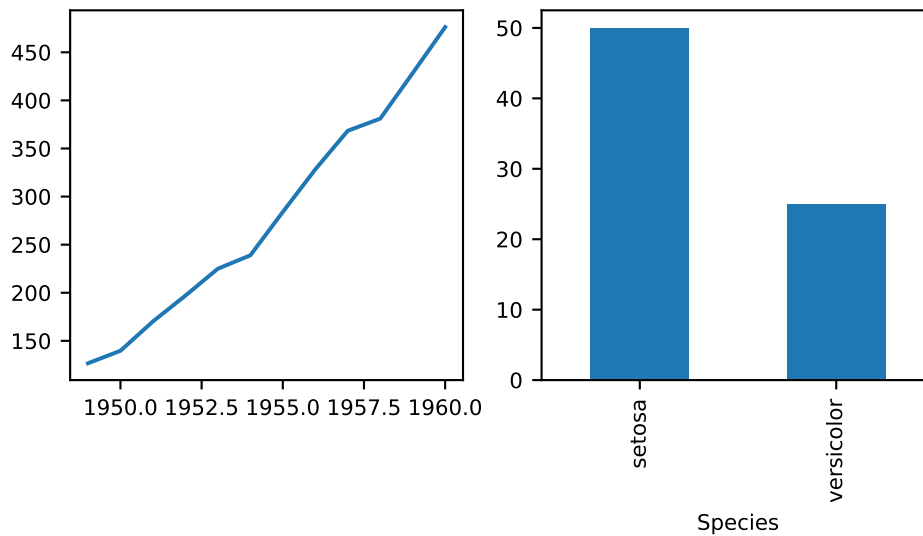


Figura 12.3: Second plot

- interfacciamo il codice matplotlib con quello di pandas i *wrapper di pandas* in due modi
 1. prendendo in maniera grezza in indici (sull'asse delle x) e valori (sull'asse delle y) dalla serie
 2. utilizzando il metodo della serie

```
fig2, ax2 = plt.subplots(nrows=1, ncols=2, figsize=(6, 2.5))

# raw matplotlib style using index and values
psng = rdatasets.data("AirPassengers").rename(columns={"value": "number"})
psng["year"] = np.floor(psng.time)
# how to select several columns: use [[]]
monthly_mean = psng.groupby("year")[["number"]].mean()
ax2[0].plot(monthly_mean["number"].index, monthly_mean["number"].values)
# otherwise quicker single column (single [])
# monthly_mean = psng.groupby("year")["number"].mean()
# ax2[0].plot(monthly_mean.index, monthly_mean.values)

# pandas methods using iris: frequency of first 75 flower types
iris = rdatasets.data("iris")
freqs = iris.iloc[:75, 5].value_counts()
freqs.plot.bar(ax = ax2[1])

lb.io.export_figure(fig2, label = 'second_plot')
```

Matrice di grafici Matrice 2×2 in cui plottiamo i subset di un grafico

- utilizziamo `constrained_layout` per aggiustare le dimensioni dei plot affinché fittino la figura (buona idea quando si hanno più plot in una singola figura).

Inoltre nel caso di molteplici subplots affiancati si può controllare lo spazio bianco verticale/orizzontale tra i plot si usa `subplots_adjust` con valori percentuale, ad esempio per togliere tutto lo

```
fig.subplots_adjust(wspace=0, hspace=0)
```

- usiamo `flatten` di `numpy` per iterare sequenzialmente sugli `ax`
- evitiamo che l'ultimo (inutile) axes venga mostrato impostando `axis` ad `off`
- impostiamo che i range di `x` e `y` siano comuni per
- aggiungiamo un titolo complessivo di figura mediante `fig.suptitle`

```
iris = rdatasets.data("iris")

fig3, ax3 = plt.subplots(nrows=2, ncols=2, figsize = (5,5), constrained_layout=True)
flat_axes = ax3.flatten()

for ax, species in zip(flat_axes, ["setosa", "versicolor", "virginica"]):
    subs = iris.query(f"Species=='{species}'") # create subset
    ax.scatter(x=subs["Sepal.Length"], y=subs["Sepal.Width"], alpha = 0.5)
    ax.set_xlim((3, 9))
    ax.set_ylim((1, 5))
    ax.set_title(f"{species}") # single graph title
    ax.set_xlabel("length (cm)")
    ax.set_ylabel("width (cm)")

ax3[1, 1].axis("off") # non visualizzare il quarto grafico
fig3.suptitle("Sepal length and width scatterplot", fontsize=15)
lb.io.export_figure(fig3, label = 'third_plot')
```

Il risultato è in figura 12.4

12.3.2 Layout custom

Griglia di axes di dimensione variabile Si usa `plt.GridSpec` e poi `add_subplot` fornendo la griglia (con slicing) per generare gli `ax` (figura 12.5)

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
fig = plt.figure(figsize=(5,5))

ax1 = fig.add_subplot(grid[0, 0])
ax1.boxplot([rng.standard_normal(50), rng.standard_normal(50) + 1])

ax2 = fig.add_subplot(grid[0, 1:])
ax2.bar(x = ["A", "B", "C", "D", "E", "F"],
        height = [10, 20, 40, 5, 25, 20])
```

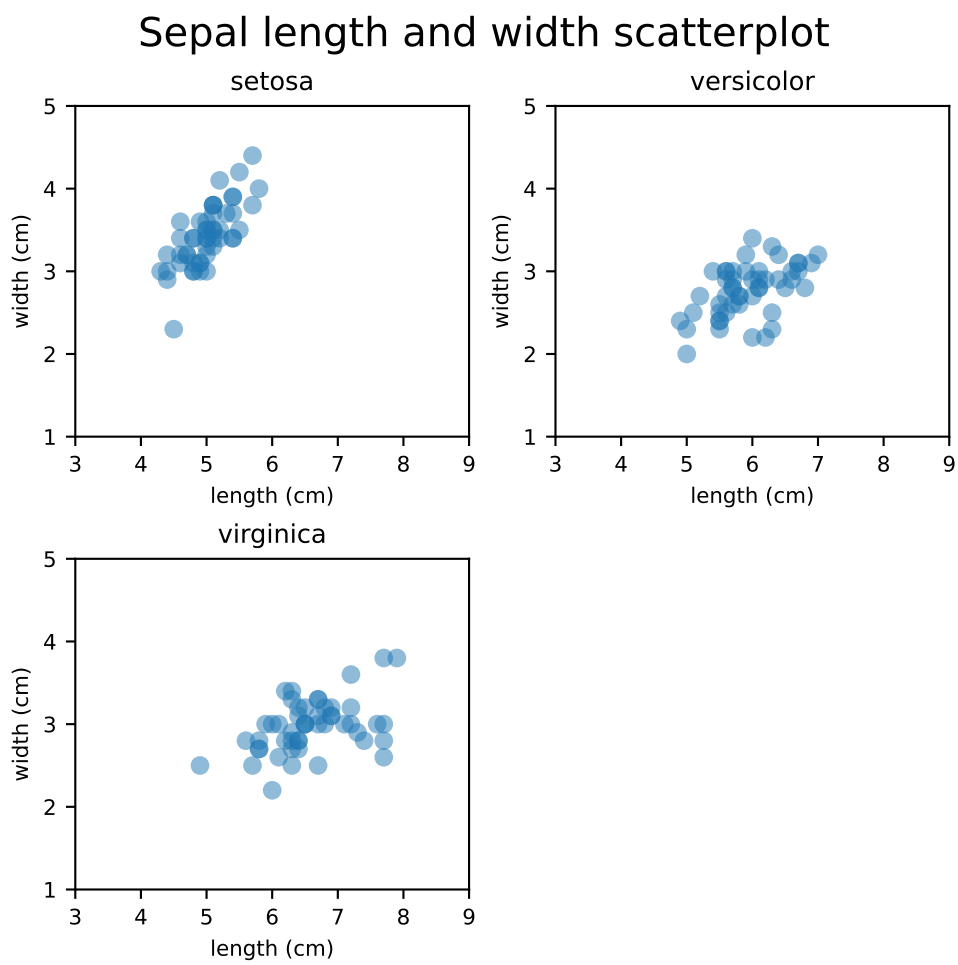


Figura 12.4: Third plot

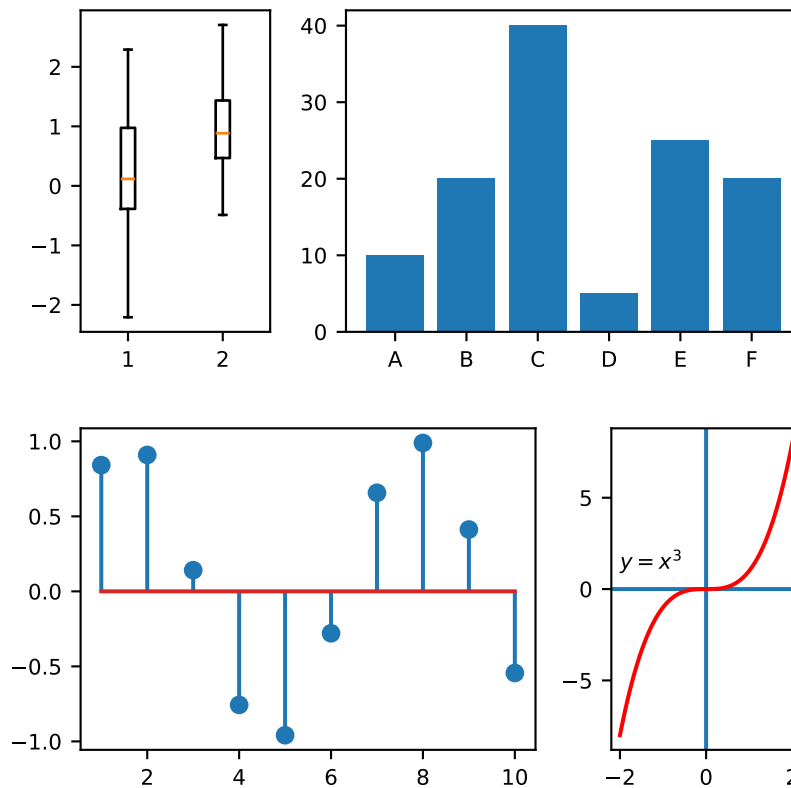


Figura 12.5: Custom grid

```
ax3 = fig.add_subplot(grid[1, :2])
x = np.arange(1, 11)
ax3.stem(x, np.sin(x))

ax4 = fig.add_subplot(grid[1, 2])
x = np.linspace(start=-2, stop=2, num=100)
ax4.axhline(0) # riferimento
ax4.axvline(0) # cartesiano
ax4.plot(x, x**3, color="red")
ax4.text(-2, 1, '$y = x^3$')

lb.io.export_figure(fig, label = 'custom_grid')
```

Axes uno dentro all'altro Il setup di sopra pone subplot affiancati e riempie tutta la figura; maggiore controllo sul setup dei subplot può essere ottenuto con `add_axes`. Questa prende come input una lista di quattro numeri che specificano le coordinate [`left`, `bottom`, `width`, `height`] nel range da 0 (in basso a sinistra della figura) a 1 in alto a destra (figura 12.6).

```
fig = plt.figure(figsize = (3,3))
```

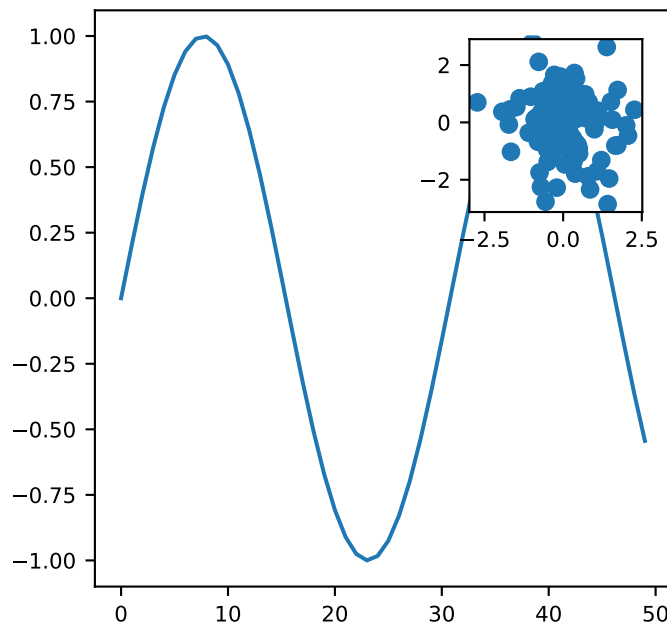



Figura 12.6: Custom subplots

```
ax1 = fig.add_axes([0, 0, 1, 1]) # assi standard
ax2 = fig.add_axes([0.65, 0.65, 0.3, 0.3]) # assino piccolino a 0.65, 0.65
                                           # della figura ed estensione 0.3x0.3

ax1.plot(np.sin(np.linspace(0, 10)))
ax2.scatter(rng.standard_normal(100), rng.standard_normal(100))
lb.io.export_figure(fig, label = 'custom_subplots')
```

12.4 Configurazioni

Ogni volta che matplotlib si carica legge runtime configuration (rc) che valgono per ciascun plot creato.

12.4.1 Ottenimento e modifica

È possibile:

- listare le configurazioni indagando il dict `plt.rcParams`

```
plt.rcParams # tutte le configurazioni
plt.rcParams["figure.figsize"] # dimensioni figura
```

- modificarle mediante la funzione `plt.rc`

```
# impostare le dimensioni delle figure a 8.5 cm (figsize prende una
# lista di 2 misure in pollici come input)
plt.rc("figure", figsize = [8.5/2.54] * 2)
```

- salvare le configurazioni in `.config/matplotlib/matplotlibrc`. Ad esempio per salvare l'impostazione come default inserire

```
figure(figsize:    3.346 , 3.346  # figure size in inches
```

12.4.2 Ripristino impostazioni default

È possibile ripristinare i valori di default con la funzione `plt.rcParamsDefaults()`.

12.4.3 Cambiare stile

Sono disponibili stili di grafico diversi che servono per avere un array di configurazioni già pronto:

- per listare gli stili disponibili

```
plt.style.available
```

- per impostare lo stile dei grafici per tutto il resto della sessione

```
plt.style.use('default')
```

- per impostare uno stile temporaneamente (es per una serie di grafici) si può usare un context manager:

```
with plt.style.context('stylename'):
    make_a_plot()
    make_another_plot()
```

12.5 Grafici utili

Nel seguito alcuni grafici utili fatti mediante matplotlib o seaborn (che è un wrapper di più alto livello di matplotlib).

Per il plotting si farà preferibilmente uso del metodo `.plot` delle `Series` di `pandas` (che risulta particolarmente integrato con `matplotlib`).

Da ricordare che

- `pandas` di default plotta *l'indice* della serie/dataframe (indice di riga) sull'asse x e il valore della serie sull'asse y
- A tal proposito, a volte può essere utile trasporre il dataframe di elaborazione per invertire nomi colonne ed indici di riga (accessore `pd.DataFrame.T`) prima di procedere effettivamente a `plot`

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pylabmisc as lb
import seaborn as sns
sns.set_theme()

# dataset
iris = sns.load_dataset('iris')
cd = lb.datasets.load("compact.csv")
political = lb.datasets.load("political.csv")
```

```

>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> import pylabmisc as lb
>>> import seaborn as sns
>>> sns.set_theme()

>>> # dataset
>>> iris = sns.load_dataset('iris')
>>> cd = lb.datasets.load("compact.csv")
>>> political = lb.datasets.load("political.csv")

```

12.5.1 Linee

In figura 12.7 la durata media dei cd di una collezione per anno di pubblicazione. Se ci fossero state altre variabili oltre a durata (a parte quella di grouping) sarebbe stato plottato anche il loro andamento

```

>>> # durata media cd per anno
>>> cd.head() # dataset
   durata  anno
0      143  1966
1      150  1967
2      241  1994
3      337  1991
4      246  1994

>>> cd.groupby("anno").mean().head() # media annua di durata
   durata
anno
1958    212.0
1960    142.0
1961    150.0
1964    162.5
1965    155.75

# plotting
fig, ax = plt.subplots()
cd.groupby("anno").mean().plot.line(ax = ax)
fig = ax.get_figure()
lb.io.export_figure(fig, label="pandas_lines", caption = 'Diagramma linee')

```

12.5.2 Diagramma a barre

Si usi `bar` oppure `barh` per le barre orizzontali

```

>>> political.head()
   party_id  poid
0      3-dem    1
1      3-dem    1
2      3-dem    1
3      3-dem    1

```

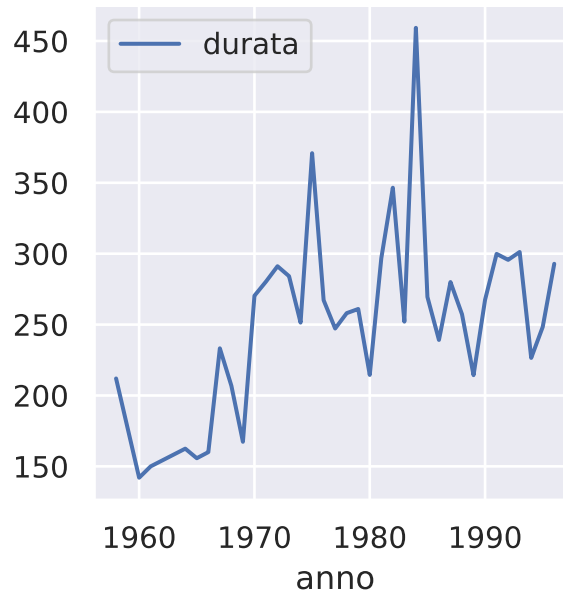


Figura 12.7: Diagramma linee

```

4      3-dem      1
>>> political.party_id.astype("category").value_counts(sort = False)
party_id
1-rep      74
2-ind     111
3-dem      91
Name: count, dtype: int64

```

Per il diagramma figura 12.8 come farlo con pandas

```

fig, ax = plt.subplots()
political.party_id.astype("category").value_counts(sort = False).plot.bar(ax=ax)
fig = ax.get_figure()
lb.io.export_figure(fig, label="pandas_barre", caption = 'Diagramma a barre')

```

12.5.3 Istogramma

Utilizzando i metodi di pandas in figura 12.9

```

fig, ax = plt.subplots()
ax = iris.sepal_length.plot.hist(density = True, bins = range(9), xlim = [0, 8])
iris.sepal_length.plot.density(ax = ax)
fig = ax.get_figure()
lb.io.export_figure(fig, label="pandas_histo", caption = 'Istogramma (pandas syntax)')

```

12.5.4 Scatterplot

Uno rapido con pandas in figura 12.10 con alpha shading

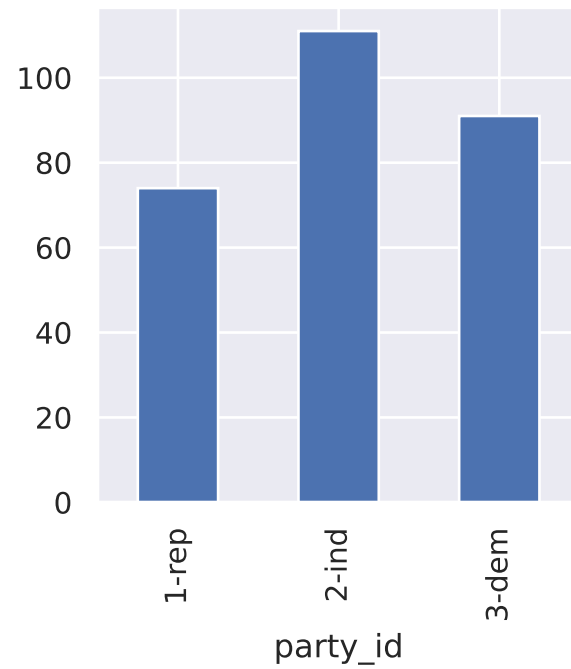


Figura 12.8: Diagramma a barre

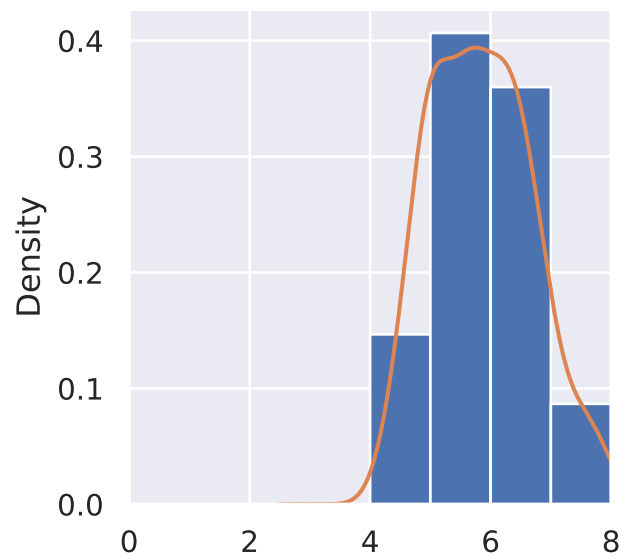


Figura 12.9: Istogramma (pandas syntax)

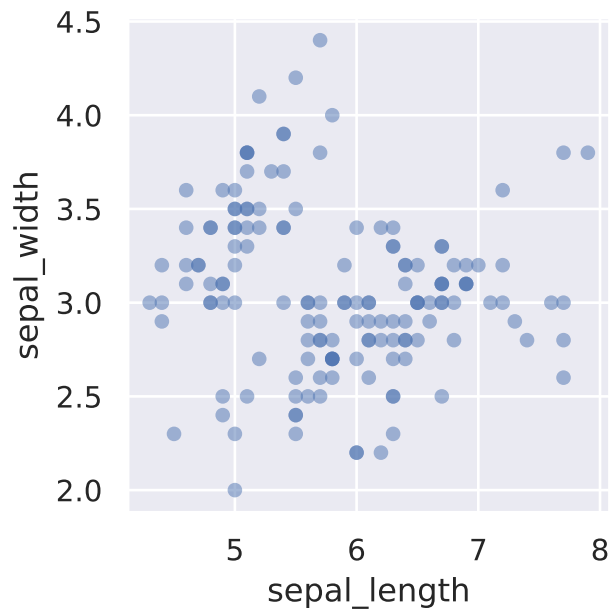


Figura 12.10: Scatterplot (pandas)

```
fix, ax = plt.subplots()
iris.plot.scatter(x = 'sepal_length', y = 'sepal_width', alpha = 0.5, ax=ax)
fig = ax.get_figure()
lb.io.export_figure(fig, label="pandas_scatter", caption = 'Scatterplot (pandas)')
# plt.close()
```

Uno più elaborato con colorazione condizionale, alpha shading e fatto con **seaborn** in figura 12.11

```
# data
group1 = pd.DataFrame({'x': np.random.normal(10, 1.2, 2000),
                       'y': np.random.normal(10, 1.2, 2000),
                       'group': np.repeat('A',2000) })
group2 = pd.DataFrame({'x': np.random.normal(14.5, 1.2, 2000),
                       'y': np.random.normal(14.5, 1.2, 2000),
                       'group': np.repeat('B',2000) })
group3 = pd.DataFrame({'x': np.random.normal(9.5, 1.5, 2000),
                       'y': np.random.normal(15.5, 1.5, 2000),
                       'group': np.repeat('C',2000) })
df = pd.concat([group1, group2, group3])

# Plot
fig, ax = plt.subplots()
ax = sns.scatterplot(x='x', y='y', data=df, hue='group', alpha = 0.30)
# fig = ax.get_figure()
lb.io.export_figure(fig, label="sns_scatter", caption = 'Scatterplot')
```

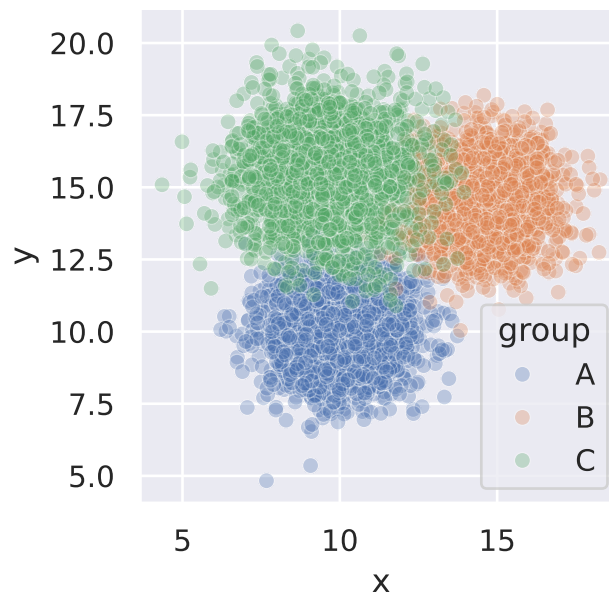


Figura 12.11: Scatterplot

12.5.5 Matrice di scatterplot

Invece per la matrice di scatterplot ne mettiamo senza con regressione (12.12 e con colorazioni 12.13.

```
# primo
fig, ax = plt.subplots()
plot = sns.pairplot(iris, kind="reg")
fig = plot.fig
lb.io.export_figure(fig, label="sns_pair1", caption = 'Pairplot 1', scale = 0.5)
# plt.close()

# secondo
fig, ax = plt.subplots()
plot = sns.pairplot(iris, kind="scatter", hue="species", markers=["o", "s", "D"], palette="Set2")
fig = plot.fig
lb.io.export_figure(fig, label="sns_pair2", caption = 'Pairplot 2', scale = 0.5)
# plt.close()
```

12.5.6 Boxplot

In figura 12.14

```
fig, ax = plt.subplots()
ax = iris.boxplot(by = "species", column="sepal_length")
fig = ax.get_figure()
lb.io.export_figure(fig, label="pandas_boxplot", caption = 'Boxplot')
```

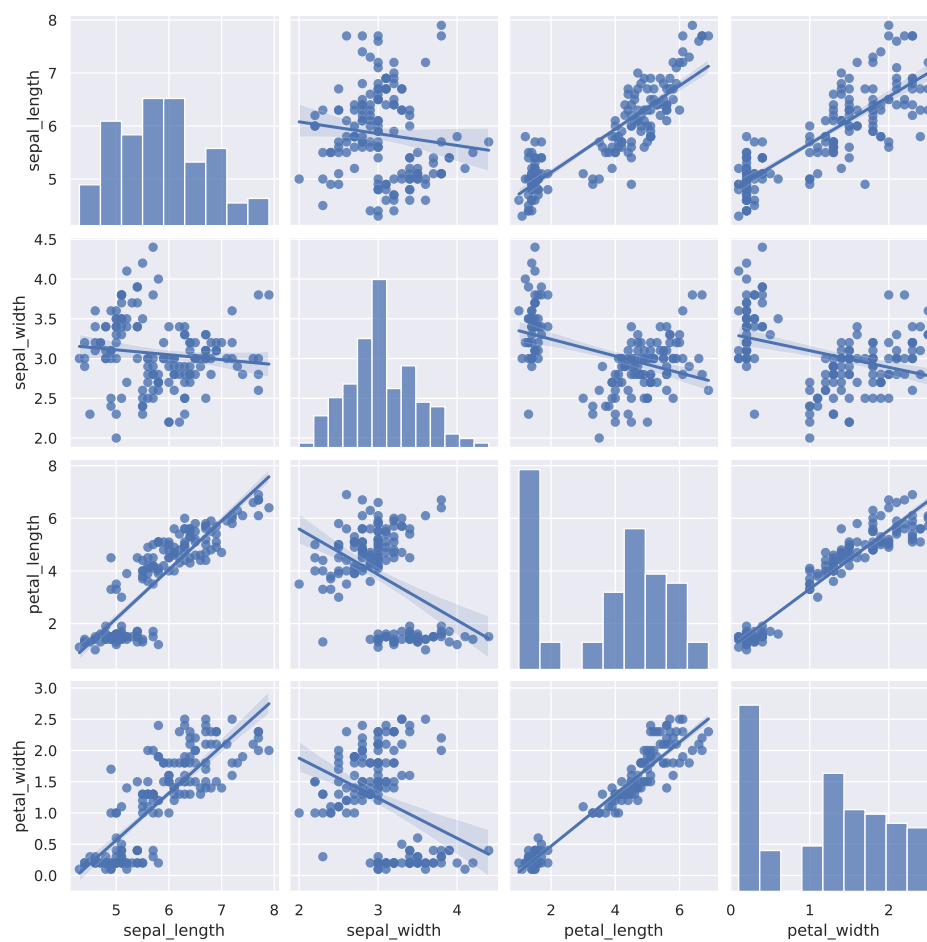


Figura 12.12: Pairplot 1

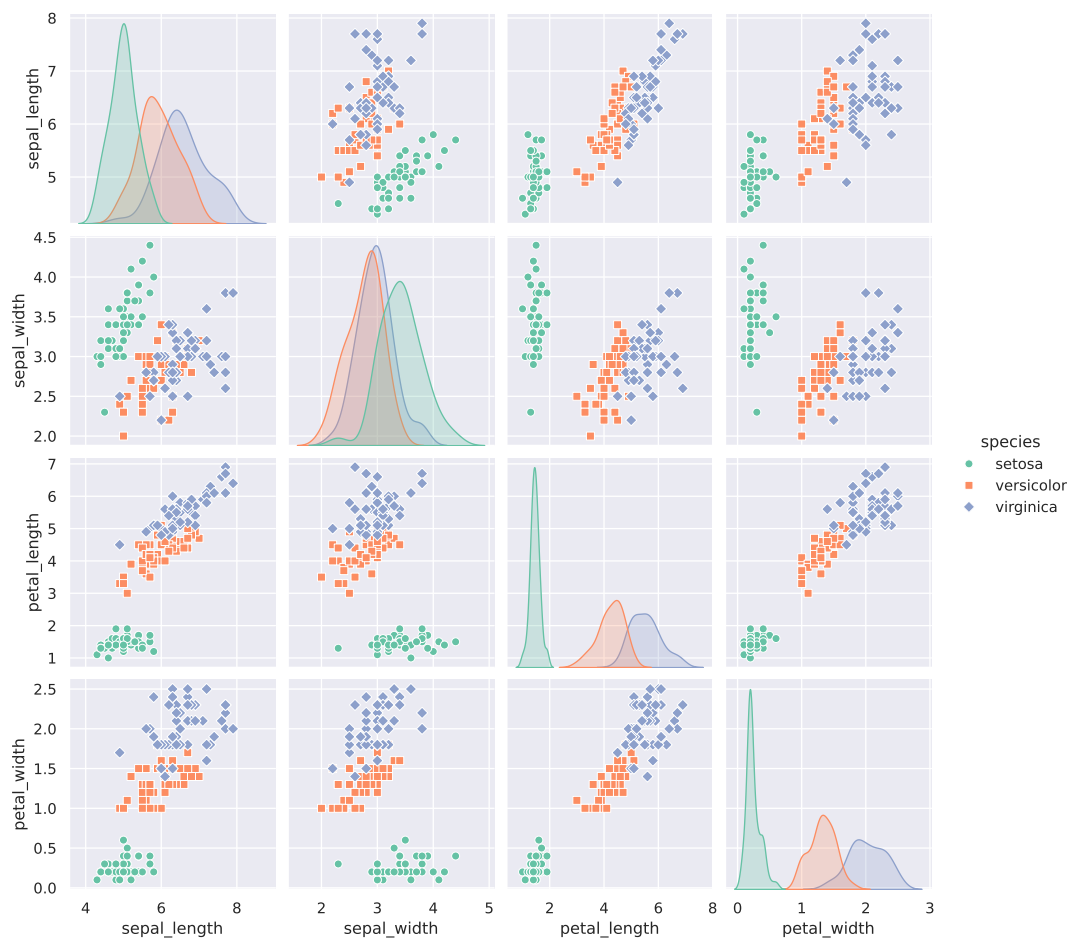


Figura 12.13: Pairplot 2

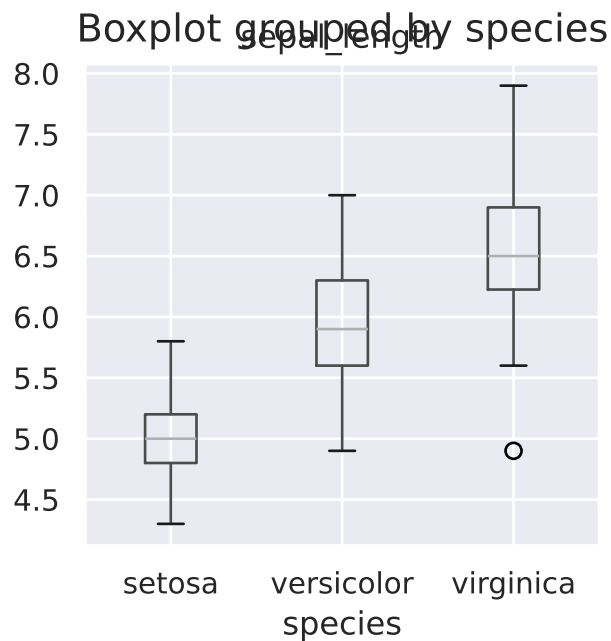


Figura 12.14: Boxplot

12.5.7 Correlation matrix

In figure 12.15

```
corr = iris.select_dtypes("float").corr()

fig, ax = plt.subplots()
ax = sns.heatmap(corr,
                  cmap="RdBu",
                  vmin = -1, vmax = 1,
                  annot = True, fmt=".2f",
                  xticklabels=corr.columns.values, yticklabels=corr.columns.values)
fig = ax.get_figure()
lb.io.export_figure(fig, label="corrmatrix", caption = 'Correlation matrix')

# fig, ax = plt.subplots()
# ax = plt.matshow()
# fig = ax.get_figure()
# lb.io.export_figure(fig, label="corrmatrix", caption = 'Correlation matrix')
```

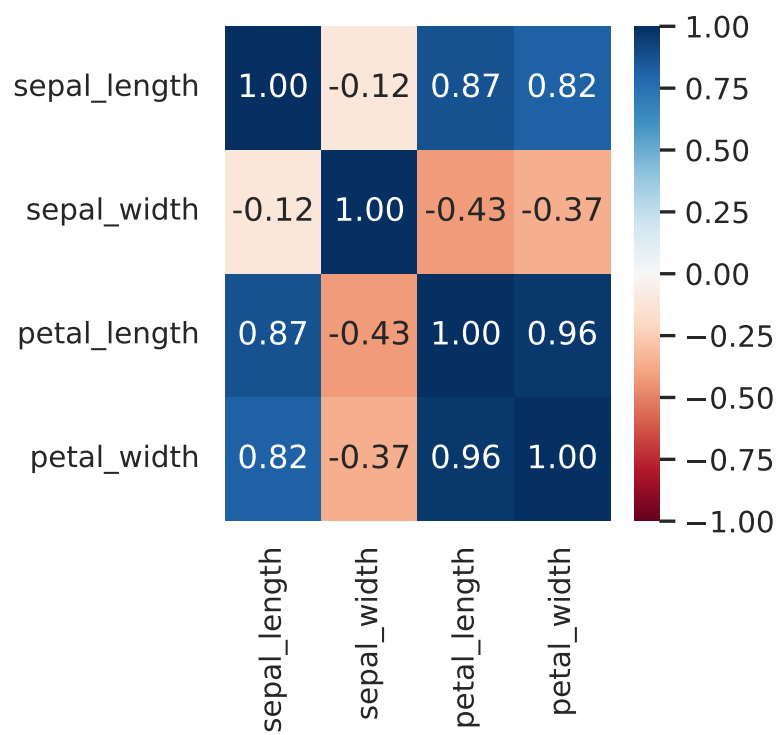


Figura 12.15: Correlation matrix

Parte III

Cookbook

Capitolo 13

Algebra lineare

Contents

13.1 Setup	231
13.2 Algebra lineare con numpy	231

13.1 Setup

Importiamo le librerie qui usate

```
import numpy as np
from scipy import stats
```

13.2 Algebra lineare con numpy

Per trasporre si usa il metodo `transpose` degli array o l'attributo `T`

```
X = np.arange(12).reshape((3, 4))
X.transpose()
X.T           # usa il metodo swapaxes che è più generale
```

Per il prodotto tra matrici si usa `np.dot` o con la chiocciola:

```
np.dot(X.T, X)
X.T @ X
# anche X.T.dot(X)
```

Altre funzioni utili sono riportate in tabella 13.1

Funzione	Descrizione
<code>np.diag</code>	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array
<code>np.trace</code>	Compute the sum of the diagonal elements
<code>np.linalg.det</code>	Compute the matrix determinant
<code>np.linalg.eig</code>	Compute the eigenvalues and eigenvectors of a square matrix
<code>np.linalg.inv</code>	Compute the inverse of a square matrix
<code>np.linalg.pinv</code>	Compute the Moore-Penrose pseudo-inverse inverse of a square matrix
<code>np.linalg.qr</code>	Compute the QR decomposition
<code>np.linalg.svd</code>	Compute the singular value decomposition (SVD)
<code>np.linalg.solve</code>	Solve the linear system $\mathbf{Ax} = \mathbf{b}$ for \mathbf{x} , where \mathbf{A} is a square matrix
<code>np.linalg.lstsq</code>	Compute the least-squares solution to $\mathbf{y} = \mathbf{Xb}$

Tabella 13.1: Funzioni per algebra lineare

Capitolo 14

Statistica descrittiva

Contents

14.1	Info generali	234
14.2	Univariate	234
14.2.1	Statistiche varie	234
14.3	Bivariate	235
14.3.1	Tabelle di contingenza	235
14.3.2	Covarianza e correlazione	236
14.3.3	Tabelle pivot	236
14.3.4	Tabella trial	237
14.3.5	Stratificate	238

```
>>> import numpy as np
>>> import pandas as pd
>>> import tableone
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'tableone'

>>> # dataset di prova
>>> rng = np.random.default_rng(123)
>>> df1 = pd.DataFrame(rng.random((1000, 5)), columns=["a", "b", "c", "d", "e"])
>>> df1[:,2] = np.nan
>>> df2 = pd.DataFrame({
...     'name': ['Luca', 'Silvio', 'Luisa', 'Andrea', 'Giovanni'],
...     'age' : [21, 22, 23, 24, 25],
...     'income' : np.arange(5),
...     'group' : list("aaabb")
... })
>>> df3 = pd.DataFrame({"x": ["a", "a", "a", "a", "a", "b", "b"],
...                     "y": ["1", "2", "2", "2", "2", "1", "2"],
...                     "g": ["trt", "ctrl", "trt", "ctrl", "trt", "ctrl", "trt"]})

>>> categ_df = pd.DataFrame({"x": ["a", "a", "a", "a", np.nan, "b", "b"],
...                          "y": ["1", "2", "1", "2", "2", "1", "2"]})
```

```
>>> group_df = pd.DataFrame({'key1' : list('xyyzzzwww'),
...                           'key2' : ['one', 'two'] * 5,
...                           'data1' : rng.random(10),
...                           'data2' : rng.random(10)},
...                           index = list("abcdefghil"))
>>> strata_df = pd.DataFrame({"x": np.random.randn(7),
...                           "y": np.random.randn(7),
...                           "z": np.random.randn(7),
...                           "g": ["trt", "ctrl", "trt", "ctrl", "trt", "ctrl", "trt"]})
```

14.1 Info generali

Usare `info` ed `head` sul `DataFrame` di interesse.

```
>>> df1.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 5 columns):
#   Column  Non-Null Count  Dtype
---  -
0    a      500 non-null      float64
1    b      500 non-null      float64
2    c      500 non-null      float64
3    d      500 non-null      float64
4    e      500 non-null      float64
dtypes: float64(5)
memory usage: 39.2 KB
>>> df1.head()
   a      b      c      d      e
0  NaN  NaN  NaN  NaN  NaN
1  0.812095  0.923345  0.276574  0.819755  0.889893
2  NaN  NaN  NaN  NaN  NaN
3  0.629940  0.927407  0.231908  0.799125  0.518165
4  NaN  NaN  NaN  NaN  NaN
```

14.2 Univariate

14.2.1 Statistiche varie

Per le statistiche descrittive utilizzando `pd.Series`/`pd.DataFrame` consultare le funzioni riportate in tabella 14.1.

```
>>> # categoriche (frequenze)
>>> categ_df.x.value_counts()
x
a    4
b    2
Name: count, dtype: int64
>>> categ_df.x.value_counts(dropna=False)
```

Metodo	Series	DataFrame	Descrizione
count	✓	✓	numero di dati non mancanti
value_counts	✓	✓	frequenze
describe	✓	✓	set di statistiche descrittive
sum, prod	✓	✓	somma/prodotto dei valori
cumsum, cumprod, cummin, cummax	✓	✓	cumulate varie
diff, pct_change	✓	✓	differenze prime e variazione percentuale
min, max	✓	✓	minimo e massimo
mean	✓	✓	media
median	✓	✓	mediana
quantile	✓	✓	quantili
var, std	✓	✓	varianza/stddev campionaria
skew, kurt	✓	✓	skewness/kurtosis campionarie
argmin, argmax	✓		posizione (intera) di minimo/massimo
idxmin, idxmax	✓	✓	indici di massimo o minimo

Tabella 14.1: Metodi per analisi descrittiva

```
x
a      4
b      2
NaN    1
Name: count, dtype: int64
```

```
>>> # quantitative (count = non missing)
>>> df1.describe().transpose()
      count      mean      std      min      25%      50%      75%      max
a  500.0  0.502773  0.283937  0.001069  0.277015  0.504669  0.751380  0.999170
b  500.0  0.493140  0.282222  0.001018  0.266441  0.492543  0.726648  0.992696
c  500.0  0.502857  0.284280  0.001569  0.262469  0.496649  0.744064  0.999853
d  500.0  0.514312  0.292133  0.002091  0.273060  0.515317  0.768407  0.998832
e  500.0  0.504114  0.282020  0.000802  0.267343  0.505314  0.743535  0.997637
```

14.3 Bivariate

14.3.1 Tabelle di contingenza

Sono una speciale tipo di tabella pivot dove si applica il conteggio degli elementi in ciascun gruppo. Vi è una funzione apposta, `pd.crosstab`

```
>>> pd.crosstab(categ_df.x, categ_df.y, margins=True) # aggiungi totali
y      1  2  All
x
a      2  2    4
b      1  1    2
All    3  3    6
>>> pd.crosstab(categ_df.x, categ_df.y, dropna=False, margins=True) # display na
y      1  2  All
x
a      2  2  4.0
```

```

b      1      1      2.0
NaN    0      1      NaN
All    3      4      7.0
>>> pd.crosstab(categ_df.x, categ_df.y, margins=True, normalize = 'columns') # % col
y      1      2      All
x
a  0.666667  0.666667  0.666667
b  0.333333  0.333333  0.333333

```

14.3.2 Covarianza e correlazione

I metodi `cov` e `corr` permettono di ottenere la covarianza e correlazione per gli elementi di un dataframe.

```

>>> df1.cov()
      a      b      c      d      e
a  0.080620 -0.002448 -0.004181  0.003793  0.000086
b -0.002448  0.079649 -0.001561  0.006011  0.003534
c -0.004181 -0.001561  0.080815 -0.002689  0.000998
d  0.003793  0.006011 -0.002689  0.085342  0.000171
e  0.000086  0.003534  0.000998  0.000171  0.079535
>>> df1.corr() # pearson
      a      b      c      d      e
a  1.000000 -0.030547 -0.051794  0.045725  0.001079
b -0.030547  1.000000 -0.019462  0.072907  0.044398
c -0.051794 -0.019462  1.000000 -0.032377  0.012445
d  0.045725  0.072907 -0.032377  1.000000  0.002081
e  0.001079  0.044398  0.012445  0.002081  1.000000
>>> df1.corr(method='spearman')
      a      b      c      d      e
a  1.000000 -0.029228 -0.046313  0.045479 -0.000649
b -0.029228  1.000000 -0.020557  0.072495  0.044796
c -0.046313 -0.020557  1.000000 -0.033100  0.010191
d  0.045479  0.072495 -0.033100  1.000000  0.000381
e -0.000649  0.044796  0.010191  0.000381  1.000000

```

14.3.3 Tabelle pivot

Sono tabelle bivariate con statistiche stratificate per gruppi formati da righe e colonne (per intenderci il `tapply` con doppio indice di R).

Possono essere prodotte con i metodi che si vedranno sotto velocemente col metodo `pivot_table` dei `DataFrame`, che come default usa `mean` come funzione di aggregazione.

```

>>> group_df
      key1 key2  data1  data2
a      x  one  0.593121  0.764104
b      y  two  0.353471  0.638191
c      y  one  0.336277  0.956624
d      z  two  0.399734  0.178105
e      z  one  0.915459  0.434077

```

```

f    z    two    0.822278    0.137480
g    w    one    0.480418    0.837667
h    w    two    0.929802    0.768947
i    w    one    0.950948    0.244235
l    w    two    0.863556    0.815336
>>> group_df.pivot_table(index = "key1",      # cosa porre in riga
...                        columns = "key2",    # cosa porre in colonna
...                        values = "data1",     # dati per sintesi
...                        aggfunc = "median",   # metodo di DataFrameGroupBy
...                        margins = True)       # aggiungi i totali
key2      one      two      All
key1
w      0.715683  0.896679  0.896679
x      0.593121      NaN  0.593121
y      0.336277  0.353471  0.344874
z      0.915459  0.611006  0.822278
All    0.593121  0.822278  0.707699

>>> # utilizzo di funzioni custom: dovranno prendere in input una
>>> # serie e ritornare uno scalare

>>> def custom_fun(s): # s è una serie
...     return s.min() # qua la custom è inutile ma le cose si fanno così
...
>>> group_df.pivot_table(index = "key1",      # cosa porre in riga
...                        columns = "key2",    # cosa porre in colonna
...                        values = "data1",     # dati per sintesi
...                        aggfunc = custom_fun)
key2      one      two
key1
w      0.480418  0.863556
x      0.593121      NaN
y      0.336277  0.353471
z      0.915459  0.399734

```

14.3.4 Tabella trial

Utilizzare la libreria `tableone`.

```

tab1_df = tableone.load_dataset('pn2012')
tab1_df.info()
tab1_df.head()
ft = {0: "alive", 1: "dead"}
tab1_df["group"] = pd.Categorical(tab1_df.death.map(ft))
select = ['Age', 'SysABP', 'Height', 'Weight', 'ICU', 'group']
categ = ['ICU', 'group']
groupby = 'group'
nonnormal = ['Age']
labels={'death': 'mortality'}

```

Metodo	Descrizione
<code>any, all</code>	Vero se qualcuno o tutti sono veri (nel senso di Python)
<code>count</code>	numero di non NA
<code>cummin, cummax</code>	minimo e massimo cumulato
<code>cumsum</code>	somma cumulata
<code>cumprod</code>	produttoria cumulata
<code>first, last</code>	primo, ultimo
<code>mean</code>	media
<code>median</code>	mediana
<code>min, max</code>	minimo, massimo
<code>nth</code>	n-esimo elemento con dati ordinati
<code>prod</code>	prodotto
<code>quantile</code>	quantile
<code>rank</code>	ranghi
<code>size</code>	dimensione del gruppo
<code>sum</code>	somma
<code>std, var</code>	deviazione standard e varianza campionaria

Tabella 14.2: Metodi ottimizzati grouped

```

tab1 = tableone.TableOne(tab1_df,
                        columns=select,
                        categorical=categ,
                        groupby=groupby,
                        nonnormal=nonnormal,
                        rename=labels,
                        pval=False)

```

```
tab1
```

14.3.5 Stratificate

Si applica il classico split-apply-combine ossia:

1. si splitta la struttura dati (**Series** o **DataFrame**) mediante il metodo **groupby**: a questo si può passare alternativamente liste, array, dict, series o funzioni per effettuare il mapping ai gruppi.
Di default **groupby** agisce splittando per sulle righe (**axis** = "index"), ma si può specificare **axis** = "columns" se si vuole splittare per colonna.
Il metodo **groupby** ritorna un oggetto **GroupBy**:
 - è un iterabile sui "chunk" di dati (quindi funziona bene con i **for**) e fornisce un set di metodi già pronti da applicare
 - può essere soggetto a *indexing*, selezionando i subset di dati su cui performare le operazioni
2. si effettua l'elaborazione sui chunk di dati, alternativamente
 - l'oggetto **GroupBy** ha un set di metodi builtin già disponibili da applicare a tutti i chunk di dati. I metodi disponibili derivano da quelli

della classe di riferimento quindi se è una `SeriesGroupBy` i metodi principali sono ereditati da quelli delle `Series`, viceversa se un `DataFrameGroupBy` dai `DataFrame`. In tabella 14.2 sono riportati alcuni metodi notevoli per `DataFrame`.

- si possono applicare definire funzioni custom
 - si possono effettuare analisi ancor più custom (es una funzione a tutto il dataset).
3. viene riassembleto il tutto: il tipo di oggetto ritornato dipende dalle elaborazioni eseguite.

14.3.5.1 Splitting

Vediamo vari metodi di splitting:

- basato su valori di variabili

```
>>> group_df
   key1 key2  data1  data2
a    x  one  0.593121  0.764104
b    y  two  0.353471  0.638191
c    y  one  0.336277  0.956624
d    z  two  0.399734  0.178105
e    z  one  0.915459  0.434077
f    z  two  0.822278  0.137480
g    w  one  0.480418  0.837667
h    w  two  0.929802  0.768947
i    w  one  0.950948  0.244235
l    w  two  0.863556  0.815336

>>> # Splitting di DataFrame sulla base di una variabile
>>> df_spl = group_df.groupby('key2')
>>> df_spl.mean(numeric_only = True) # se no essendoci key2 da errore
      data1      data2
key2
one    0.655244  0.647341
two    0.673768  0.507612

>>> # Splitting di DataFrame sulla base di due variabili
>>> df_spl = group_df.groupby(['key1', 'key2'])
>>> df_spl.mean()
      data1      data2
key1 key2
w     one   0.715683  0.540951
      two   0.896679  0.792142
x     one   0.593121  0.764104
y     one   0.336277  0.956624
      two   0.353471  0.638191
z     one   0.915459  0.434077
      two   0.611006  0.157792
```

- sulla base degli indici di riga

```

>>> # recode mediante dict
>>> groups = {"a" : 'g1', "b" : 'g1',
...           "c" : 'g1', "d" : 'g1',
...           "e" : 'g2', "f" : 'g2',
...           "g" : 'g2', "h" : 'g2',
...           "i" : 'g2', "l" : 'g2'}
>>> df_spl = group_df.groupby(groups)
>>> df_spl.count()
      key1  key2  data1  data2
g1        4    4      4      4
g2        6    6      6      6

>>> # Splitting con Series (esempio logiche)
>>> groups = (group_df.key1 == "w") & (group_df.key2 == "one")
>>> df_spl = group_df.groupby(groups)
>>> df_spl.count()
      key1  key2  data1  data2
False     8    8      8      8
True      2    2      2      2

>>> # Splitting con funzione/predicato: viene applicata al valore degli indici
>>> df_spl = group_df.groupby(lambda x: x in ("b", "c", "d"))
>>> df_spl.count()
      key1  key2  data1  data2
False     7    7      7      7
True      3    3      3      3

>>> # anche grouping sulla base di indici è possibile
>>> # modifichiamo un attimo il dataframe per mostrare
>>> # la funzionalità. Si usa il progressivo/nome dell'indice
>>> # specificato in level

>>> group_df2 = group_df.set_index('key2')
>>> group_df2.groupby(level = 0).count()
      key1  data1  data2
key2
one        5      5      5
two        5      5      5
>>> group_df2.groupby(level = 'key2').count()
      key1  data1  data2
key2
one        5      5      5
two        5      5      5

```

14.3.5.2 Convertire indici di grouping in variabili

Negli esempi precedenti i risultati vengono restituiti con un indice, eventualmente gerarchico, composto a partire dalle chiavi di raggruppamento. Non essendo ciò sempre desiderabile, se si desidera avere qualcosa di più classico dove gli indici vengono invece messi in opportune colonne, a `groupby` si fornisce il parametro `as_index = False`.


```

>>> group_df
  key1 key2  data1  data2
a    x  one  0.593121  0.764104
b    y  two  0.353471  0.638191
c    y  one  0.336277  0.956624
d    z  two  0.399734  0.178105
e    z  one  0.915459  0.434077
f    z  two  0.822278  0.137480
g    w  one  0.480418  0.837667
h    w  two  0.929802  0.768947
i    w  one  0.950948  0.244235
l    w  two  0.863556  0.815336

>>> # aggregazione "flat"
>>> keys = ['key1', 'key2']
>>> df_spl = group_df.groupby(keys, as_index = False)
>>> df_spl.mean(numeric_only = True)
  key1 key2  data1  data2
0    w  one  0.715683  0.540951
1    w  two  0.896679  0.792142
2    x  one  0.593121  0.764104
3    y  one  0.336277  0.956624
4    y  two  0.353471  0.638191
5    z  one  0.915459  0.434077
6    z  two  0.611006  0.157792

```

14.3.5.3 Memorizzare in un dict lo split

Essendo il `GroupBy` un iterabile, è possibile elaborarlo, ad esempio trasformandolo in un `dict`, per avervi facile accesso:

```

>>> pieces = dict(list(group_df.groupby('key2')))
>>> pieces['two']
  key1 key2  data1  data2
b    y  two  0.353471  0.638191
d    z  two  0.399734  0.178105
f    z  two  0.822278  0.137480
h    w  two  0.929802  0.768947
l    w  two  0.863556  0.815336

```

14.3.5.4 Iterazione sui gruppi

Un oggetto `GroupBy` è iterabile quindi funziona bene con i `for`: questo genera una sequenza di tuple di due elementi, contenenti nome del gruppo (chiave) e chunk di dati di dati il pezzo di dati

```

>>> # chiave singola
>>> for group, data in group_df.groupby('key2'):
...     print(f"group = {group}")
...     print(data, "\n") # simple DataFrame
...
group = one

```

	key1	key2	data1	data2
a	x	one	0.593121	0.764104
c	y	one	0.336277	0.956624
e	z	one	0.915459	0.434077
g	w	one	0.480418	0.837667
i	w	one	0.950948	0.244235

	key1	key2	data1	data2
b	y	two	0.353471	0.638191
d	z	two	0.399734	0.178105
f	z	two	0.822278	0.137480
h	w	two	0.929802	0.768947
l	w	two	0.863556	0.815336

```
>>> # chiave multipla: il primo elemento della tupla è una
>>> # tupla con le chiavi
>>> for (key1, key2), data in group_df.groupby(['key1', 'key2']):
...     print(f"key1 = {key1}, key2 = {key2}")
...     print(data, "\n")
... 
```

	key1	key2	data1	data2
g	w	one	0.480418	0.837667
i	w	one	0.950948	0.244235

	key1	key2	data1	data2
h	w	two	0.929802	0.768947
l	w	two	0.863556	0.815336

	key1	key2	data1	data2
a	x	one	0.593121	0.764104

	key1	key2	data1	data2
c	y	one	0.336277	0.956624

	key1	key2	data1	data2
b	y	two	0.353471	0.638191

	key1	key2	data1	data2
e	z	one	0.915459	0.434077

	key1	key2	data1	data2
d	z	two	0.399734	0.178105
f	z	two	0.822278	0.137480

14.3.5.5 Scelta delle variabili di analisi

Se il dataset è molto grande e si analizzano poche colonne per sottogruppi, conviene fare lo splitting solamente di queste ultime; alternativamente è possibile splittare tutto il dataset e procedere ad analisi dei subset in un secondo momento (senza dover rifare lo splitting).

Per farlo, gli oggetti `GroupBy` accettano l'indexing:

```
>>> df_spl = group_df.groupby('key2') # grouping/splitting

>>> df_spl['data1'] # column selection as usual
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7fb4a18c3af0>

>>> sel = ['data1', 'data2'] # due colonne
>>> df_spl[sel]
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fb4a18a7110>
>>> df_spl[sel].mean() # elaborazione
      data1    data2
key2
one    0.655244  0.647341
two    0.673768  0.507612
```

14.3.5.6 Applicare funzioni di aggregazione custom

È possibile applicare funzioni che sintetizzano un insieme di dati in un unico valore passandole al metodo `aggregate` (o la sua abbreviazione `agg`) di un oggetto `GroupBy`:

```
>>> def range(x):
...     return x.max() - x.min()
...
>>> sel = ["data1", "data2"]
>>> group_df.groupby('key2')[sel].aggregate(range)
      data1    data2
key2
one    0.614671  0.712389
two    0.576331  0.677856
```

14.3.5.7 Elaborazioni custom

Si potrebbe voler aggregare usando:

- diverse funzioni (es `media` e `stdev`) per ciascuna colonna: nel caso usare una lista di funzioni e/o stringhe con nomi di metodi scelti

```
>>> df_spl = df2.groupby('group')
>>> sel = ["income", "age"]

>>> def range(x):
...     return x.max() - x.min()
...
>>> # lista
>>> analyses = ['mean', range]
```

```
>>> df_spl[sel].agg(analyses)
      income      age
      mean range mean range
group
a         1.0      2  22.0      2
b         3.5      1  24.5      1

>>> # lista di tuple con nome colonna e funzione applicata
>>> analyses = [('Media', 'mean'), ('Range', range)]
>>> df_spl[sel].agg(analyses)
      income      age
      Media Range Media Range
group
a         1.0      2  22.0      2
b         3.5      1  24.5      1
```

- diverse funzioni su colonne differenti (es media per la prima colonna, stdev per la seconda).

```
>>> # dict con variabili analizzate e analisi effettuate
>>> analyses = {'age' : ["max", "std"], 'income' : "std"}
>>> df_spl.agg(analyses)
      age      income
      max      std      std
group
a      23  1.000000  1.000000
b      25  0.707107  0.707107
```

14.3.5.8 Trasformazioni group-wise

Le aggregazioni effettuate mediante `aggregate` collassano un insieme di dati ad un numero. Viceversa:

- `transform` applica una funzione ad una serie: se l'output è un singolo valore questo viene broadcastato a tutti i membri del gruppo; se è un vettore (della stessa dimensione del gruppo) viene restituito. Eventuali parametri vanno specificati post funzione separati da virgole. Ad esempio lo scarto del gruppo dalla media di gruppo:

```
>>> def center(x):
...     return x - x.mean()
...
>>> def group_mean(x):
...     return x.mean()
...
>>> df_spl = group_df.groupby("key2")
>>> sel = ["data1", "data2"]
>>> df_spl[sel].transform(center)
      data1      data2
a -0.062124  0.116763
b -0.320297  0.130579
c -0.318968  0.309283
```

```

d -0.274034 -0.329507
e  0.260215 -0.213265
f  0.148509 -0.370132
g -0.174826  0.190325
h  0.256034  0.261336
i  0.295703 -0.403107
l  0.189788  0.307724
>>> df_spl[sel].transform(group_mean)
      data1      data2
a  0.655244  0.647341
b  0.673768  0.507612
c  0.655244  0.647341
d  0.673768  0.507612
e  0.655244  0.647341
f  0.673768  0.507612
g  0.655244  0.647341
h  0.673768  0.507612
i  0.655244  0.647341
l  0.673768  0.507612

```

- `apply` applica la funzione passata ai `DataFrame` creati dallo splitting; la funzione passata gli pu restituire uno scalare, una list/array o anche un dataframe.

Eventuali parametri della funzione specificati in seguito e separati da virgola

```

>>> def report(df, var = 'data1', method = 'high', n = 2):
...     if method == 'high':
...         sel = df[var].nlargest(n).index
...     elif method == 'low':
...         sel = df[var].nsmallest(n).index
...     return df.loc[sel, ]
...
>>> group_df
      key1 key2      data1      data2
a      x  one  0.593121  0.764104
b      y  two  0.353471  0.638191
c      y  one  0.336277  0.956624
d      z  two  0.399734  0.178105
e      z  one  0.915459  0.434077
f      z  two  0.822278  0.137480
g      w  one  0.480418  0.837667
h      w  two  0.929802  0.768947
i      w  one  0.950948  0.244235
l      w  two  0.863556  0.815336

>>> # applicata a tutto il dataframe: estrae i due valori più alti di
>>> # data1 overall
>>> report(group_df)
      key1 key2      data1      data2

```

```
i    w    one  0.950948  0.244235
h    w    two  0.929802  0.768947
```

```
>>> # stratified by key2: estrae i due valori più alti di data1 entro
```

```
>>> # gruppi di key2
```

```
>>> df_spl = group_df.groupby('key2')
```

```
>>> df_spl.apply(report, include_groups = False)
```

```
      key1      data1      data2
key2
one  i      w  0.950948  0.244235
     e      z  0.915459  0.434077
two  h      w  0.929802  0.768947
     l      w  0.863556  0.815336
```

```
>>> # uso di parametri della funzione: estrazione per data2
```

```
>>> df_spl.apply(report, var = 'data2', include_groups = False)
```

```
      key1      data1      data2
key2
one  c      y  0.336277  0.956624
     g      w  0.480418  0.837667
two  l      w  0.863556  0.815336
     h      w  0.929802  0.768947
```

Capitolo 15

Probabilità e simulazione

Contents

15.1 Combinatoria	247
15.2 Generazione di numeri casuali con <code>numpy.random</code> .	248
15.3 Variabili casuali in <code>scipy.stats</code>	249
15.3.1 Funzioni e parametri principali	249
15.3.2 Uso interattivo rapido	250
15.3.3 Freezing di una distribuzione	250
15.3.4 Uso del generatore di <code>numpy</code>	251
15.4 Altri argomenti	251
15.4.1 Bootstrap CI	251

Importiamo le librerie qui usate

```
>>> import itertools
>>> import math
>>> import numpy as np
>>> import pandas as pd
>>> import scipy
>>> from scipy import stats
```

15.1 Combinatoria

Fattoriale

```
>>> math.factorial(4)
24
```

Coefficiente binomiale

```
>>> n = 4
>>> k = 2
>>> scipy.special.comb(n, k)
np.float64(6.0)
```

Permutazioni di un vettore

```
>>> data = [1,2,3]
>>> list(itertools.permutations(data))
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

Combinazioni di elementi di un vettore

```
>>> list(itertools.combinations(data, 2))
[(1, 2), (1, 3), (2, 3)]
```

Prodotto cartesiano

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> list(itertools.product(a,b)) # eventualmente posto in np.array
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]
```

15.2 Generazione di numeri casuali con `numpy.random`

Si crea innanzitutto un generatore di numeri casuali (impostando seme e tipologia, volendo) dopodiché si utilizza questo per generare dati. Alternativamente si può

- utilizzando in generatore di default; il tutto è semplificato mediante `default_rng`

```
>>> rng = np.random.default_rng(seed = 6315744)
>>> rv = rng.standard_normal(5) # random vector
>>> rv
array([ 0.02015758,  2.22358201,  0.94127564, -0.75983371, -2.37769907])
>>> rm = rng.standard_normal((2, 3)) # random matrix
>>> rm
array([[ 0.25145986,  0.10362785, -1.26726019],
       [ 0.46240554, -0.14667618, -0.6394399 ]])
```

- specificare il generatore utilizzato: occorre creare un generatore, alimentarlo con un bitgenerator (eg default o Mersenne-Twister) a cui si è fornito il seed impostato: ad esempio per un generatore standard e Mersenne-Twister (utilizzato da R) fatti a mano

```
>>> from numpy.random import Generator, PCG64, MT19937
>>> seed = 2154
>>> std = Generator(PCG64(seed = seed)) # equivalente al default
>>> std.standard_normal(5)
array([ 2.65496506, -0.20262369, -0.85302693, -0.37646063, -0.30786317])
>>> mt = Generator(MT19937(seed = seed))
>>> mt.standard_normal(5)
array([ 0.73327039,  1.19398753, -0.66427529, -0.42821724,  0.7421379 ])
```


Metodo	Descrizione
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in place
<code>uniform</code>	Draw samples from a uniform distribution
<code>integers</code>	Draw random integers from a given low-to-high range
<code>binomial</code>	Draw samples a binomial distribution
<code>standard_normal</code>	Estrazioni da una normale 0, 1
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1) distribution

Tabella 15.1: Metodi per la generazione di numeri casuali in `numpy`

Famiglia	Funzione	Famiglia	Funzione
Bernoulli	<code>bernoulli</code>	Uniforme cont.	<code>uniform</code>
Binomiale	<code>binom</code>	Esponenziale	<code>expon</code>
Geometrica	<code>geom</code>	Normale	<code>norm</code>
Binomiale neg.	<code>nbinom</code>	Gamma	<code>gamma</code>
Ipergeometrica	<code>hypergeom</code>	Chi-quadrato	??
Poisson	<code>poisson</code>	Beta	<code>beta</code>
Uniforme disc.	<code>randint</code>	T di Student	<code>t</code>
		F	??
		Logistica	??
		Lognormale	??
		Weibull	??
		Pareto	??

Tabella 15.2: Funzioni `scipy.stats.*` per variabili casuali in Python

Generazione di un seme casuale Per generare in maniera casuale un seme, la prima volta si può utilizzare `secrets.randbits`

```
>>> import secrets
>>> secrets.randbits(128)
30861334410814909315069050256516047118
```

il numero varia di volta in volta quindi necessario copiarlo/incollarlo la prima.

Metodi per generatori casuali I metodi di interesse sono in tabella 15.1; allo stato attuale le distribuzioni non sono molte ma si potrà utilizzare questo generatore con `scipy.stats`.

15.3 Variabili casuali in `scipy.stats`

15.3.1 Funzioni e parametri principali

In `scipy.stats` le variabili casuali di maggiore interesse sono riportate in tabella 15.2, i relativi metodi sono riportati in tabella 15.3.

Metodo	Descrizione
rvs	generazione di numeri casuali; equivalente di r* di R
pdf, pmf	probability density e mass function; equivalente di d* di R
cdf	cumulative distribution function; equivalente di p* di R
ppf	percent point function (inversa di cdf); equivalente di q* di R
sf	Survival Function (1-CDF)
isf	Inverse Survival Function (Inverse of SF)
stats	media, varianza, (Fisher's) skew, or (Fisher's) kurtosis
moment	non-central moments of the distribution

Tabella 15.3: Metodi variabili casuali **scipy**

I parametri di maggior interesse sono **loc** (locazione) e **scale** (dispersione); alcune quantitative anche un parametro **shape**. Nel caso della normale **loc** è la media ed è impostata di default a 0, **scale** la deviazione standard ed è impostata a 1.

15.3.2 Uso interattivo rapido

Un esempio di utilizzo interattivo rapido con normale a seguire. Per facilitare, le funzioni supportano il broadcasting

```
>>> stats.norm.rvs(size = 5) # estrazioni di casuali (come rnorm(5))
array([-1.23720195,  1.35402239, -0.41170338,  0.02320987,  0.22394222])
>>> stats.norm.pdf(0.5)      # densità: come dnorm(0.5)
np.float64(0.35206532676429947)
>>> stats.norm.cdf(1.96)     # cdf: come pnorm(1.96)
np.float64(0.9750021048517795)
>>> stats.norm.ppf(0.5)      # quantile: come qnorm(0.5)
np.float64(0.0)

>>> # esempio con broadcasting
>>> stats.norm.pdf([0.025, 0.5, 0.975])
array([0.39881763, 0.35206533, 0.24801872])
```

15.3.3 Freezing di una distribuzione

Spesso capita di dover lavorare più volte con distribuzioni dai parametri differenti. Onde evitare di dover reinserire i parametri in ogni chiamata possiamo effettuare il freezing di una distribuzione

```
>>> n01 = stats.norm(loc = 0, scale = 1) # mu=0, sd=1 sono i valori di default
>>> n01.rvs(size = 5)
array([-0.65604941, -2.58053666, -0.47319104, -0.53861024,  0.24196367])

>>> n25 = stats.norm(loc = 2, scale = 5) # mu=2, sd=5
>>> n25.rvs(size = 5)
array([0.92272942, 7.80571504, 3.41693524, 0.36969143, 2.13931505])
```

15.3.4 Uso del generatore di numpy

Passiamo il generatore di numeri casuali in numpy nel metodo della funzione `freezata`

```
>>> rng = np.random.default_rng(302132)

>>> n01 = stats.norm() # esempio d'uso:
>>> n01.rvs(size = 5, random_state = rng) # usato al momento
array([-0.14191574,  0.42922204, -0.33735776,  0.20712603, -0.9709426 ])
>>> n01.rvs(size = 5, random_state = rng) # dell'estrazione
array([ 0.64705971, -1.05564341,  0.94984921, -0.30577826, -0.61788831])

>>> rng2 = np.random.default_rng(302132) # riproduzione sequenza
>>> n01.rvs(size = 5, random_state = rng2)
array([-0.14191574,  0.42922204, -0.33735776,  0.20712603, -0.9709426 ])
>>> n01.rvs(size = 5, random_state = rng2)
array([ 0.64705971, -1.05564341,  0.94984921, -0.30577826, -0.61788831])
```

15.4 Altri argomenti

15.4.1 Bootstrap CI

```
>>> from scipy.stats import bootstrap

>>> rng = np.random.default_rng(6235)
>>> data_gen = stats.norm()
>>> data = data_gen.rvs(100, random_state = rng)

>>> # 95 bca CI of median
>>> res = bootstrap(data = [data], # ocio qui: "sequence of array-like"
...                 statistic = np.median,
...                 n_resamples = 10000,
...                 random_state = rng)

>>> res.confidence_interval # ci
ConfidenceInterval(low=np.float64(-0.21318771294834601), high=np.float64(0.17332161369498447))
>>> res.confidence_interval.low # ci
np.float64(-0.21318771294834601)
>>> res.confidence_interval.high # ci
np.float64(0.17332161369498447)
>>> res.bootstrap_distribution # valori delle stats nei resample
array([-0.08377,  0.09051194,  0.08447688, ..., -0.17939695,
        -0.13573812,  0.16842878], shape=(10000,))
```


Capitolo 16

Test statistici

Contents

16.1 Setup	254
16.2 Medie	254
16.2.1 Test t: 1 gruppo vs valore teorico	254
16.2.2 Test t: 2 gruppi indipendenti	254
16.2.3 Anova (2+ gruppi indipendenti)	254
16.2.4 Test t: 2 gruppi appaiati	256
16.2.5 Anova per misure ripetute (2+ gruppi appaiati)	256
16.3 Non parametric	257
16.3.1 Wilcoxon	257
16.3.2 Mann Whitney	257
16.3.3 Kruskal Wallis	257
16.3.4 Friedman test	258
16.4 Proporzioni	258
16.4.1 Test binomiale e CI clopper pearson	258
16.4.2 Test di Fisher	258
16.4.3 Chisquare	259
16.4.4 McNemar	259
16.4.5 Q di Cochran	259
16.5 Tassi	260
16.5.1 Comparazione 2 tassi	260
16.6 Correlazione	260
16.6.1 Pearson	260
16.6.2 Spearman	261
16.6.3 Tests	261
16.7 Varianze	261
16.7.1 Test di Bartlett	261
16.7.2 Test di Levene	261
16.7.3 Test di Fligner	261
16.8 Sopravvivenza	261
16.8.1 Logrank test	262
16.9 Agreement	262

16.9.1 Cohen's K	262
16.9.2 Fleiss K	262
16.9.3 Lin coefficient	262
16.10 Reliability/consistency	263
16.10.1 Cronbach α	263
16.10.2 ICC	263
16.11 Multiplicity	263
16.12 Test simulativi	263

16.1 Setup

Importiamo le librerie qui usate

```
>>> import numpy as np
>>> import pandas as pd
>>> import pingouin as pg
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'pingouin'
>>> from scipy import stats
```

16.2 Medie

16.2.1 Test t: 1 gruppo vs valore teorico

```
>>> x = [5.5, 2.4, 6.8, 9.6, 4.2]
>>> stats.ttest_1samp(x, popmean = 4)
TtestResult(statistic=np.float64(1.3973913920955365), pvalue=np.float64(0.23482367964...))
>>> pg.ttest(x, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.2.2 Test t: 2 gruppi indipendenti

```
>>> np.random.seed(123)
>>> trt = np.random.normal(size=18)
>>> ctrl = np.random.normal(size=22)
>>> stats.ttest_ind(trt, ctrl, equal_var = False)
TtestResult(statistic=np.float64(0.62859959726889), pvalue=np.float64(0.5339329415063...))
>>> pg.ttest(trt, ctrl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.2.3 Anova (2+ gruppi indipendenti)

```
>>> df = pg.read_dataset('anova')
Traceback (most recent call last):
```

```

File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> df.head()
   x  y
0  a  1
1  b  2
2  c  3
>>> df["Hair color"].value_counts()
Traceback (most recent call last):
  File "/home/l/.local/lib/python3.13/site-packages/pandas/core/indexes/base.py", line 3805, in
    return self._engine.get_loc(casted_key)
         ~~~~~
  File "index.pyx", line 167, in pandas._libs.index.IndexEngine.get_loc
  File "index.pyx", line 196, in pandas._libs.index.IndexEngine.get_loc
  File "pandas/_libs/hashtable_class_helper.pxi", line 7081, in pandas._libs.hashtable.PyObject
  File "pandas/_libs/hashtable_class_helper.pxi", line 7089, in pandas._libs.hashtable.PyObject
KeyError: 'Hair color'

```

The above exception was the direct cause of the following exception:

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/l/.local/lib/python3.13/site-packages/pandas/core/frame.py", line 4102, in __geti
    indexer = self.columns.get_loc(key)
  File "/home/l/.local/lib/python3.13/site-packages/pandas/core/indexes/base.py", line 3812, in
    raise KeyError(key) from err
KeyError: 'Hair color'
>>> # oneway classica
>>> pg.anova(dv='Pain threshold', between='Hair color', data=df, detailed = True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> chunks = [data["Pain threshold"].values
...           for color, data in df.groupby("Hair color")]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "/home/l/.local/lib/python3.13/site-packages/pandas/core/frame.py", line 9183, in groupb
    return DataFrameGroupBy(
        obj=self,
        ...<7 lines>...
        dropna=dropna,
    )
  File "/home/l/.local/lib/python3.13/site-packages/pandas/core/groupby/groupby.py", line 1329,
    grouper, exclusions, obj = get_grouper(
        ~~~~~
        obj,
        ~~~~
    ...<5 lines>...
        dropna=self.dropna,
        ~~~~~
    )

```

```

~
File "/home/l/.local/lib/python3.13/site-packages/pandas/core/groupby/grouper.py",
    raise KeyError(gpr)
KeyError: 'Hair color'
>>> stats.f_oneway(*chunks)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'chunks' is not defined
>>> # non assumendo numerosità comuni e/o varianza costante
>>> pg.welch_anova(dv='Pain threshold', between='Hair color', data=df)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?

```

16.2.4 Test t: 2 gruppi appaiati

```

>>> pre = [5.5, 2.4, np.nan, 9.6, 4.2]
>>> post = [6.4, 3.4, 6.4, 11., 4.8]
>>> stats.ttest_rel(pre, post, nan_policy="omit")
TtestResult(statistic=np.float64(-5.901869285972221), pvalue=np.float64(0.00971277159...))
>>> pg.ttest(pre, post, paired=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?

```

16.2.5 Anova per misure ripetute (2+ gruppi appaiati)

```

>>> # dataset in formato long
>>> df = pg.read_dataset('rm_anova')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> df.head()
   x  y
0  a  1
1  b  2
2  c  3
>>> pg.rm_anova(dv='DesireToKill', within='Disgustingness',
...             subject='Subject', data=df, detailed=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> # dataset in formato wide
>>> df = pg.read_dataset('rm_anova_wide')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> df.head()
   x  y
0  a  1
1  b  2

```



```
2 c 3
>>> pg.rm_anova(df)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.3 Non parametric

16.3.1 Wilcoxon

```
>>> pre = np.array([20, 22, 19, 20, 22, 18, 24, 20, 19, 24, 26, 13])
>>> post = np.array([38, 37, 33, 29, 14, 12, 20, 22, 17, 25, 26, 16])
>>> stats.wilcoxon(pre, post)
WilcoxonResult(statistic=np.float64(20.5), pvalue=np.float64(0.2880859375))
>>> pg.wilcoxon(pre, post, correction = False)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> pg.wilcoxon(pre, post) # con correzione di continuità
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.3.2 Mann Whitney

```
>>> trt = np.random.uniform(low=0, high=1, size=20)
>>> ctrl = np.random.uniform(low=0.2, high=1.2, size=20)
>>> stats.mannwhitneyu(trt, ctrl, use_continuity=True)
MannwhitneyuResult(statistic=np.float64(149.0), pvalue=np.float64(0.17192970543827346))
>>> pg.mwu(trt, ctrl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

TODO:
[scipy.stats.brunnermunzel](#)

16.3.3 Kruskal Wallis

```
>>> df = pg.read_dataset('anova')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> # pingouin
>>> pg.kruskal(data=df, dv='Pain threshold', between='Hair color')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> # scipy
>>> stats.kruskal(*chunks)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'chunks' is not defined
```

16.3.4 Friedman test

Tipo un wilcoxon con più colonne di 2

```
>>> # dati da friedman.test in R
>>> df = pd.DataFrame(np.array([5.40, 5.50, 5.55,
...                             5.85, 5.70, 5.75,
...                             5.20, 5.60, 5.50,
...                             5.55, 5.50, 5.40,
...                             5.90, 5.85, 5.70,
...                             5.45, 5.55, 5.60,
...                             5.40, 5.40, 5.35,
...                             5.45, 5.50, 5.35,
...                             5.25, 5.15, 5.00,
...                             5.85, 5.80, 5.70,
...                             5.25, 5.20, 5.10,
...                             5.65, 5.55, 5.45,
...                             5.60, 5.35, 5.45,
...                             5.05, 5.00, 4.95,
...                             5.50, 5.50, 5.40,
...                             5.45, 5.55, 5.50,
...                             5.55, 5.55, 5.35,
...                             5.45, 5.50, 5.55,
...                             5.50, 5.45, 5.25,
...                             5.65, 5.60, 5.40,
...                             5.70, 5.65, 5.55,
...                             6.30, 6.30, 6.25])).reshape(22,3),
...                  columns = ["t0", "t1", "t2"])
>>> stats.friedmanchisquare(df.t0, df.t1, df.t2)
FriedmanchisquareResult(statistic=np.float64(11.142857142857132), pvalue=np.float64(0))
>>> pg.friedman(df)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.4 Proporzioni

16.4.1 Test binomiale e CI clopper pearson

```
>>> test = stats.binomtest(3, n=15, p=0.1) #p è la probabilità sotto h0 da rifiutare
>>> test
BinomTestResult(k=3, n=15, alternative='two-sided', statistic=0.2, pvalue=0.184061069)
>>> test.proportion_ci()
ConfidenceInterval(low=0.04331200510583602, high=0.48089113380685317)
```

16.4.2 Test di Fisher

Si ha per le tabelle 2x2

```
>>> # odds ratio (stima) calcolato è diverso da quello di R (vedi doc), p-uguale
>>> tea = np.array([[3, 1], [1, 3]])
>>> stats.fisher_exact(tea)
```

```
SignificanceResult(statistic=np.float64(9.0), pvalue=np.float64(0.48571428571428565))
```

TODO:
stats.barnard_exact

16.4.3 Chisquare

Per le tabelle $n \times m$

```
>>> obs = np.array([[10, 10, 20],
...                 [20, 20, 20]])
>>> stats.chi2_contingency(obs)
Chi2ContingencyResult(statistic=np.float64(2.7777777777777777), pvalue=np.float64(0.24935220877
[18., 18., 24.])))
>>> data = pg.read_dataset('chi2_independence')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> pg.chi2_independence(data, x='sex', y='target')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.4.4 McNemar

```
>>> data = pg.read_dataset('chi2_mcnemar')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> pg.chi2_mcnemar(data, 'treatment_X', 'treatment_Y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
```

16.4.5 Q di Cochran

Mc nemar per più tempi/trattamenti su stessi soggetti

```
>>> df = pg.read_dataset('cochran')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?
>>> df.head()
   t0  t1  t2
0  5.40 5.50 5.55
1  5.85 5.70 5.75
2  5.20 5.60 5.50
3  5.55 5.50 5.40
4  5.90 5.85 5.70
>>> df_wide = df.pivot_table(index="Subject", columns="Time", values="Energetic")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/l/.local/lib/python3.13/site-packages/pandas/core/frame.py", line 9509, in pivot_
    return pivot_table(
```

```

        self,
        ...<9 lines>...
        sort=sort,
    )
File "/home/l/.local/lib/python3.13/site-packages/pandas/core/reshape/pivot.py", line 111, in _internal_pivot_table
    table = __internal_pivot_table(
        data,
        ...<9 lines>...
        sort,
    )
File "/home/l/.local/lib/python3.13/site-packages/pandas/core/reshape/pivot.py", line 111, in __internal_pivot_table
    raise KeyError(i)
KeyError: 'Energetic'
>>> pg.cochran(df_wide)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined. Did you mean: 'p'?

```

16.5 Tassi

16.5.1 Comparazione 2 tassi

Il test di poisson di python verifica che la differenza tra tassi sia nulla (quello di R che il rapporto sia unitario)

```

>>> # poisson.test(c(11, 6+8+7), c(800, 1083+1050+878))
>>> stats.poisson_means_test(11, 800, 6+8+7, 1083+1050+878)
SignificanceResult(statistic=np.float64(1.5342150126346437), pvalue=np.float64(0.1386))
>>> # i risultati sono diversi ma il manuale di python dice

```

I risultati di questo test sono differenti da quelli di R ma la documentazione di python dice che ha maggior potenza del test poissoniano esatto di R.

16.6 Correlazione

```

>>> # generare dati
>>> mean, cov = [4, 6], [(1, .5), (.5, 1)]
>>> x, y = np.random.multivariate_normal(mean, cov, 30).T
>>> data = {"x": x, "y": y}
>>> df = pd.DataFrame(data)

```

16.6.1 Pearson

```

>>> stats.pearsonr(df.x, df.y)
PearsonRResult(statistic=np.float64(0.42350936826041014), pvalue=np.float64(0.0196978))
>>> pg.corr(df.x, df.y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined

```

16.6.2 Spearman

```
>>> stats.spearmanr(df.x, df.y)
SignificanceResult(statistic=np.float64(0.35973303670745277), pvalue=np.float64(0.0508737485072))
>>> pg.corr(df.x, df.y, method="spearman")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined
```

16.6.3 Tests

```
>>> pg.rcorr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined
```

16.7 Varianze

Vediamo le funzioni per la comparazione di k varianze sotto diverse ipotesi sempre meno restrittive

16.7.1 Test di Bartlett

Testa parametricamente la differenza di varianze ipotizzando una distribuzione normale del carattere nella popolazione. Se a 2 gruppi è il test F.

```
>>> a = [8.88, 9.12, 9.04, 8.98, 9.00, 9.08, 9.01, 8.85, 9.06, 8.99]
>>> b = [8.88, 8.95, 9.29, 9.44, 9.15, 9.58, 8.36, 9.18, 8.67, 9.05]
>>> c = [8.95, 9.12, 8.95, 8.85, 9.03, 8.84, 9.07, 8.98, 8.86, 8.98]
>>> stats.bartlett(a, b, c)
BartlettResult(statistic=np.float64(22.789434813726768), pvalue=np.float64(1.1254782518834628e-05))
```

16.7.2 Test di Levene

Testa parametricamente la differenza di varianze non ipotizzando distribuzioni normali

```
>>> stats.levene(a, b, c)
LeveneResult(statistic=np.float64(7.584952754501659), pvalue=np.float64(0.002431505967249681))
```

16.7.3 Test di Fligner

Equivalente non parametrico

```
>>> stats.fligner(a, b, c)
FlignerResult(statistic=np.float64(10.803687663522238), pvalue=np.float64(0.00450826080004775))
```

16.8 Sopravvivenza

Utilizziamo la libreria `lifelines`

16.8.1 Logrank test

```
>>> T1 = [1, 4, 10, 12, 12, 3, 5.4]
>>> E1 = [1, 0, 1, 0, 1, 1, 1]
>>> T2 = [4, 5, 7, 11, 14, 20, 8, 8]
>>> E2 = [1, 1, 1, 1, 1, 1, 1, 1]

>>> from lifelines.statistics import logrank_test
>>> results = logrank_test(T1, T2, event_observed_A=E1, event_observed_B=E2)
>>> results.print_summary()
<lifelines.StatisticalResult: logrank_test>
      t_0 = -1
  null_distribution = chi squared
degrees_of_freedom = 1
      test_name = logrank_test

---
test_statistic    p  -log2(p)
      0.09 0.77      0.38
>>> # results.p_value, results.test_statistic
```

16.9 Agreement

16.9.1 Cohen's K

```
>>> from statsmodels.stats.inter_rater import cohens_kappa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'statsmodels'
>>> cohens_kappa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cohens_kappa' is not defined
```

16.9.2 Fleiss K

```
>>> from statsmodels.stats.inter_rater import fleiss_kappa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'statsmodels'
>>> fleiss_kappa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fleiss_kappa' is not defined
```

16.9.3 Lin coefficient

```
>>> from statsmodels.stats.inter_rater import fleiss_kappa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'statsmodels'
```

```
>>> fleiss_kappa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'fleiss_kappa' is not defined
```

16.10 Reliability/consistency

16.10.1 Cronbach α

```
>>> pg.cronbach_alpha
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined
```

16.10.2 ICC

```
>>> pg.intraclass_corr
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pg' is not defined
```

16.11 Multiplicity

```
scipy.stats.tukey_hsd
scikit_posthocs.posthoc_dunn
```

```
statsmodels.stats.multitest.multipletests
scipy.stats.false_discovery_control
```

```
pg.multicomp
pg.pairwise_gameshowell
pg.pairwise_tukey
pg.pairwise_tests
pg.pairwise_corr
pg.ptests
```

16.12 Test simulativi

Importiamo le librerie qui usate

```
>>> from scipy import stats

>>> stats.bootstrap
<function bootstrap at 0x7fb4a266b6a0>
>>> stats.permutation_test
<function permutation_test at 0x7fb4a26947c0>
>>> stats.monte_carlo_test
```

```
<function monte_carlo_test at 0x7fb4a266b920>
```


Capitolo 17

Integrazione con R

Contents

17.1 Interscambio dataset	265
17.1.1 Da R a Python	265
17.1.2 Da Python a R	266
17.2 Chiamare R da Python: rpy2	266
17.2.1 Importazione di pacchetti	267
17.2.2 Ottenimento di dati con rpy2	267
17.2.3 Valutare stringhe di R	268
17.2.4 Creazione di vettori	268
17.2.5 Conversione DataFrame a R	268
17.2.6 Utilizzo di funzioni	268

17.1 Interscambio dataset

17.1.1 Da R a Python

In generale, per esportare un dataset di R verso Python, importare il dataset in R e usare `lbmisc::pddf`.

Alternativamente, per importare direttamente un dataset R noto in Python si possono importare usare i pacchetti `rdatasets` o `statsmodels`, entrambi usano

```
>>> import statsmodels.api as sm
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'statsmodels'
>>> airquality = sm.datasets.get_rdataset("airquality", "datasets")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sm' is not defined
>>> df = airquality.data
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'airquality' is not defined
```

```

>>> df
      x      y
0  5.318795  8.358522
1  4.412340  6.286310
2  5.528292  5.922329
3  3.257367  4.568498
4  2.157919  4.965227
5  4.815540  5.729637
6  5.240592  6.028069
7  2.110286  4.274727
8  3.637569  4.370217
9  3.254288  6.432150
10 3.774571  6.805685
11 5.620999  6.257528
12 3.861018  5.481841
13 4.432908  4.455020
14 2.084009  6.682313
15 4.004256  6.038398
16 4.775493  4.913518
17 4.433651  4.828241
18 3.748947  6.991816
19 4.386092  6.887341
20 2.983572  5.262313
21 5.353514  7.021039
22 3.315623  3.290577
23 4.514892  5.962784
24 3.521322  6.269538
25 2.741949  4.471717
26 3.046622  5.546362
27 3.871165  5.307241
28 5.413727  6.313684
29 4.494251  6.815937
>>> print(airquality.__doc__)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'airquality' is not defined

```

17.1.2 Da Python a R

Utilizzare `pylbmisc.io.rdf`.

17.2 Chiamare R da Python: rpy2

Si installa

```
pip install --user rpy2
```

Quattro moduli principali:

- `rpy2.robj`: high-level interface. basato su `rpy2.rinterface`

- `rpy2.interactive`: high-level interface basata su `rpy2.robjects` per l'uso interattivo
- `rpy2.rlike`: funzioni e dati in python che mimano funzionalità di R
- `rpy2.rinterface`: interfaccia di basso livello

Il modulo che ci interessa di più è `robjects`, effettuiamo le seguenti importazioni:

```
>>> import rpy2.robjects as ro                # tutto il resto ..

>>> from rpy2.robjects import pandas2ri      # traduzione data.frame
>>> from rpy2.robjects.packages import importr # importazione pacchetti
>>> from rpy2.robjects.packages import data   # importazione dati
```

17.2.1 Importazione di pacchetti

```
>>> base = importr('base')
>>> utils = importr('utils')
>>> stats = importr('stats')
>>> lme4 = importr("lme4")
```

17.2.2 Ottenimento di dati con rpy2

Per ottenere `lme4::sleepstudy`

```
>>> sleepstudy = data(lme4).fetch('sleepstudy')['sleepstudy']
>>> print(utils.head(sleepstudy))
  Reaction Days Subject
1    249.6    0    308
2    258.7    1    308
3    250.8    2    308
4    321.4    3    308
5    356.9    4    308
6    414.7    5    308
```

È ancora in formato R, per avere un `DataFrame` pandas se lo vogliamo analizzare in python tocca fare sta roba

```
>>> with (ro.default_converter + pandas2ri.converter).context():
...     df = ro.conversion.get_conversion().rpy2py(sleepstudy)
...
>>> df.head()
  Reaction  Days Subject
1  249.5600  0.0    308
2  258.7047  1.0    308
3  250.8006  2.0    308
4  321.4398  3.0    308
5  356.8519  4.0    308
```

17.2.3 Valutare stringhe di R

rpy2 esegue R, per accedere al namespace si usa `rpy2.robjects.r`.

```
>>> pi = ro.r['pi']
>>> pi[0]
3.141592653589793

>>> # possiamo scrivere anche codice più complesso
>>> res = ro.r("""
... set.seed(123)
... f <- function(x) mean(rnorm(x))
... f(10)
... """)
>>> res[0]
0.0746256440971619
```

17.2.4 Creazione di vettori

```
>>> ints = ro.IntVector([2, 1, 3])
>>> print(ints)
[1] 2 1 3

>>> floats = ro.FloatVector([1.1, 2.2, 3.3])
```

17.2.5 Conversione DataFrame a R

Se vogliamo applicare funzioni di R questo è il modo di procedere; si usa `rpy2.robjects.pandas2ri` con sintassi speculare a quanto già visto (si usa `py2rpy` invece di `rpy2py`):

```
>>> import pandas as pd
>>> df = pd.DataFrame({'int1': [1,2,3], 'int2': [4, 5, 6]})

>>> with (ro.default_converter + pandas2ri.converter).context():
...     r_df = ro.conversion.get_conversion().py2rpy(df)
...
```

17.2.6 Utilizzo di funzioni

```
>>> # applicazione di una funzione
>>> res1 = base.sort(ints)
>>> print(res1)
[1] 1 2 3

>>> # funzione con parametri
>>> res2 = base.sort(floats, decreasing=True)
>>> print(res2)
[1] 3.3 2.2 1.1

>>> # utilizzare funzioni r su un dataframe R (ottenuto sopra)
```

```
>>> print(base.summary(r_df))
      int1      int2
Min.   :1.0  Min.   :4.0
1st Qu.:1.5  1st Qu.:4.5
Median :2.0  Median :5.0
Mean   :2.0  Mean   :5.0
3rd Qu.:2.5  3rd Qu.:5.5
Max.   :3.0  Max.   :6.0

>>> # svolgimento di una analisi
>>> lm1 = stats.lm('Reaction ~ Days', data = sleepstudy)
>>> # print(base.summary(lm1)) #verbose output
>>> fm1 = lme4.lmer('Reaction ~ Days + (Days | Subject)', data = sleepstudy)
>>> # print(base.summary(fm1)) #verbose output
```


Capitolo 18

Misc cookbook

Contents

18.1 Validazione DataFrame con pandera	271
18.2 Logging	273
18.3 SymPy	273
18.4 Ottenere codice di oggetti	274
18.5 Esecuzione parallela	274
18.6 File di configurazione	275
18.7 Calendario	276
18.8 Tcl/Tk	277
18.9 Telegram	277

18.1 Validazione DataFrame con pandera

`pandera` permette di definire descrizioni del dataset che vogliamo avere e validare quello che effettivamente abbiamo. Vi sono due api attualmente, una classica e una più simile a `dataclass`/`pydantic`. Usiamo questa seconda.

```
>>> import pandera as pa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'pandera'
```

Iniziamo con un dataset di esempio dove è necessario fare preprocessing e validare

```
>>> # Uno schema più evoluto su dati più real-life
>>> df = pd.DataFrame({
...     "idx" : [1,2,3,4,5,6],
...     "adate": ["2020-01-02", "2021-01-01", "2022-01-02"] * 2,
...     "state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
...     "ohio" : [1, 1, 1, 0, 0, 0],
...     "year": [str(y) for y in [2000, 2001, 2002, 2001, 2002, 2003]],
...     "pop": [str(p) for p in [1.5, 1.7, 3.6, np.nan, 2.9, 3.2]]
... })
```

Definiamo una classe che descrive le caratteristiche

```
>>> # meglio usare i tipi di dato builtin
>>> class Model(pa.DataFrameModel):
...     idx: int = pa.Field(unique=True)
...     adate: pd.DatetimeTZDtype = pa.Field(dtype_kwargs={
...         "unit": "s",
...         "tz": "UTC"
...     })
...     state: pd.CategoricalDtype = pa.Field(dtype_kwargs={
...         "categories": ["Ohio", "Nevada"],
...         "ordered": False
...     })
...     ohio: bool = pa.Field()
...     year: int = pa.Field(ge=2000, le=2005)
...     pop: float = pa.Field(ge=0, nullable=True)
...     # configurazioni generali: nome Config obbligatorio
...     class Config:
...         coerce = True # coercizione al tipo specificato, prechecks
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pa' is not defined
```

Una volta fatto questo effettuiamo la validazione in un `try`; se va viene salvato in `out` il `DataFrame` valido, se non va in `errs` gli errori

```
>>> try:
...     out = Model.validate(df, lazy = True)
...     # print(out)
... except pa.errors.SchemaErrors as err:
...     errs = err.failure_cases # dataframe of schema errors
...     # print("Schema errors and failure cases:")
...     # print(errs.to_string())
...     # print("\nDataFrame object that failed validation:")
...     # print(err.data) # invalid dataframe
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'Model' is not defined
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'pa' is not defined
```

Alcune considerazioni:

- purtroppo ad ora `nullable` (ossia accettare dati mancanti) deve essere specificato sempre (issue), al massimo si può usare il `partialling` con qualcosa del genere


```
>>> from functools import partial
>>> NullableField = partial(pa.Field, nullable=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pa' is not defined
>>> StdField = partial(pa.Field, coerce=True, nullable=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pa' is not defined
```

- esperimenti con interfaccia classica in `src/tests`
- per funzioni di check custom vedere la documentazione qui

Una procedura di importazione e cleaning La sequenza potrebbe essere:

- importare con qualsivoglia strumento (es `pd.read_csv`);
- coercire quanto di interesse per l'analisi con il metodo `transform` (eventualmente verificare qui l'introduzione di valori mancanti)
- validare quanto coercito mediante pandera eventualmente

18.2 Logging

```
>>> import logging

>>> level = logging.DEBUG
>>> fmt = "[% (levelname)s] %(asctime)s - %(message)s"
>>> logging.basicConfig(level = level, format = fmt)

>>> logging.info("some useful info")
>>> logging.debug("debug info")
>>> logging.error("some problems")
```

18.3 SymPy

Integrazione `integrate` è utilizzato per integrali sia indefiniti che definiti

1. definire i simboli impiegati
2. definire l'espressione che ne fa uso
- 3.

```
from sympy import *
init_printing(use_unicode=False, wrap_line=False)
x = Symbol('x')
y = Symbol('y')

# indefinito
integrate(x**2 + x + 1, x)
```

```

# definito da 0 a +infinito
expr = exp(-x**2)
integrate(expr,(x,0,oo) )

expr = (3 + x)/(12*6)
integrate(expr,(x, -3, 9) )

# definito usando due variabili
expr=exp(-x**2 - y**2)
integrate(expr,(x,0,oo),(y,0,oo))

```

18.4 Ottenere codice di oggetti

Il modulo `inspect` permette la stampa del codice sorgente di oggetti (moduli, classi, funzioni ecc):

```

>>> import inspect
>>> import re

>>> lines = inspect.getsource(re.compile)
>>> print(lines)
def compile(pattern, flags=0):
    "Compile a regular expression pattern, returning a Pattern object."
    return _compile(pattern, flags)

```

18.5 Esecuzione parallela

Usiamo `multiprocessing`, che effettua una parallizzazione tipo quella di R a livello di processo e `Pool` che applica una funzione in maniera parallela cambiando il parametro di input.

```

from multiprocessing import Pool

import os
import numpy as np
import time

# a function with no particular setting (note how we don't have to
# define the imports which is automatically handled by the fork

def parallelized(s):
    """A test function which extract 500 random number from standard
normal and calculates the mean.
    """

    rng = np.random.default_rng(seed = s)
    rv = rng.standard_normal(500) # random vector
    return {"pid" : os.getpid(),
            "seed": s,
            "mean" : rv.mean()}

```

```
def main():
    start_t = time.perf_counter()
    # Pool can be used for parallel execution of a function across
    # multiple input values, distributing the input data across processes
    # data parallelism).
    rng_seeds = [1, 1, 3]
    with Pool() as p: # spawna 1 processo per core (usandoli tutti)
        # apply a function to different input, each in one separate process
        res = p.map(parallelized, rng_seeds)
    stop_t = time.perf_counter()

    os.getpid() # pid of the current process
    print(res)  # pid of working process are different
                # results are the same in the first two cases (same seed)
    print(f"Time elapsed {stop_t - start_t:.2f}")
```

```
main()
```

Per

- limitare il numero di processi da spawnare ad esempio fare Pool(5)
- i tre metodi principali che pool fornisce sono map, imap e imap_unordered

18.6 File di configurazione

Per la scrittura di file .ini utilizzare la libreria `configparser`. Se interessa solo la lettura allo stato attuale è disponibile anche `tomllib` per i file `toml` che sono una evoluzione degli .ini

```
import configparser

# Scrittura
data = {
    'customer': "ajeje",
    'acronym': "brazorv",
    'title': "un titolo",
    'created': "2020-01-01",
    'url': "lbraglia.altervista.org"
}
fpath = "/tmp/asd.ini"
configs = configparser.ConfigParser()
configs["project"] = data # project è la sezione del file .ini
with open(fpath, 'w') as f:
    configs.write(f)

# Lettura
configs = configparser.ConfigParser()
configs.read(fpath)
```

```
print(configs["project"]["url"])

# modifica
configs["project"]["url"] = "lbraglia.github.io"
```

18.7 Calendario

`calendar` fornisce funzionalità del calendario (utile es per trovare l' x -esimo martedì del mese tal dei tali).

```
>>> import calendar
>>> import pprint
>>> from datetime import date

>>> # Stampare il calendario del mese scelto (es del mese corrente)
>>> oggi = date.today()
>>> c = calendar.TextCalendar()
>>> c.prmonth(oggi.year, oggi.month)
    maggio 2025
lu ma me gi ve sa do
           1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

>>> # ottenere numeri di giorni del mese come lista di liste (ogni lista è una settimana)
>>> m = calendar.monthcalendar(oggi.year, oggi.month)
>>> pprint.pprint(m)
[[0, 0, 0, 1, 2, 3, 4],
 [5, 6, 7, 8, 9, 10, 11],
 [12, 13, 14, 15, 16, 17, 18],
 [19, 20, 21, 22, 23, 24, 25],
 [26, 27, 28, 29, 30, 31, 0]]

>>> # la struttura di dati va per ogni settimana da lunedì a domenica
>>> # 0 vuol dire che il giorno non appartiene al mese, altrimenti il numero è
>>> # il progressi

>>> # stampa di tutti i sabato e domenica di un mese con numero di giorno
>>> for week in m:
...     sab = week[calendar.SATURDAY]
...     dom = week[calendar.SUNDAY]
...     if sab: # se è diverso da 0 vi è un sabato in quella settimana
...         print("sab", sab)
...     if dom: # questo dovrebbe essere sempre vero
...         print("dom", dom)
...
sab 3
dom 4
sab 10
```

```
dom 11
sab 17
dom 18
sab 24
dom 25
sab 31
```

18.8 Tcl/Tk

```
from pathlib import Path
from tkinter.filedialog import askopenfilename

title = "Select the study protocol file to be imported"
initialdir = "/tmp"
filetypes = [("Formati", ".docx .doc .pdf")]
fpath = Path(
    askopenfilename(title=title, initialdir=initialdir, filetypes=filetypes)
)
```

18.9 Telegram

Un esempio di invio di messaggi di monitoraggio ai termini di un task (backup fatto mediante rsync)

```
import asyncio
import datetime as dt
import os
import telegram
from pylbmisc.tg import bot_token, user_id, group_id

# telegram message
async def send_message(start, end):
    diff = end - start
    msg = """Backup completato. \n Inizio: {0} \n Termine: {1} \n Impiegati (min): {2}: """.format(
        start.strftime("%d/%m/%Y - %H:%M:%S"),
        end.strftime("%d/%m/%Y - %H:%M:%S"),
        diff.total_seconds()/60
    )
    bot = telegram.Bot(bot_token("winston_lb_bot"))
    async with bot:
        await bot.send_message(text=msg, chat_id=user_id("lucailgarb"))

if __name__ == '__main__':
    os.system("mount usb_backup")
    start = dt.datetime.now()
    os.system("rsync -avru -L --delete doc_ricordi/ usb_backup/doc_ricordi/")
    os.system("umount usb_backup")
    end = dt.datetime.now()
    asyncio.run(send_message(start = start, end = end))
```