Python

7 giugno 2023

Indice

Parte I

Linguaggio

Capitolo 1

Introduzione

1.1 Caratteristiche del linguaggio

Il python è un linguaggio:

- interpretato, di alto livello;
- OOP: **qualsiasi cosa è un oggetto** di una determinata classe, avente determinate caratteristiche e metodi;
- l'indentazione è importante ai fini dell'interpretazione del codice;
- si utilizza `#` per commento;
- se una istruzione deve continuare su due righe occorre usare \ al termine della prima;
- non serve `;` al termine di una istruzione (a meno che non si vogliano porre due istruzioni sulla stessa linea, alfine di separarle);

1.2 Setup

Pacchetti:

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

con gli ultimi due literate programming con latex (modulo `pythontex`).
Per impostare la versione 3 di python come default:

```
python -V
update-alternatives --install /usr/bin/python python /usr/bin/python3 1
update-alternatives --install /usr/bin/pip pip /usr/bin/pip3 1
update-alternatives --install /usr/bin/pydoc pydoc /usr/bin/pydoc3 1
```

Comunque negli script meglio continuare a fare shabang con `python3` esplicito
per sicurezza.

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

Tabella 1.1: Comandi `python-mode` di emacs

1.4 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 piu 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
```

1.5 Gestione sistema

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

1.5.1 Aggiornamento di sistema periodico

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

1.5.2 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).

1.5.3 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 si 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 # da github

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

## Disinstallazione pacchetto
pip uninstall project_name
```

1.5.4 Virtual environments

I virtual environments fanno si 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 l l 4096 12 apr 09.32 bin
drwxr----- 2 l l 4096 12 apr 09.32 include
drwxr----- 3 l l 4096 12 apr 09.32 lib
lrwxrwxrwx 1 l l    3 12 apr 09.32 lib64 -> lib
```

```
-rw-r----- 1 l l   69 12 apr 09.32 pyvenv.cfg
drwxr----- 3 l l 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 si 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

1.6 ipython

Per l'installazione

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

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

```
%quickref
```

1.6.1 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
```

1.6.2 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
```

1.6.3 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 si 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

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__', '__delatt
```

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

```
>>> a = 1
>>> id(a)
10861192
```

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 funzione[1], per **coercire da un tipo all'altro**:

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

[1] Questo 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

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
  - 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)
pow(x, y)    elevamento a potenza
```

```
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 omesso default a 0)
math.floor(x)     maggiore intero <= x
math.ceil(x)      minor intero >= x
```

I numeri supportano anche operatori bitwise

```
x | y    bitwise or
x ^ y    bitwise exclusive or
x & y    bitwise and
x << n   x shifted left by n bits
x >> n   x shifted right by n bits
~x the bits of x inverted
```

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 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 sequenze[2] sono quelle di tabella 2.3 Agli iterabili (di cui le sequenze fanno parte) si possono applicano le funzioni di tabella 2.4 per coercirli a sequenze.

2.5.1 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

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

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

Tabella 2.2: Operatori comuni per le sequenze

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

Tabella 2.3: Operatori per la coercizione a sequenze

Funzione	Attività
`list(iter)`	converti l'iterabile ad una lista
`str(iter)`	converti l'iterabile ad una stringa
`tuple(iter)`	converti l'iterabile ad una tuple

Tabella 2.4: Operatori per la coercizione a sequenze

- 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

```
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

2.5.2 Stringhe

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

```
>>> # up/lowcase: testing e coercizione
>>> "PROVA".islower()
False
>>> "PROVA".isupper()
True
>>> "PROVA".lower()
'prova'
>>> "prova".upper()
'PROVA'
>>> "uno due tre prova".capitalize()
'Uno due tre prova'
>>> "uno due tre prova".title()
'Uno Due Tre Prova'

>>> # centering
>>> "prova".center(70)
'                                  prova                                  '
>>> "prova".ljust(40)
'prova                                   '
>>> "prova".rjust(40)
'                                   prova'
```

```
>>> # rimozione spazi iniziali/finali
>>> "   test".lstrip()
'test'
>>> "test   ".rstrip()
'test'

>>> # test inizio e fine
>>> "prova".startswith("f")
False
>>> "prova".endswith("a")
True

>>> # ricerca da sx (prima occorrenza)
>>> "aiuola".find("a")
0
>>> "aiuola".find("u")
2
>>> "aiuola".find("x")
-1

>>> # ricerca da destra (ultima occorrenza)
>>> "aiuola".rfind("a")
5

>>> # formattazione
>>> "{0} ha {1} anni".format("Luca", 25)
'Luca ha 25 anni'
>>> "{0[1]}".format(['x','y'])
'y'
>>> "{0:.2f}".format(3)
'3.00'

>>> # testing alfanumerico
>>> "BRGLCU83S04H223C".isalnum()
True
>>> "BRGLCU83S04H223C".isalpha()
False

>>> # join con stringa usata come separatore di un iterabile
>>> " ".join("foo")
'f o o'
>>> "".join(["a", "b", "c"])
'abc'

>>> # partizionamento/splitting
>>> "amicici".partition("c")
('ami', 'c', 'ici')
>>> "amicici".rpartition("c")
('amici', 'c', 'i')
```

```
>>> "amicici".split("c")
['ami', 'i', 'i']
>>> "linea1\nlinea2".splitlines()
['linea1', 'linea2']

>>> # rimpiazzo
>>> "prova".replace("a", "e")
'prove'

>>> # aggiunta di 0 iniziali
>>> "123".zfill(4)
'0123'
```

2.5.3 Liste

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

2.5.3.1 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']
```

2.5.3.2 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)`.

2.5.3.3 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'
```

2.5.3.4 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)]
>>> print(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
>>> print([x for x in vec if x >= 0])
[0, 2, 4]

>>> ## Applicare una funzione a tutti gli elementi di una lista
>>> print([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
>>> print([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
>>> print([">=0" if i>=0 else "<0" for i in vec])
['<0', '<0', '>=0', '>=0', '>=0']

>>> ## creiamo una lista composta da tuple
```

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

>>> ## 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)]
>>> print(t)
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

2.5.3.5 Metodi utili per le liste

```
>>> l = []

>>> # append, aggiunge un elemento
>>> l.append(1)

>>> # aggiunge elementi da un iterabile
>>> l.extend([1,2,3])

>>> # rimuove tutti gli elementi
>>> l.clear()

>>> # indice di un elemento (prima occorrenza)
>>> ["a", "x", "c", "x"].index("x")
1

>>> # inserisce un oggetto prima dell'indice
>>> a = ["a", "c", "d"]
>>> a.insert(1, "b")

>>> # rimuove la prima occorrenza di un oggetto
```

```
>>> a.remove("c")

>>> # rimuove il valore all'indice specificato e lo ritorna
>>> x = a.pop(1)

>>> # conta le occorrenze di un dato elemento
>>> [1,2,1,3].count(1)
2

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

>>> #stampa reversed
>>> x.reverse()
```

2.5.4 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')
>>> print(atuple)
('robots', 77, 93, 'try')
>>> ## subset
>>> print(atuple[:3])
('robots', 77, 93)
>>> ## tuple nested
>>> a = (1, 2)
>>> b = ('x', 'y')
>>> c = (a, b)
>>> print(c)
((1, 2), ('x', 'y'))

>>> ## modifica diretta? no.
>>> 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
>>> print(btuple)
([3, 2], 'abc')

>>> ## Metodi utili, uguali a quelli delle liste
>>> atuple.count(93)
1
```

```
>>> atuple.index("try")
3
```

2.5.5 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'
```

2.6 Classi mapping e set

2.6.1 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}
>>> print(d)           # stampa complessiva
{'planet': 'earth', 'region': 'europe', 'prefix': 39}
>>> print(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
>>> print(d.keys())    # ottenere le chiavi
dict_keys(['planet', 'region', 'prefix', 'foo'])
>>> print('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)
```

2.6.1.1 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()
```

2.6.1.2 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
```

2.6.2 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'}
>>> ## Gli elementi doppi vengono eliminati
>>> print(basket)
{'pear', 'apple', 'orange'}

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

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

>>> ## Test appartenenza
>>> print('orange' in basket)
True
>>> print('strawberry' in basket)
False

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

2.6.2.1 Operatori/metodi utili

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

>>> ## Operazioni insiemistiche
>>> print("Set difference: ", a - b) # lettere in a ma non in b
Set difference:  {'r', 'b', 'd'}
>>> print("Set or: ", a | b) # lettere in a o in b
```

```
Set or:  {'a', 'l', 'r', 'd', 'z', 'm', 'b', 'c'}
>>> print("Set and: ", a & b) # lettere sia in a che in b
Set and:  {'a', 'c'}
>>> print("Set ^: ", a ^ b) #xor: lettere in a o in b ma non in entrambi
Set ^:  {'m', 'b', 'l', 'r', 'z', 'd'}

>>> ## 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
```

2.6.2.2 Set comprehension

Le comprehension sono ammesse anche nei set

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

2.7 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.

2.7.1 Sintassi

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

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

>>> # spezzato
>>> x: int
>>> x = "4"  # qua viene dato errore
```

```
>>> # 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
```

2.7.2 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 .
```

2.7.3 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"
```

2.7.4 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:
    ...
```

2.7.5 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
```

2.7.6 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

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 espresioni (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

```
<        minore
<=       minore o uguale
>        maggiore
>=       maggiore o uguale
==       equal
!=       not equal
is       due oggetti sono identici
is not   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 numeri 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

Le funzioni sono first class object, anche se non hanno metodi di particolare interesse.

4.1 Definizione

Avviene mediante la keyword `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

In definizione, *gli argomenti devono essere specificati nell'ordine di cui sopra* e sono:

1. *positional argument* (ad esempio `arg1`): si specifica solamente il nome dell'argomento. In chiamata è obbligatorio specificarli, inserendo alternativamente il valore o `nome = valore`; solo in questo secondo caso sarà possibile seguire un ordine di argomenti diverso da quello specificato in definizione;

2. *keyword argument* (ad esempio `arg2`): in sede di chiamata non sono obbligatori da specificare perchè in definizione si è fornito un valore di default;

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 Di default gli argomenti passati alle funzioni possono essere passati in sede di chiamata sia per posizione che utilizzando la keyword. Per leggibilità e performance si può in sede di 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
```

4.2 Chiamata di funzione

In chiamata si deve:

- specificare tutti gli argomenti per i quali in definizione non si siano specificati dei valori default;
- specificare gli argomenti posizionali prima dei keyworded;
- gli argomenti posizionali verranno associati nell'ordine specificato in definizione;
- se si specificano tutti `nome = valore` l'ordine può differire da quello della 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

La valutazione dei valori di default:

- 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 il valore di default è un oggetto modificabile (lista, dict, classe) modifiche entro il corpo della funzione rimarranno per le prossime chiamate. Fino a che si tratta di costanti non ci sono problemi:

```
>>> def f(x = 1):
...     print(x)
...     x = 2
...
>>> f() # stampa 1
1
>>> f() # non è vero che stampa 2 (una costante, 1, non è modificabile)
1
```

Venendo però ad oggetti modificabili, le modifiche ai parametri di default rimangono per le prossime chiamate. Ad esempio la seguente funzione accumula gli argomenti passati ad essa in chiamate successive

```
>>> def f(x, L = []): # una lista è modificabile
...     L.append(x)
...     return L
...
>>> print(f(1))
[1]
>>> print(f(2))
[1, 2]
>>> print(f(3))
[1, 2, 3]
```

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

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

4.3 Altre amenità

4.3.1 Valore ritornato

Se si utilizza l'istruzione `return` come nel caso precedente, 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`.

4.3.2 Stringa di documentazione

Da porre nella definizione 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.

4.3.3 Espressioni lambda

La keyword `lambda` serve per creare piccole funzioni anonome. Ad esempio la seguente ritorna la somma dei due argomenti passati:

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

Le funzioni anonime:

- sintatticamente possono essere usate dove è necessaria una funzione (il suo nome), ad esempio per definire una funzione al volo e usarla in una chiamata il seguente è codice valido:

  ```
  >>> (lambda x, y: print(x * y))(2,3)
  6
  ```

- sono limitate a una singola espressione come corpo
- dietro le quinte avviene la creazione di una funzione normale mediante `def`

4.4 Programmazione funzionale

In Python le funzioni sono first class object (quindi possono essere date in input e ritornate come output. Questo rende possibile la programmazione funzionale, nella quale si fa uso di alcune caratteristiche del linguaggio come:

- in merito ai dati analizzati: iterabili e iteratori (quindi anche generatori ed espressioni generatrici) e le funzioni utili sugli stessi come `enumerate`, `sorted`, `any`, `all`, `zip`
- il modulo `itertools`: iteratori comuni e funzioni per elaborarli
- funzioni builtin tipiche della pf: `map`, `filter` etc
- il modulo `functools`: high order functions (funzioni che processano funzioni modificandole)
- il modulo `operator` contiene un insieme di funzioni che corrispondono agli operatori del python e possono aiutare in un approccio funzionale evitandoci di scrivere funzioni triviali per effettuare singole operazioni
- piccole funzioni e lambda expressions

4.4.1 Funzioni classiche della pf

Duplicano un po' le funzionalità svolte dalle espressioni generatrici o list comprehension ma sono un classico della programmazione funzionale.

4.4.1.1 `map` e `starmap`

La builtin

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

restituisce un iteratore sulla sequenza

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

Ad esempio:

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

Viceversa:

```
itertools.starmap(function, iterable)
```

crea un iteratore che valuta la funzione utilizzando argomenti ottenuti dall'iterabile; da utilizzare al posto di `map` quando gli argomenti sono già stati raggruppati in tuple zippati nell'iterabile passato

```
>>> import itertools

>>> def custom_f(x, y, z):
...    return(x * y - z)
...
>>> res = list(itertools.starmap(custom_f, [(2,5,2), (3,2,1), (10,3,2)]))
>>> print(res)
[8, 5, 28]
```

4.4.1.2 `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`

4.4.1.3 `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

>>> res = reduce(add, [1,2,3])
>>> res
6

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

4.4.1.4 `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]
```

4.4.2 Function factory

Le function factory fanno uso della capacità del linguaggio di ritornare una funzione che fa uso del namespace della chiamante per la risoluzione di alcune free variable. Vediamo una implementazione di una function factory che crea funzioni potenze in base al parametro passatogli

```
>>> def make_power(n):
...     def power(x):
...             print(x ** n)
...     return power
...
>>> square = make_power(2)
>>> square(3) # restituisce 9
9

>>> # definizione con funzione anonima è più compatta
>>> # a testo, forse meno leggibile
```

```
>>> def lambda_power(n):
...     return lambda x: print(x ** n)
...
>>> squarel = lambda_power(2)
>>> squarel(3) # restituisce 9
9
```

4.4.3 Composizione di funzioni

Sempre sfruttando questa caratteristica del linguaggio, per comporre diverse funzioni in una sola si può definire la seguente

```
>>> def compose(*funs):
...     """
...     Return a new function which is composition in math sense
...     compose(f,g,...)(x) = f(g(...(x)))
...     """
...     def worker(data, funs = funs):
...         result = data
...         for f in reversed(funs):
...             result = f(result)
...         return result
...     return worker
...
>>> def add2(x):
...     return x+2
...
>>> def mul2(x):
...     return x*2
...
>>> 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]
```

4.4.4 Decorators

Un decorator è una funzione, solitamente[1], che wrappa un'altra funzione modificandone il comportamento standard in qualche modo; 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
```

[1] Decorator può essere qualsiasi callable, ovvero un oggetto che presenta il metodo `__call__`

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

foo = decorator(foo)  # Manually decorate

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

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 si che ad una funzione base vengano applicati più decoratori 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)
...
```

4.4.5 Funzioni con singledispatch

Sono funzioni tipo le S3 di R che stabiliscono come comportarsi (evitando `if`/`elif`) in ragione dell'input fornito.
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 0x7f2d2a0551c0>

>>> 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
```

4.4.6 Partialling di una funzione

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

```
>>> 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
```

4.4.7 List of functions

Se vogliamo applicare un insieme di funzioni ad un input comune possiamo fare essere ottenuto mediante un ciclo su tuple che contiene le funzioni. Un esempio banale è:

```
>>> def per1(x):
...     return x
...
>>> def per2(x):
...     return x*2
...
>>> def per3(x):
...     return x*3
...
>>> fun_tuple = (per1, per2, per3)
>>> res = [f(2) for f in fun_tuple]
>>> print(res)
[2, 4, 6]
```

Capitolo 5

Input/Output

5.1 File testuali

5.1.1 File di testo semplice

La funzione `open` prende in input un path e un mode (`r` per la lettura, `w` per la scrittura, `a` per l'appending) 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 `print` (specificando il file).

```
with open(file = '/tmp/test.txt', mode = 'w') as f:
    print("test", file = f)
```

Alternativamente il metodo `write` (scrive il contenuto della stringa passatagli al file, restituendo il numero di caratteri scritti)

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']
        ...
```

Python	JSON
`dict`	object
`list`, `tuple`	array
`str`	string
`int`, `float`	number
`True`	true
`False`	false
`None`	null

Tabella 5.1: Encoding JSON

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

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

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

Tabella 5.2: Decoder JSON

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 Utilities di sistema

5.2.1 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]
```

5.2.2 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/tmpwocmkjue
>>> # 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.

5.2.3 Interfaccia col filesystem

Si usa oggigiorno `pathdir`, per alcune cose marginali rimasto `os` e `shutil`

5.2.3.1 Cambiare directory di lavoro

Si usa os

```
>>> import os
>>> os.chdir('/tmp')
```

5.2.3.2 `pathlib`: creazione/rimozione di file e directory

```
>>> from pathlib import Path

>>> # Navigazione
>>> Path.cwd()
PosixPath('/tmp')
>>> Path.home()
PosixPath('/home/l')

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

>>> # Creazione della directory
>>> d.mkdir() # probs
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/pathlib.py", line 1117, in mkdir
```

```
    os.mkdir(self, mode)
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/barbaz/whahaa/yeah'
>>> d.mkdir(parents = True) # no probs

>>> # Rimozione
>>> rm = Path('/tmp/barbaz')
>>> rm.rmdir() # must be empty...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/pathlib.py", line 1157, in rmdir
    os.rmdir(self)
OSError: [Errno 39] Directory not empty: '/tmp/barbaz'
>>> import shutil
>>> shutil.rmtree(rm) # vedere dopo per shutil

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

5.2.3.3 `pathlib`: manipolazione di path e metodi utili

Si usava in passato `os.path` (utilizzata tuttora sotto la scocca), l'interfaccia ad oggetti nuova è `pathlib.Path`. Alcuni dati e operazioni di base:

```
>>> f = Path("/usr/bin/python")
>>> d = Path("~/.sintesi")
>>> de = d.expanduser() # sostituire la tilde per l'amor del cielo

>>> ## 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'

>>> ## creare nuovi path con /
>>> subdir = "sintesi_cs"
>>> d / subdir
PosixPath('~/.sintesi/sintesi_cs')
>>> d.joinpath(subdir) # stessa cosa
PosixPath('~/.sintesi/sintesi_cs')

>>> # test vari
>>> f.exists()
True
>>> d.exists()      # ocio a ~ ed exists
False
>>> de.exists()
```

```
True
>>> d.is_dir()
False
>>> f.is_file()
True
>>> f.is_symlink()   # symlink stuff
True
>>> f.readlink()
PosixPath('python3')
>>> import os
>>> os.path.realpath('/usr/bin/python3')
'/usr/bin/python3.11'

>>> # manipolazioni per file (ma applicabili anche a directory)
>>> f2 = Path("/etc/apt/sources.list")
>>> f2.name  # nome file
'sources.list'
>>> f2.stem  # nome file senza estensione
'sources'
>>> f2.suffix # estensione - per tar.gz usare suffixes
'.list'

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

>>> f.with_name("foo.list") # crea un path con nome cambiato
PosixPath('/usr/bin/foo.list')
>>> f.with_stem("ssd") # crea un path con stem cambiato e lasciando estensione
PosixPath('/usr/bin/ssd')
>>> f.with_suffix(".deb") # crea un path con estensione cambiato e uguale stem
PosixPath('/usr/bin/python.deb')

>>> # Altre operazioni: path assoluti e relativi
>>> f2.relative_to("/etc") # crea un path relativo escludendo quanto passato
PosixPath('apt/sources.list')

>>> f3 = Path("Makefile") # ottenimento di un path assoluto dal relativo
>>> f3.absolute()
PosixPath('/tmp/Makefile')
>>> f3.resolve()  # questo fixa anche symlink e cazzi vari
PosixPath('/tmp/Makefile')
```

5.2.3.4 `pathlib`: listing di directory e glob

```
>>> # iterdir: restituire i path del contenuto di una directory (yield)
>>> list(de.iterdir())
[PosixPath('/home/l/.sintesi/sintesi_misc'), PosixPath('/home/l/.sintesi/sintesi_cs'), PosixPat

>>> # glob: cerca i file che matchano. evitare tilde
>>> # glob restituisce un iteratore btw che qua listordinimo con sorted
```

```
>>> sorted(de.glob("*/*.pdf"))  # cerca nella sottodirectory figlie
[PosixPath('/home/l/.sintesi/dae/lm.pdf'), PosixPath('/home/l/.sintesi/dae/py_descrip
>>> sorted(de.glob("**/*.pdf")) # cerca in tutto l'albero
[PosixPath('/home/l/.sintesi/dae/lm.pdf'), PosixPath('/home/l/.sintesi/dae/py_descrip
>>> sorted(de.rglob("*.pdf"))   # stessa cosa di sopra ma più compatta
[PosixPath('/home/l/.sintesi/dae/lm.pdf'), PosixPath('/home/l/.sintesi/dae/py_descrip
```

5.2.3.5 `shutil`: utilities shell-like di alto livello

```
>>> import shutil

>>> # rimuovere directory senza far complimenti
>>> p = Path("/tmp/foobar/baz/asdomar")
>>> p.mkdir()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/pathlib.py", line 1117, in mkdir
    os.mkdir(self, mode)
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/foobar/baz/asdomar'
>>> shutil.rmtree(Path("/tmp/foobar"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/shutil.py", line 722, in rmtree
    onerror(os.lstat, path, sys.exc_info())
  File "/usr/lib/python3.11/shutil.py", line 720, in rmtree
    orig_st = os.lstat(path, dir_fd=dir_fd)
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/foobar'
```

Capitolo 6

Debugging ed eccezioni

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.

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

Tabella 6.1: Comandi debug

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)
%debug
! a
! b
u
quit
```

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 nomescript.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
```

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")
...
Ops valore non coercibile
```

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
```

Capitolo 7

Object Oriented Programming

Prima della programmazione ad oggetti approfondiamo le regole di scoping, ovvero come Python associa i valori ai nomi che trova nelle sintassi. Questo perchè l'implementazione della programmazione ad oggetti si basa su trick di scope.

7.1 Regole di scope

7.1.1 Namespace

Un *namespace* è qualcosa che mappa nomi ad oggetti (il contenuto) presenti in memoria; sono *creati in diversi momenti* e hanno *differente durata*.
I namespace principali sono:

- l'insieme dei nomi **builtin**: creato quando l'interprete python si avvia e dura sino al termine dell'esecuzione);
- l'insieme di nomi di un **modulo** (sia esso `__main__` o moduli importati): è reso disponibile quando esso viene caricato (e anche esso generalmente dura sino al termine dell'esecuzione);
- i nomi di **funzione**: è creato quando essa è chiamata e viene distrutto quando la funzione ritorna o lancia una eccezione (ad esempio warning) non gestita; le funzioni ricorsive hanno un namespace per ogni chiamata.

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

```
>>> ## stampa dei nomi builtin del linguaggio
>>> dir(__builtins__)
['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__
>>> ## stampa dei nomi presenti nel modulo globale (workspace)
>>> dir()
['Path', 'S', 'TemporaryDirectory', 'TemporaryFile', 'X', 'Y', '_', '__annotations__', '__built
>>> ## stampa i nomi di un modulo importato
>>> import sys
```

```
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', '__interactive
>>> ## stampa i nomi in una funzione
>>> def f():
...     a = 1
...     b = 2
...     print(dir())
...
>>> f()      ## stampa i nomi di una funzione
['a', 'b']
```

7.1.2 Attributi

In Python il termine indica *qualsiasi* nome che segue un punto: ad esempio in `z.real`, `real` (qualsiasi cosa sia) è un attributo di `z`.
Gli attributi possono essere read-only o scrivibili; nel secondo caso:

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

7.1.3 Ricerca standard ed eccezioni: `global` e `nonlocal`

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*, ossia quella all'interno del quale la funzione corrente è stata definita;
3. la enclosing della enclosing, e così via;
4. il *namespace del modulo* corrente, sia esso main o importato
5. il namespace delle builtin functions.

Alcuni casi particolari:

- se un nome è dichiarato globale mediante la keyword `global`, allora la lettura e scrittura da tale variabile va a inficiare il namespace del modulo, indipendentemente da dove essa sia stata effettuata
- `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)
...
>>> def scope_nonlocal_read():
```

```
...     a = 3
...     def nested():
...         nonlocal a
...         print(a)
...     nested()
...
>>> def scope_nonlocal_write():
...     a = 3
...     def nested():
...         nonlocal a
...         a = 4
...     nested()
...     print(a)
...
>>> def scope_global():
...     global a
...     print(a)
...
>>> scope_local()
2
>>> scope_nonlocal_read()
3
>>> scope_nonlocal_write()
4
>>> scope_global()
1
```

7.2 Classi

Quello che fa la definizione di una classe è semplicemente porre un altro namespace all'interno di quello nel quale ci si trova.

7.2.1 Definizione e class object

La definizione di una classe fa uso di `class` associato ad un nome:

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

Gli `statement` sono assegnazione di variabili e definizione di funzioni con una sintassi particolare (i *metodi* della classe).
Una volta che si esce dalla 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 (per lettura/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*.

```
>>> # definizione di classe
>>> class firstClass:
...     """La mia prima classe"""
...     a = 0
...     def get_a(self):
...         return(self.a)
...     def set_a(self, x):
...         self.a = x
...
>>> # modifica di un attributo di una classe
>>> firstClass.a = 1
>>> print(firstClass.__doc__) ## stampa la docstring della classe
La mia prima classe

>>> # istanziamento mediante uso del costuttore di default
>>> x = firstClass()
>>> print(x.get_a())          ## stampa 1
1
>>> x.set_a(3)
>>> print(x.get_a())          ## stampa 3
3
```

7.2.2 Metodi

7.2.2.1 Definizione

I metodi di una classe presentano una definizione particolare:

- il primo argomento serve sempre per riferirsi all'oggetto istanziato: il nome `self` è del tutto arbitrario, ma preferibile perchè standard;

- gli altri argomenti sono parametri normali da passare alla chiamata di funzione.

Per Python chiamare `object.function()` è equivalente a chiamare `classe.function(object)`; chiamare un metodo con una lista di n argomenti equivale chiamare la funzione corrispondente con una lista di argomenti creata inserendo l'oggetto del metodo come primo argomento.

7.2.2.2 Costruttore custom

Quando si crea un oggetto istanza, se non abbiamo specificato il *costruttore* ne viene utilizzato uno di default; se viceversa si desidera specificare 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()
>>> print(x.value())
0+0i
>>> y = Complex(1, 2)
>>> print(y.value())
1+2i
```

7.2.2.3 Scope

A parte le variabili locali, la ricerca continua nel modulo dove la classe è definita (ad esempio nel quale dobbiamo usare import per permettere al modulo della nostra classe di usare codice da un'altra).

7.2.3 Dati della classe/oggetto

7.2.3.1 Dati condivisi tra istanze o no

I dati/variabili di una classe iniziano ad esistere dal momento in cui vengon assegnati (sia in definizione della classe o usando un suo 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 siano 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'
```

Bisogna prestare attenzione a quando come dato di classe vi è un tipo mutabile, perchè la chiamata di metodi da diverse istanze andrà a modificare un dato comune (e non è spesso quello che si vuole). Un esempio erroneo:

```
>>> class Dog:
...     tricks = []             # mistaken use of a class variable
...     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                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

e la versione corretta

```
>>> class Dog:
...     def __init__(self, name):
...         self.name = name
...         self.tricks = []    # creates a new empty list for each dog
...     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']
```

7.2.3.2 Accesso agli elementi e data hiding

Di un oggetto istanziato si può accedere ai valori o funzioni senza problemi: nel Python non vi è un concetto di *data hiding*. Pertanto:

- si tratta solamente di una convenzione (ma da adottare) di usare i metodi messi a disposizione invece che modificare le variabili direttamente;
- se si desidera celare un elemento, sempre come convenzione gli si può dare un nome che inizi con underscore (es alle cose che dovrebbero essere considerate come private).
- sempre come convenzione, e per incrementare la celatura, se si da un nome che inizia con due underscore, 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 e stampa 0
0
```

7.3 Ereditarietà

7.3.1 Definizione classe derivata

Una classe che eredita da un'altra in Python si definisce nel seguente modo

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

ponendo la classe base dalla quale si eredita tra parentesi (nel caso di classe definita in un pacchetto possiamo mettere tra parentesi `pacchetto.ClasseBase`). La definizione e l'istanziazione per il resto è simile a una generica classe, ad eccezione che:

- la classe di base è utilizzare per risolvere nomi, nel caso non vengano trovati nella classe derivata. La ricerca è ricorsiva verso l'alto nella gerarchia delle classi (ad esempio se anche la classe di base eredita da qualcos'altro)
- Le classi derivate possono specializzare ad esempio i metodi ridefinendoli con il medesimo nome della classe base

7.3.2 Ereditarietà multipla

Nel caso definiamo una classe come segue

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

la classe eredita da tutte e tre le classi di base. Indicativamente, la risoluzione di nomi funziona prima in profondità (cercando anche nelle classi dalle quali queste ereditano) `Base1`, poi poi passando a `Base2` (e quindi in profondità) e infine per `Base3`(e quindi sempre in profondità). Si evita di cercare due volte nella stessa classe se vi sono sovrapposizioni nell'albero genealogico.

Capitolo 8

Classi notevoli

8.1 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.

8.1.1 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 0x7f2d2a082cb0>
>>> 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)
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 cosi il processo di iterazione;

3. quando non vi sono altri elementi sui quali iterare, `next()` solleva l'eccezione `StopIteration` e `for` termina.

8.1.2 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
```

8.1.3 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

8.1.4 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 0x7f2d2a098660>
>>> r.__next__
<method-wrapper '__next__' of generator object at 0x7f2d2a098660>

>>> 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.

8.1.5 Funzioni e operatori utili su iteratori

8.1.5.1 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 l
2 e
3 t
```

8.1.5.2 Operatori

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

8.1.6 Il modulo itertools

Contiene funzioni per la creazione e gestione di iteratori

8.1.6.1 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
```

8.1.6.2 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']
```

8.1.6.3 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')
```

8.1.6.4 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)

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

8.1.6.5 Prodotto cartesiano

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

8.2 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.

8.2.1 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__`

8.2.2 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

8.3 `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 inizializzatore 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)

8.3.1 `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

8.3.2 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}"
```

8.3.3 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

8.3.4 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
  ```

Capitolo 9

Moduli e pacchetti

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

9.1 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.

9.2 Moduli

9.2.1 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).

9.2.2 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 utilizzale

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

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

- `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)

9.2.3 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')
   ```

9.2.4 Un template

```
#!/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()
```

9.3 Pacchetti

9.3.1 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 si 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

9.3.1.1 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
```

9.3.1.2 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`):

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

References da esterno L'utilizzo (sempre da uno script residente nella directory di `sound`) può essere, alternativamente:

```
# importazione di un modulo
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

# importazione di un modulo, vrs2
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)

# importazione di una funzione
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 di `sound.effects.echo` e di `sound.filters.equalizer` nelle prime linee si può usare:

```
from sound.effects import echo
from ..effects import echo
from . import equalizer
```

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

Di base `import` applicato ad un pacchetto esegue innanzitutto `__init__.py` per eventuali configurazioni/inizializzazioni.
Sebbene sia considerata bad practice, se in `__init__.py` si imposta la variabile

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

e si comanda

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

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

9.4 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.

9.4.1 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

9.4.2 `pyproject.toml`

In `pylb/pyproject.toml` specificare le main config del repoin 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`).

9.4.3 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
```

9.4.4 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.

9.4.5 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`.

9.4.6 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/
```

9.5 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/`.

9.5.1 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.

9.5.2 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.

9.5.3 Building

9.5.3.1 Setup autobuilding API

Per fare si 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
```

9.5.3.2 Build definitivo

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

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
```

latex

```
l@m740n:~/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@m740n:~/src/pypkg/pylb/docs$ make latexpdf
l@m740n:~/src/pypkg/pylb/docs$ okular _build/latex/pylb.pdf
```

9.5.4 Setup di readthedocs

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

9.6 Inserimento di script

Per inserire script 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`.

9.7 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 programmiamo 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

9.8 Timing/temporizzazione

Per misurare il tempo per l'esecuzione usiamo le funzionalità di ipython e soprattutto il magic command `timeit`, il quale esegue codice in maniera ripetuta e stampa statistiche descrittive.
Di fatto usa qualcosa del genere

```
import time
start = time.time()
for i in range(iterations):
    # some code to run here
elapsed_per = (time.time() - start) / iterations
```

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)
     ...:
  ```

9.9 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`

9.9.1 Tempo

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

9.9.2 Memoria

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

9.10 Altri strumenti per lo sviluppo

Controllo del codice Effettuarlo mediante il tool `flake8`

```
flake8 project_dir
```

Vedere la relativa pagina di manuale

Capitolo 10

Testing

10.1 Introduzione e concetti

10.1.1 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

10.1.2 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

10.2 `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`.

10.2.1 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@m740n:~/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@m740n:~/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.

10.2.2 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:

```
l@m740n:~/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
```

10.2.3 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))
```

Capitolo 11

Cookbook

11.1 Importare dataset

11.1.1 Di R

Si possono importare dati dal pacchetto Rdatasets attraverso il pacchetto `rdatasets` o `statsmodels`, entrambi usano

```
>>> import statsmodels.api as sm
>>> airquality = sm.datasets.get_rdataset("airquality", "datasets")
>>> df = airquality.data
>>> df
     Ozone  Solar.R  Wind  Temp  Month  Day
0     41.0    190.0   7.4    67      5    1
1     36.0    118.0   8.0    72      5    2
2     12.0    149.0  12.6    74      5    3
3     18.0    313.0  11.5    62      5    4
4      NaN      NaN  14.3    56      5    5
..     ...      ...   ...   ...    ...  ...
148   30.0    193.0   6.9    70      9   26
149    NaN    145.0  13.2    77      9   27
150   14.0    191.0  14.3    75      9   28
151   18.0    131.0   8.0    76      9   29
152   20.0    223.0  11.5    68      9   30

[153 rows x 6 columns]
>>> print(airquality.__doc__)
.. container::

   ========== ===============
   airquality R Documentation
   ========== ===============

   .. rubric:: New York Air Quality Measurements
      :name: airquality
```

```
.. rubric:: Description
   :name: description

Daily air quality measurements in New York, May to September 1973.

.. rubric:: Usage
   :name: usage

::

   airquality

.. rubric:: Format
   :name: format

A data frame with 153 observations on 6 variables.

======== =========== ======= ======================
``[,1]`` ``Ozone``   numeric Ozone (ppb)
``[,2]`` ``Solar.R`` numeric Solar R (lang)
``[,3]`` ``Wind``    numeric Wind (mph)
``[,4]`` ``Temp``    numeric Temperature (degrees F)
``[,5]`` ``Month``   numeric Month (1--12)
``[,6]`` ``Day``     numeric Day of month (1--31)
======== =========== ======= ======================

.. rubric:: Details
   :name: details

Daily readings of the following air quality values for May 1, 1973 (a
Tuesday) to September 30, 1973.

-  ``Ozone``: Mean ozone in parts per billion from 1300 to 1500 hours
   at Roosevelt Island

-  ``Solar.R``: Solar radiation in Langleys in the frequency band
   4000-7700 Angstroms from 0800 to 1200 hours at Central Park

-  ``Wind``: Average wind speed in miles per hour at 0700 and 1000
   hours at LaGuardia Airport

-  ``Temp``: Maximum daily temperature in degrees Fahrenheit at La
   Guardia Airport.

.. rubric:: Source
   :name: source

The data were obtained from the New York State Department of
Conservation (ozone data) and the National Weather Service
(meteorological data).
```

```
   .. rubric:: References
      :name: references

   Chambers, J. M., Cleveland, W. S., Kleiner, B. and Tukey, P. A.
   (1983) *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

   .. rubric:: Examples
      :name: examples

   ::

      require(graphics)
      pairs(airquality, panel = panel.smooth, main = "airquality data")
```

11.1.2 da Stata

Si usa *webuse* che di default li scarica da qui.

```
>>> auto = sm.datasets.webuse("auto")
>>> auto
             make  price  mpg  rep78  ...  turn  displacement  gear_ratio   foreign
0     AMC Concord   4099   22    3.0  ...    40           121        3.58  Domestic
1       AMC Pacer   4749   17    3.0  ...    40           258        2.53  Domestic
2      AMC Spirit   3799   22    NaN  ...    35           121        3.08  Domestic
3   Buick Century   4816   20    3.0  ...    40           196        2.93  Domestic
4   Buick Electra   7827   15    4.0  ...    43           350        2.41  Domestic
..            ...    ...  ...    ...  ...   ...           ...         ...       ...
69      VW Dasher   7140   23    4.0  ...    36            97        3.74   Foreign
70      VW Diesel   5397   41    5.0  ...    35            90        3.78   Foreign
71      VW Rabbit   4697   25    4.0  ...    35            89        3.78   Foreign
72    VW Scirocco   6850   25    4.0  ...    36            97        3.78   Foreign
73      Volvo 260  11995   17    5.0  ...    37           163        2.98   Foreign

[74 rows x 12 columns]
```

11.2 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"
```

11.3 Generazione numeri casuali

```
>>> import numpy as np
>>> rng = np.random.default_rng(123) # seed
>>> rng.standard_normal(10)
array([-0.98912135, -0.36778665,  1.28792526,  0.19397442,  0.9202309 ,
        0.57710379, -0.63646365,  0.54195222, -0.31659545, -0.32238912])
```

11.4 Esecuzione di programmi esterni

Si usi `subprocess.run`, specificando eventuali comandi composti da più token come lista

```
import subprocess
subprocess.run("ls") # senza riprendersi i risultati
subprocess.run(["git", "clone", repo, localdir]) #
```

11.5 Tcl/Tk

11.5.1 Path a file

```
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)
)
```

11.6 Telegram

Parte II

Scientific Stack

Capitolo 12

Numpy

I dati provengono in una gran varietà di formati (documenti, immagini, suoni, misurazioni varie etc); nonostante l'apparente eterogeneità possono essere rappresentati come array di numeri (vettori di numeri, matrici di colori). Diviene pertanto fondamentale gestire efficientemente gli array numerici.
Le liste non sono un modo efficiente perché devono permettere la diversità degli elementi; quando come spesso avviene gli elementi contenuti sono omogenei, si possono adottare strutture dati più efficienti. Il builtin `array` è una, ma fornisce solamente un modo per immagazzinare dati.
Il pacchetto `numpy` fornisce la struttura dati richiesta e altro, principalmente:

- 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

Il template per l'importazione è

```
>>> import numpy as np
```

12.1 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 attribito `shape`, tuple che definisce la struttura dell'array;

12.1.1 Creazione

Il modo generale per creare un `ndarray` è mediante l'uso della funzione `array`, al quale si passa dati di tipo sequenza (liste, tuple ecc); oltre a questa diverse funzioni sono utili per la generazione di array. Alcuni esempi a seguire:

```
>>> np.array([ 1,  2,  3])           # creazione a partire da una lista
array([1, 2, 3])
>>> x = np.array([[1, 2, 3],         # creazione a partire da una lista
...               [4, 5, 6]])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> 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.full((3, 5), 2)               # array 3x5 riempito di 2
array([[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]])
>>> np.empty(3)                      # array non inizializzati (dati in memoria)
array([4.9e-324, 9.9e-324, 1.5e-323])
>>> np.zeros_like(x)                 # array di 0 della stessa shape di x (un array)
array([[0, 0, 0],
       [0, 0, 0]])
>>> np.ones_like(x)                  # array di 1 "     "      "     "  "  "    "
array([[1, 1, 1],
       [1, 1, 1]])
>>> np.empty_like(x)                 # array non inizializzato "     "  "  "    "
array([[0, 0, 0],
       [0, 0, 0]])
>>> 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
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
>>> np.random.random((3, 3))         # array 3x3 con valori casuali uniformi tra 0  e
array([[0.5538661 , 0.96649402, 0.30961011],
       [0.24311885, 0.84107225, 0.12756299],
       [0.1191217 , 0.47908548, 0.40700473]])
>>> np.random.normal(0, 1, (3, 3))   # array 3x3 da normale mu=0, sd=1
array([[-0.00344499, -0.25308588, -0.11567522],
       [-0.68592283,  0.2511829 , -0.00415532],
       [ 1.27726141,  0.29611952,  1.23731544]])
>>> np.random.randint(0, 10, (3, 3)) # array 3x3 con interi nell'intervallo [0,10 )
array([[0, 9, 0],
       [4, 6, 2],
       [5, 0, 8]])
>>> np.eye(3)                        # array 3x3 con matrice identita
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

12.1.2 `dtype`, coercizione e testing

I tipi di dato forniti da `numpy`, chiamati `dtype`, sono riportati in tabella 12.1. Per utilizzarli in sede di creazione dell'array specificare l'omonimo parametro in uno di questi due modi possibili

```
np.zeros(10, dtype = 'int16')
np.zeros(10, dtype = np.float_)
```

dtype	Descrizione
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
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)
float_	Shorthand for float64.
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
complex_	Shorthand for complex128.
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 12.1: dtypes di numpy

Coercizione del `dtype` Il modo per convertire una array da un tipo ad un altro è usare il metodo `astype`

```
>>> a = np.zeros(10, dtype = np.float_)
>>> b = a.astype(np.int8)
>>> b
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int8)
```

Test del tipo Per il check del tipo torna utile che i tipi `numpy` abbiano una gerarchia per la quale ad esempio gli interi derivano dalla classe genitrice `np.integer` mentre i numeri a virgola mobile da `np.floating`. Queste classi possono essere usate congiuntamente con la funzione `np.issubdtype`.

```
>>> np.issubdtype(a.dtype, np.integer)
False
>>> np.issubdtype(a.dtype, np.floating)
True
```

12.1.3 `ndim`, `shape`, `size`, `nbytes`

Ogni array ha attributo `ndim` (numero di dimensioni) e `shape` (numero di elementi per ciascuna dimensione), `size` (numero di elementi complessiva) ed `nbytes` (peso in memoria complessivo in bytes, dipendente dal peso del tipo adottato investigabile con `itemsize`).

```
>>> z1  = np.zeros(shape = 3)                          #one dimension
>>> z2i = np.zeros(shape = (3,4), dtype = np.int8)
>>> z2f = np.zeros(shape = (3,4), dtype = np.float32)
>>> z3  = np.zeros(shape = (3,4,5))

>>> z2i.ndim
2
>>> z2i.shape
(3, 4)
>>> z2i.size
12
>>> z2i.nbytes
12
>>> z2f.nbytes
48
```

Per modificarla la forma si può assegnare all'attributo `shape` o utilizzare il metodo `reshape` (ricordandosi di salvare)

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> a.shape = (2,2)
>>> a # C mode (row major) di default
array([[0, 1],
       [2, 3]])
```

```
>>> # uso di reshape, array a due dimensioni 4x1
>>> b = a.reshape((4,1))
>>> b
array([[0],
       [1],
       [2],
       [3]])

>>> # tornare ad un array a una dimensione di 4
>>> c = a.reshape((4,))
>>> c
array([0, 1, 2, 3])

>>> # o stessa cosa ma ancora più comodo
>>> d = a.reshape(-1)
>>> d
array([0, 1, 2, 3])
```

Oltre all'uso di `reshape` (che ritorna una vista sui dati e non copia se non necessario) vi sono altri due metodi per tornare ad una struttura dati unidimensionale ossia i metodi `ravel` (fa il flattening ma non produce una copia dei dati) e `flatten` (fa il flattening producendo sempre una copia).

```
>>> b.flatten()
array([0, 1, 2, 3])
>>> b.ravel()
array([0, 1, 2, 3])
```

Sebbene le due cose sembrino simili, `ravel` ritorna 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 perché non vi è copia di memoria ma bisogna essere più attenti con le modifiche.

12.2 Indexing

L'indexing serve per accedere in lettura e scrittura agli elementi di un array.

12.2.1 Indexing numerico

Strutture con 1 dimensione L'indexing per strutture ad una dimensione funziona similmente a quella dei dati builtin con `start:stop:step` aventi default, rispettivamente, 0, dimensione e 1. Ad esempio:

```
>>> x = np.arange(10)
>>> x[1]
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 in start:stop:step, si ha step negativo,
>>> # i default di start e stop sono swappati.
>>> # un modo conveniente per fare il reverse

>>> x[::-1]   # all elements, reversed
array([ 9,  1,  7,  6,  5,  4,  5,  0, -1,  0])
>>> x[5::-2]  # reversed every other from index 5
array([ 5,  5, -1])
```

Anche liste e array numerici sono ammessi come indice:

```
>>> select = [1, 2, 4, 5]
>>> x[select]
array([-1,  0,  4,  5])
>>> x[np.array(select)]
array([-1,  0,  4,  5])
```

Quando si usa un array come indice, la struttura (shape) ritornata dipende da quella dell'indice, non del dato indicizzato:

```
>>> x
array([ 0, -1,  0,  5,  4,  5,  6,  7,  1,  9])
>>> ind = np.array([[2, 1],
...                 [4, 7]])
>>> x[ind]
array([[ 0, -1],
       [ 4,  7]])
```

Strutture multidimensionali e broadcasting L'indexing di una struttura multidimensionale funziona fornendo per ogni dimensione, separata da virgola tra parentesi quadra, qualcosa che possa essere utilizzato come indice.

```
>>> # 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]
0

>>> # tridimensionale: [tabella, righe, colonne]
>>> y = np.arange(24).reshape(2,3,4)
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
```

```
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y[0,1,1]
5

>>> # elementi multipli con slicing e liste
>>> x[ [0,1], [1,2] ]
array([1, 6])
>>> x[ :2, 1: ]      # sino riga 2 esclusa, colonna 1 inclusa e sgg
array([[1, 2, 3],
       [5, 6, 7]])
>>> x[ ::-1, ::-1 ]
array([[11, 10,  9,  8],
       [ 7,  6,  5,  4],
       [ 3,  2,  1,  0]])

>>> # notare la differenza di dimensione dell'oggetto restituito
>>> # laddove c'è una slice, si tiene la dimensione
>>> # (se necessaria dipenderà poi dall'applicazione)
>>> x[2,  1:].shape    # array ad una dimensione
(3,)
>>> x[2:, 1:].shape    # array a due dimensioni
(1, 3)

>>> # uso di array
>>> row = np.array([0, 1, 2])
>>> col = np.array([2, 1, 3])
>>> x[row, col]
array([ 2,  5, 11])

>>> # indici possono anche essere misti (es lista e array),
>>> # anche con dati booleani introdotti in seguito
>>> x[row, [0,2,1]]
array([0, 6, 9])
```

L'accesso a determinate righe o colonne di un array può essere fatto combinando indici e slicing, utilizzando una slice vuota (`:`). La regola è: se un indice non è specificato si prendono tutti i dati su quella dimensione:

```
>>> # bidimensionale
>>> x[0, :] # prima riga
array([0, 1, 2, 3])
>>> # se si fornisce un solo indice alla struttura multidimensionale
>>> # viene interpretato nel primo asse/dimensione
>>> x[0] # equivalente a x[0, :]
array([0, 1, 2, 3])
>>> x[:, 0] # prima colonna
array([0, 4, 8])

>>> # tridimensionale
>>> y[0]         # prima tabella
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> y[0, :, :]   # same
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> y[:, 0, :]   # prima riga di tutte le tabelle
array([[ 0,  1,  2,  3],
       [12, 13, 14, 15]])
>>> y[1, 1]      # seconda riga della seconda tabella
array([16, 17, 18, 19])
```

Soffermiamoci sull'uso di array (e liste) per gli indici, nel caso multidimensionale. Funziona il *broadcasting* per l'accoppiamento degli indici:

```
>>> row
array([0, 1, 2])
>>> col
array([2, 1, 3])
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> x[row, col]
array([ 2,  5, 11])
```

In questa operazione abbiamo selezionato gli elementi `0,2`, `1,1` e `2,3` ordinatamente perché l'accoppiamento degli indici segue le regole del broadcasting. Se per esempio combinassimo un vettore colonna e uno riga tra gli indici, otterremmo un risultato/estrazione a due dimensioni

```
>>> rowt = row.reshape(3,1)
>>> x[rowt, col]
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

ogni valore di riga è matchato con ogni vettore colonna così come avviene nel broadcasting delle operazioni aritmetiche

```
>>> rowt * col
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

È importante ricordare che con l'indexing di liste e array la struttura ritornata riflette la shape degli indici post broadcast, non la shape dell'array indicizzato.

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 = [2,2,4,4,4,6,6,6,6,6,6,6]
>>> 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 cosi non è per cazzi suoi
>>> # (vedi vanderplas fancy indexing nel caso, ma anche chi se ne ciava)
```

12.2.2 Indexing logico (boolean masking)

Possiamo sfruttare il broadcasting (sez 12.5) per creare un array di booleani da utilizzare poi come indice per effettuare selezioni. Ad esempio:

```
>>> 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])
```

12.2.3 Subarray come viste

L'utilizzo dell'indexing mostra come gli array siano puntatori.
Una distinzione fondamentale rispetto alle liste è che lo slicing degli array presenta una vista dello stesso array; pertanto eventuali modifiche si ripercuoteranno sull'array/struttura di base indipendentemente da dove sono state effettuate.

```
>>> x = np.arange(10)
>>> x_slice = x[5:8]
>>> x_slice[2] = 123456
>>> x
array([     0,      1,      2,      3,      4,      5,      6, 123456,
            8,      9])

>>> # non si copiano dati ma si creano due nomi che
>>> # puntano alla stessa memoria qui
>>> y = x
```

```
>>> x[2] = -9999
>>> y
array([     0,      1,  -9999,      3,      4,      5,      6, 123456,
            8,      9])
>>> y[0] = -111
>>> x
array([  -111,      1,  -9999,      3,      4,      5,      6, 123456,
            8,      9])
```

Creare copie di array A volte è necessario creare copie dell'array invece che modificare l'originale. Per farlo utilizzare il metodo `copy`

```
>>> x = np.arange(9).reshape(3, 3)
>>> x_copy = x.copy()
>>> x_copy[0,0] = -999
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Indexing logico e copie Come eccezione, se si assegna una selezione di dati mediante array booleano ad un nuovo oggetto viene sempre fatta una copia (non è una vista come nella prossima sezione).

```
>>> b = np.arange(9).reshape(3, 3)
>>> c = b[b > 4]
>>> c[:] = 0
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

12.3 Elaborazioni di array

12.3.1 Concatenazione: `concatenate`, `vstack`, `hstack`

`numpy.concatenate` prende una tupla o una lista di array e li unisce

```
>>> # unidimensionale
>>> x = np.array([1, 2, 3])
>>> 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])

>>> # bidimensionale
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> 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]])
```

Soprattutto per lavorare con array di dimensioni miste, può essere `numpy.vstack` e `numpy.hstack`, funzioni di convenienza

```
>>> x = np.array([1, 2, 3])
>>> grid = np.array([[9, 8, 7],
...                  [6, 5, 4]])
>>> y = np.array([[99],
...               [99]])

>>> # vertically stack the arrays
>>> np.vstack([x, grid])
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
>>> # horizontally stack the arrays
>>> np.hstack([grid, y])
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Analogamente `np.dstack` effettuerà lo stack sulla terza dimensione.

12.3.2 Splitting: `split`, `vsplit`, `hsplit`

`split`, suddivide un array molteplici array. Si usano passando una lista di indici per indicare dove fare i tagli:

```
>>> # unidimensionale
>>> x = [1, 2, 3, 99, 99, 3, 2, 1]
>>> x1, x2, x3 = np.split(x, [3, 5])
>>> x1
array([1, 2, 3])
>>> x2
array([99, 99])
>>> x3
array([3, 2, 1])

>>> # bidimensionale
>>> x = np.arange(10).reshape(5,2)
>>> x
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

```
>>> x1, x2, x3 = np.split(x, [1, 3])
>>> x1
array([[0, 1]])
>>> x2
array([[2, 3],
       [4, 5]])
>>> x3
array([[6, 7],
       [8, 9]])
```

`vsplit` e `hsplit` sono funzioni di convenienza analoghe

```
>>> 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])
>>> 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])
>>> left
array([[9, 8],
       [6, 5]])
>>> right
array([[7],
       [4]])
```

Anche qui `numpy.dsplit` effettua il taglio sul terzo asse.

12.3.3 Ripetizione: `repeat`, `tile`

Per ripetere i singoli argomenti si usa `repeat`, per una struttura dati `tile`.

`repeat`

```
>>> x = np.array([1,2,3])
>>> x.repeat(2)
array([1, 1, 2, 2, 3, 3])
>>> x.repeat([3,2,1])
array([1, 1, 1, 2, 2, 3])

>>> # Array multidimensionali possono avere
>>> # ripetizione lungo un certo asse

>>> x = np.arange(4).reshape(2,2)
```

```
>>> x.repeat(2, axis = 0)
array([[0, 1],
       [0, 1],
       [2, 3],
       [2, 3]])
>>> x.repeat([2,3], axis = 1)
array([[0, 0, 1, 1, 1],
       [2, 2, 3, 3, 3]])
```

`tile` Il secondo argomento e il numero/struttura di copie: se intero viene effettuata una copia per riga, con una tuple si specifica la strtuura finale:

```
>>> x = np.arange(4).reshape(2,2)
>>> np.tile(x, 2)
array([[0, 1, 0, 1],
       [2, 3, 2, 3]])
>>> np.tile(x, (2,3))
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]])
```

12.3.4 Sorting: `sort`, `argsort`

sort Per ordinare un array `numpy` si usa il metodo `sort`. Nel caso di array multidimensionale, specificare `axis` per la dimensione di sorting:

```
>>> # unidimensional
>>> x = np.array([2,1,3,4])
>>> x.sort()
>>> x
array([1, 2, 3, 4])

>>> # bidimensional
>>> rand = np.random.RandomState(42)
>>> X = rand.randint(0, 10, (4, 6))
>>> # sorting di ogni colonna
>>> np.sort(X, axis=0)
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]])
>>> # sort di ogni riga
>>> np.sort(X, axis=1)
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]])
>>> # ultimi due casi si perdono eventuali relazioni
>>> # tra righe e colonne
```

`argsort` Se servono gli indici degli elementi riordinati 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])
```

12.3.5 Operazioni insiemistiche

Per effettuare operazioni di tipo insiemistico le funzioni sono le seguenti

```
-----------------------------------------------------
Funzione         Descrizione
-----------------------------------------------------
unique(x)          Compute the sorted, unique elements in x
intersect1d(x, y)  Compute the sorted, common elements in x and y
union1d(x, y)      Compute the sorted union of elements
in1d(x, y)         Compute a boolean array indicating whether each element of x
                   is contained in y
setdiff1d(x, y)    Set difference, elements in x that are not in y
setxor1d(x, y)     Set symmetric differences; elements that are in either of
                   the arrays, but not both
-----------------------------------------------------
```

12.4 Universal functions

12.4.1 Universal functions

Le universal functions, o *ufunctions*, sono funzioni (vanno chiamate come `np.ufunc()`) che vengono eseguite su tutti gli elementi di array in maniera vettorizzata. Ci sono funzioni:

- *unarie*: si applicano ad un array separatamente, le più importanti in tabella 12.2
- *binarie*: si applicando a più array (le più importanti in tab 12.3)
- altre ufuncs si trovano in `scipy.special`

12.4.2 Aritmetica vettorizzata

Anche le operazioni aritmetiche funzionano come *ufuncs*.

```
>>> x = np.arange(4)
>>> x
array([0, 1, 2, 3])
>>> -x
array([ 0, -1, -2, -3])
>>> x - 5
array([-5, -4, -3, -2])
```

```
>>> x * 2
array([0, 2, 4, 6])
>>> x ** 2
array([0, 1, 4, 9])
>>> x / 2
array([0. , 0.5, 1. , 1.5])
>>> x // 2
array([0, 0, 1, 1])
>>> x % 2
array([0, 1, 0, 1])
>>> -(0.5*x + 1) ** 2
array([-1.  , -2.25, -4.  , -6.25])
```

e sono di fatto un wrapper attorno alle seguenti ufuncs binarie, funzionanti grazie al broadcasting (sez. 12.5) `numpy`:

```
+    np.add          Addition (e.g., 1 + 1 = 2)
-    np.subtract     Subtraction (e.g., 3 - 2 = 1)
-    np.negative     Unary negation (e.g., -2)
*    np.multiply     Multiplication (e.g., 2 * 3 = 6)
/    np.divide       Division (e.g., 3 / 2 = 1.5)
//   np.floor_divide Floor division (e.g., 3 // 2 = 1)
**   np.power       Exponentiation (e.g., 2 ** 3 = 8)
%  np.mod       Modulus/remainder (e.g., 9 % 4 = 1)
```

Funzione	Descrizione
`abs`, `fabs`	absolute value
`sqrt`	square root
`exp`, `expm1`	esponenziale e $exp(x) - 1$
`log`, `log10`, `log2`, `log1p`	Natural log, log base 10, log base 2, and $log(1 + x)$, respectively
`sin`, `cos`, `tan`	trigonometriche
`arcsin`, `arccos`, `arctan`	trigonometriche inverse
`sign`	funzione segno: 1 se positivo, 0 se zero, o -1 se negative
`ceil`	the smallest integer greater than or equal to each element
`floor`	the largest integer less than or equal to each element
`rint`	Round elements to the nearest integer, preserving the dtype
`modf`	Return fractional and integral parts of array as separate array
`isnan`	Return boolean array indicating whether each value is NaN (Not a Number)
`isfinite`, `isinf`	each element is finite (non-inf , non-NaN) or infinite, respectively
`logical_not`	Compute truth value of not x element-wise. Equivalent to `-arr`
`any`	per array booleani, testa se alcuni sono veri
`all`	per array booleani, testa se tutti sono veri
`sum`	Sum of all the elements in the array or along an axis.
`prod`	Produttoria
`mean`	Arithmetic mean. Zero-length arrays have NaN mean.
`median`	Mediana
`percentile`	Percentile
`std`, `var`	Standard deviation and variance.
`min`, `max`	Minimum and maximum.
`argmin`, `argmax`	Indices of minimum and maximum elements, respectively.
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

Tabella 12.2: *ufuncs* unarie

Funzione	Descrizione
`maximum`, `fmax`	Element-wise maximum. `fmax` ignores `NaN`
`minimum`, `fmin`	Element-wise minimum. `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument

Tabella 12.3: *ufuncs* binarie

12.4.3 Aggregazione e prodotto cartesiano per ufuncs binarie

12.4.3.1 Aggregates

Per le ufuncs binarie (quindi ad esempio anche le operazioni matematiche) possiamo applicare:

- `reduce` applica una ufuncs agli elementi di un array sino a che un singolo risultato rimane
- `accumulate` fa l'operazione cumulata

```
>>> x = np.arange(1, 6)
>>> np.add.reduce(x)
15
>>> np.multiply.reduce(x)
120
>>> np.add.accumulate(x)
array([ 1,  3,  6, 10, 15])
```

Ora per questi esempi semplici vi sono funzioni specifiche (`sum`, `prod`), ma il macchinario funziona con ufuncs altre rispetto a `add` e `multiply`.

12.4.3.2 Outer products

Applicare una funzione al prodotto cartesiano degli elementi di due array si fa con `outer`, usato similmente alle aggregates

```
>>> x = np.arange(1, 6)
>>> np.multiply.outer(x, x)
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

12.4.4 Creazione di ufunctions

Per la creazione di funzioni vettorizzate possiamo utilizzare l'interfaccia C (modo più generale), scrivere funzioni in puro python o utilizzare compilatori LLVM con Numba (credo).

12.4.4.1 Pure python

`numpy.frompyfunc` prende in input una funzione, il numero di input e il numero di output. Le funzioni create restituiscono sempre array. `numpy.vectorize` è una alternativa (meno generale) che permette di specificare il tipo che ritorna la funzione

```
>>> # ufunc unaria (un input, un output)
>>> def add1_worker(x):
...     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)

>>> # ufunc binaria
>>> def add2_worker(x, y):
...     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)
```

12.4.4.2 L'uso di Numba

Mediante Numba si creano funzioni veloci mediante il progetto LLVM (che traduce codice python in codice macchina eseguibile da CPU o GPU). Studiare `https://wesmckinney.com/book/advanced-numpy.html#numpy_numba` quando numba sarà arrivato su python 3.11.

12.5 Broadcasting

Si e visto come le universal functions possano essere utilizzate per operare su array. Per array della stessa dimensionalità le operazioni vengono effettuate elemento per elemento

```
>>> x = np.array([0, 1, 2])
>>> y = np.array([5, 5, 5])
>>> x + y
array([5, 6, 7])
```

Nel caso si abbia a che fare con array di diverse dimensioni opera il *broadcasting*, ossia un set di regole per applicare ufuncs binarie ad array di dimensioni non uguale.

Esempi Ad esempio permette di aggiungere una costante (che può essere pensata come un array a zero dimensioni) ad un array (di una dimensione):

```
>>> x + 5
array([5, 6, 7])
```

Possiamo pensare a questa operazione come se il valore 5 venga duplicato nell'array `[5 5 5]` prima che si effettui la somma.
Analogamente avviene per array con dimensionalità maggiore

```
>>> M = np.ones((3, 3))
>>> M
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
>>> M + x
```

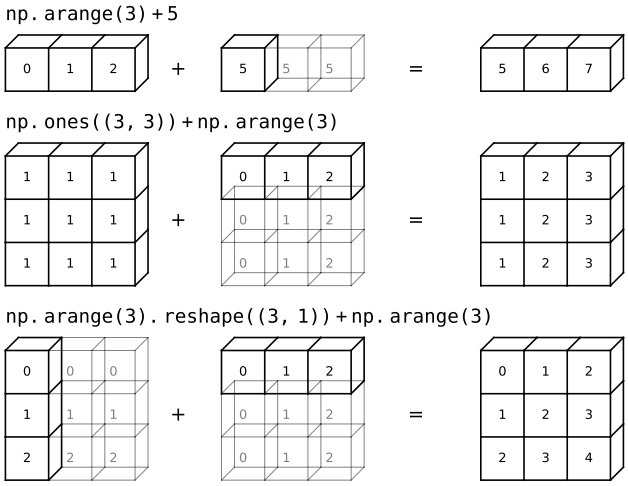

Figura 12.1: Broadcasting geometrics.

```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Qui l'array unidimensionale viene stretchato sulla seconda dimensione per matchare la forma di `M`.
Proviamo infine a sommare un array 1×3 con la sua trasposizione

```
>>> 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]])
```

In questo caso entrambi gli array sono stretchati fino a raggiungere una dimensione comune (ottenendo due matrici 3×3) dopodiché viene effettuata la somma. La geometria di questi esempi è visualizzata in figura 12.1; 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

grande[1]. 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. se la shape dei due array non matcha in una dimensione, l'array con shape 1 in quella dimensione viene stretchato per matchare l'altra dimensione;

3. se in qualsiasi dimensione le grandezze non sono uguali o una di esse è pari a 1, viene sollevato un errore

Esempio 1 Nel fare la somma di un array bidimensionale a uno monodimensionale

```
>>> M = np.ones((2, 3))
>>> a = np.arange(3)
>>> M
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> a
array([0, 1, 2])
```

si ha che le shape sono

```
>>> M.shape
(2, 3)
>>> a.shape
(3,)
```

Ora 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.]])
```

[1]Il 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).

Esempio 2 Vediamo un esempio dove entrambi gli array necessitano di broadcasting

```
>>> 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 3x3

Esempio 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,)
```

12.6 Confronto e array booleani

12.6.1 Confronto ed array booleani

Anche gli operatori `>`, `<`, `<=`, `>=`, `==`, `!=` sono implementati come ufuncs binarie e ritornano un array di booleani

```
>>> x = np.array([1, 2, 3, 4, 5])
>>> x < 3   # np.less
array([ True,  True, False, False, False])
>>> x > 3   # np.greater
array([False, False, False,  True,  True])
>>> x <= 3  # np.less_equal
array([ True,  True,  True, False, False])
>>> x >= 3  # np.greater_equal
array([False, False,  True,  True,  True])
>>> x != 3  # np.not_equal
array([ True,  True, False,  True,  True])
>>> x == 3  # np.equal
array([False, False,  True, False, False])

>>> # Confrontare due array elemento per elemento
>>> (x * 2) == (x ** 2)
array([False,  True, False, False, False])

>>> # anche su array multidimensionali
>>> rng = np.random.RandomState(0)
>>> x = rng.randint(10, size=(3, 4))
>>> x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
>>> x < 6
array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]])
```

12.6.2 Lavorare con array booleani

Vediamo un po' di applicazioni utili utilizzando

```
>>> rng = np.random.RandomState(0)
>>> x = rng.randint(10, size=(3, 4))
>>> x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

Contare gli elementi che rispettano un test Si fa analogamente a R, essendo `True` valorizzato come 1 e `False` 0

```
>>> np.sum(x < 6)              # conta overall
8
>>> np.sum(x < 6, axis = 1)    # conta per riga
array([4, 2, 2])
>>> np.sum(x > 6, axis = 0)    # conta per colonna
array([1, 1, 1, 0])
```

`any`, `all` Funzionano come in R

```
>>> np.any(x > 8)
True

>>> np.all(x < 10)
True
>>> np.all(x < 8, axis=1)
array([ True, False,  True])
```

Applicare operatori booleani Si applicando i seguenti

```
>>> x = np.array([True, False, True, False])
>>> y = np.array([True, True, False, False])
>>> 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])
```

Combinando operatori booleani per indexing (come in 12.2.2) e funzioni di aggregazione si possono effettuare statistiche per sottogruppi

```
>>> x = np.array([6,4,5,2,12])
>>> group = np.array(["a", "a", "b", "b", "a"])
>>> age = np.array([45,12,6,4,89])

>>> sel = x[(group == "a") & (age < 70)]
>>> np.median(sel)
5.0
```

La differenza con `and` e `or` `and` e `or` utilizzano l'oggetto nel complesso mentre gli operatori "bitwise" come sopra si riferiscono agli elementi che compongono l'oggetto. Con gli array non ci interessano `and` e `or`, da utilizzare con valori singoli.

12.6.3 Logica condizionale

La funzione `np.where` è la versione vettorizzata dell'espressione ternaria `x if condition else y` e permette ad esempio di fare cose del genere:

```
>>> a = np.arange(6)
>>> b = a * 2
>>> c = np.array([True, False] * 3)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> b
array([ 0,  2,  4,  6,  8, 10])
>>> c
array([ True, False,  True, False,  True, False])
>>> res = np.where(c, a, b)
>>> res
array([ 0,  2,  2,  6,  4, 10])
```

Un esempio di recoding

```
>>> a = np.arange(9).reshape(3,3)
>>> b = np.where(a > 5, 1, 0)
>>> b
array([[0, 0, 0],
       [0, 0, 0],
       [1, 1, 1]])
```

Per esprimere condizioni logiche più complicate si nestano diverse chiamate a `where`

```
>>> c = np.where(a > 6, 3, np.where(a > 3, 2, 1))
>>> c
array([[1, 1, 1],
       [1, 2, 2],
       [2, 3, 3]])
```

Oppure mediante algebra e logica pura

```
>>> d = (a >= 0) * 1 + (a > 3) * 1 + (a > 6) * 1
>>> d
array([[1, 1, 1],
       [1, 2, 2],
       [2, 3, 3]])
```

12.7 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] = "asdasdasd"
>>> y
array(['luca', 'bob', 'asdasdasd'], 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] = "asdasdasd"
>>> x
array(['apples', 'foobar', 'asdasdasd'], dtype=object)

>>> for i in range(3):
...     x[i] = x[i].capitalize()
...
>>> x
array(['Apples', 'Foobar', 'Asdasdasd'], dtype=object)

>>> # A questo array si può assegnare la qualunque ..
>>> x[1] = {1:2, 3:4}
>>> x
array(['Apples', {1: 2, 3: 4}, 'Asdasdasd'], dtype=object)
```

Capitolo 13

Pandas

`pandas` è il pacchetto di Python che fornisce strutture di dati e funzioni di utilità per l'analisi statistica standard.
Il modo standard di importarlo è

```
>>> import pandas as pd
```

Si usano spesso direttamente anche i suoi due oggetti principali quindi è spesso comodo fare quanto segue per le sessioni interattive (qui non lo sfruttiamo):

```
from pandas import Series, DataFrame
```

Spesso si usa assieme a `numpy`, che andiamo ad importare as well per il prosieguo

```
>>> import numpy as np
```

13.1 Strutture dati

Le strutture dati fornite sono

- `Series`: array unidimensionale omogeneo (di dimensione fissa) con labels; si può pensare come un `dict`;
- `DataFrame`: array a 2 dimensioni (variabili) con label, è un container di `Series`
- `Panel`: container di `DataFrame`

A queste si aggiungono altre strutture per gli indici:

- `Index`
- `MultiIndex`.

13.1.1 `Index`

L'`Index` è l'oggetto che fornisce i metadati necessari per `Series`, nonchè per righe e colonne del `DataFrame`. Alcune peculiarità:

- gli indici sono oggetti *immutabili* e non possono essere modificati dall'utente una volta creati;

- dall'immutabilità deriva il fatto che gli indici possono essere tranquillamente condivisi tra strutture di dati;
- si può usare `in` per testare la presenza di un indice in un oggetto `Index`;
- gli indici possono contenere doppi.

13.1.2 `Series`

`Series` è un array unidimensionale dotato (eventualmente) di etichette (index), che contiene oggetti dello stesso tipo. Il modo base per creare una `Series` è

```
x = pd.Series(data)
x = pd.Series(data, index = idx)
```

Dove `data` può essere lista, dict, array numpy o un valore scalare e volendo si può specificare con `index` le etichette (non necessario se data è dict).

13.1.2.1 Creazione

Gli elementi principali di una serie sono `array` (un `PandasArray`, oggetto che wrappa un array `numpy` più alcune aggiunte) ed `index` (oggetto `RangeIndex`):

```
>>> # serie senza indici
>>> x = pd.Series([4, 7, -5, 3])
>>> x
0    4
1    7
2   -5
3    3
dtype: int64
>>> x.array
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64
>>> x.index
RangeIndex(start=0, stop=4, step=1)

>>> # serie con indici
>>> a_dict = {"d": 4, "b": 1, "a":2, "c":3}
>>> y = pd.Series(a_dict)     # l'ordinamento è dettato dall'ordine del dict
>>> y                         # specificare index con ordinamento custom nel caso
d    4
b    1
a    2
c    3
dtype: int64
>>> y.index
Index(['d', 'b', 'a', 'c'], dtype='object')
```

Possiamo anche specificare gli indici in un secondo momento, assegnando direttamente

```
>>> x.index = ["asd", "foo", "bar", "baz"]
>>> x
asd    4
foo    7
bar   -5
baz    3
dtype: int64
```

Sia la serie che gli `index` hanno l'attributo `name` (senza `s` finale) che è sfruttato da pandas

```
>>> x = pd.Series({'luca': 23, 'andrea': 12, 'davide': 15})
>>> x.name = 'età_anni'
>>> x.index.name = 'giocatore'
>>> x
giocatore
luca      23
andrea    12
davide    15
Name: età_anni, dtype: int64
```

13.1.2.2 Indexing, quadre `.loc` e `.iloc`

Simile a numpy ma vi è la possibilità di usare gli index come in `dict`

```
>>> y
d    4
b    1
a    2
c    3
dtype: int64
>>> y[0]
4
>>> y[:3]
d    4
b    1
a    2
dtype: int64
>>> y[y > y.median()]
d    4
c    3
dtype: int64
>>> y[[3, 0, 1]]
c    3
d    4
b    1
dtype: int64

>>> # Uso degli indici
>>> y["a"]          # restituito un valore
2
```

```
>>> y[["b", "d"]]  # restituita una Serie
b    1
d    4
dtype: int64
```

Slicing con label/index si comporta differentemente dallo slicing standard: gli estremi inclusi

```
>>> y['d':'a']
d    4
b    1
a    2
dtype: int64
```

L'indexing mediante quadre è flessibile/liberale (ad esempio è possibile scegliere mediante interi elementi indicizzati da una stringa. Se invece si desidera una selezione più restrittiva utilizzare gli attributi `loc` e `iloc`; il primo utilizza l'indice fornito, che deve essere stringa o simile, in maniera restrittiva (es se l'array è indicizzato con interi non funziona). Vi è anche un `iloc` che prende in input solo interi (ma può selezionare anche array indicizzati da stringhe). In vari punti della doc di pandas, `loc` ed `iloc` sono i metodi consigliati per indexing.

13.1.2.3 Modifica

Funzionante mediante indici e broadcasting

```
>>> y
d    4
b    1
a    2
c    3
dtype: int64

>>> # assegnazione
>>> y[0] = 2
>>> y["a"] = 5
>>> y
d    2
b    1
a    5
c    3
dtype: int64
>>> y[:] = 0   # y = 0 sbagliato, cambia proprio l'oggetto
>>> y
d    0
b    0
a    0
c    0
dtype: int64
```

13.1.2.4 Eliminazione elementi

Si usa il metodo `drop` per una modifica temporanea (drop ritorna l'oggetto modificato) o `del` per una definitiva

```
>>> x = pd.Series([1,2,3], index = ['a', 'b', 'c'])
>>> x.drop('b') # un solo elemento
a    1
c    3
dtype: int64
>>> x
a    1
b    2
c    3
dtype: int64
>>> x.drop(['b', 'c']) # più elementi
a    1
dtype: int64


>>> ## rimozione definitiva
>>> del x['b']
>>> x
a    1
c    3
dtype: int64
```

13.1.2.5 Coercizione di tipo

Per cambiare tipo a una serie si usa `astype` fornendo una stringa, un `numpy.dtype`, un `pandas.ExtensionDtype` (vedi sezione 13.1.2.6 o un tipo builtin di Python.

```
>>> a = pd.Series(["1", "2", "3"])
>>> b = a.astype(int)
>>> b
0    1
1    2
2    3
dtype: int64
>>> c = b.astype("float")
>>> c
0    1.0
1    2.0
2    3.0
dtype: float64
```

13.1.2.6 `dtype` classici e nuovi

Nella definizione se non si specifica `dtype` nella chiamata pandas inferisce dal contesto

```
>>> x = pd.Series([0,1,2, np.nan])
>>> x.dtype
dtype('float64')
```

Storicamente il costruire sui tipi numerici di `numpy` ha portato alcune difficoltà:

- interi e booleani gestivano male i dati mancanti, motivo per cui venivano convertiti ad interi
- dataset con stringhe occupavano molta memoria (non essendoci l'equivalente di un dato categorico tipo il factor di R)
- alcuni tipi di dati (es intervalli di tempo etc) erano difficili da supportare in maniera efficiente

Per questo pandas ha sviluppato un set di propri tipi (e un modo per aggiungere tipi facilmente) che possono essere specificati in sede di chiamata (parametro `dtype`) o coercizione (metodo `astype`).
Uno, `categorical` lo si è visto mediante `cut`. Vediamo il caso precedente in cui inseriamo un dato mancante e il tutto non è convertito a float, ma viene stampato anche l'`NA` di nuova generazione:

```
>>> x = pd.Series([0,1,2, np.nan], dtype = "Int32")
>>> x
0       0
1       1
2       2
3    <NA>
dtype: Int32

>>> x.astype("boolean")
0    False
1     True
2     True
3     <NA>
dtype: boolean
```

La lista dei tipi di nuova generazione (stringa da utilizzare per indicare il tipo, classe pandas e descrizione) è riportata in tabella 13.1.

13.1.2.7 `Categorical`

È una rappresentazione a-la `factor` con interi linkati a label per risparmiare spazio rispetto alle stringhe secche. Si possono creare mediante `pd.Categorical` oppure attraverso series specificando `category` come `dtype`.

```
>>> # creazione e attributi principale
>>> c = pd.Series(list("abbccc"), dtype = 'category')


>>> # Test memoria
>>> raw = pd.Series(['apple', 'orange', 'apple', 'apple'] * 2)
>>> c = raw.astype('category')
```

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
`Int34`	`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 13.1: Tipi estesi di pandas

```
>>> from sys import getsizeof
>>> getsizeof(raw)
662
>>> getsizeof(c)
405
```

Se si vuole controllare l'ordinamento dei livelli (non lasciando all'ordinamento lessicografico) bisogna creare una istanza di `CategoricalDtype`

```
>>> from pandas.api.types import CategoricalDtype

>>> cat_type = CategoricalDtype(categories = ["b", "c", "d"],
...                             ordered = False)

>>> c = pd.Series(list("abbccdda"), dtype = cat_type)
>>> c
0    NaN
1      b
2      b
3      c
4      c
5      d
6      d
7    NaN
dtype: category
Categories (3, object): ['b', 'c', 'd']
```

Similmente a `Series.str` per l'accesso a metodi per stringhe si ha `Series.cat` per l'accesso a info/metodi categorici ad esempio l'autocompletamento ci suggerisce

```
c.cat.codes      # lista la memorizzazione numerica
c.cat.categories # lista le categorie
c.cat.ordered    # booleana autoesplicativa
```

```
c.cat.as_ordered          # metodo trasforma in ordered
c.cat.as_unordered        # metodo trasforma in unordered

c.cat.set_categories      # metodo imposta le categorie ad una nuova lista
c.cat.rename_categories   # rinomina
c.cat.reorder_categories  # cambia ordinamento

c.cat.add_categories              # aggiunge categorie alla fine della lista
c.cat.remove_categories           # toglie categorie creando mancanti
c.cat.remove_unused_categories    # toglie categorie con zero frequenze
```

13.1.2.8 Test appartenenza di elemento: `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
0       0
1       1
2       2
3    <NA>
dtype: Int32

>>> # uso di indici
>>> 'a' in x
False
>>> 'b' in x
False

>>> # uso di valori
>>> x.isin([1, 2])
0    False
1     True
2     True
3    False
dtype: boolean
```

13.1.2.9 Dati mancanti

Usiamo `np.nan` per indicare dati mancanti[1]. È considerato `NA` anche il `None` builtin di Python Vediamone 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
0    1.0
1    2.0
2    NaN
```

[1] Con 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.

```
3    3.0
4    NaN
dtype: float64

>>> # test
>>> z.isna()  # equivalentemente pd.isna(z)
0    False
1    False
2     True
3    False
4     True
dtype: bool
>>> z.notna() # negazione del precedente, equivale a pd.notna(z)
0     True
1     True
2    False
3     True
4    False
dtype: bool

>>> # utility
>>> z.dropna()   # filtrare
0    1.0
1    2.0
3    3.0
dtype: float64
>>> z.fillna(-9) # riempire
0    1.0
1    2.0
2   -9.0
3    3.0
4   -9.0
dtype: float64
```

13.1.2.10 Gestione duplicati

A livello di `Series`:

```
>>> x = pd.Series([1, 2, 2, 3, 3, 3])

>>> # utility
>>> x.duplicated()       # marca i doppi
0    False
1    False
2     True
3    False
4     True
5     True
dtype: bool
>>> x.unique()           # rendere unica ma restituisce ndarray
```

```
array([1, 2, 3])
>>> x.drop_duplicates()  # rende unica e restituisce una Series
0    1
1    2
3    3
dtype: int64
```

13.1.2.11 Elaborazione e allineamento indici

Le elaborazioni su un vettore sono simili a numpy (quindi si applicano *ufuncs* e broadcasting); viene preservato l'indice:

```
>>> x
0    1
1    2
2    2
3    3
4    3
5    3
dtype: int64
>>> np.exp(x) + 1
0     3.718282
1     8.389056
2     8.389056
3    21.085537
4    21.085537
5    21.085537
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`.

13.1.2.12 Reindexing

Consiste nel creare un nuovo oggetto, caratterizzato da un set di indici diverso, a partire da uno vecchio. 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

>>> # viene effettivamente creata una copia
>>> y[1] = 2
>>> x
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

13.1.2.13 Applicare funzioni per singolo elemento: `map`

Per applicare una funzione per singolo elemento (es builtin Python) a tutti gli elementi di una `Series` utilizzare `map`. Ad esempio per una 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
```

13.1.2.14 Effettuare recode: `map` e `replace`

`map` accetta anche un dict e nel caso 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
```

13.1.2.15 Sorting

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
d    1
a    3
b    2
c    4
dtype: int64
>>> 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
```

13.1.2.16 Discretizzazione/creazione di classi

Si usa la funzione `cut` sulla `Series` (che ha la peculiarità di restituire un oggetto `Categorical` che presenta notevoli somiglianze coi factor)

```
>>> 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 = [0, 25, 50, 75, 100, 125]
>>> labs = ["giovane", "medio", "anziano", "vecchio", "vetusto"]
>>> agecl = pd.cut(age, bins = cuts, labels = labs)
>>> agecl
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']
```

Se si fornisce un intero a `cut` crea un numero equivalente di categorie equispaziate tra minimo e massimo; `qcut` invece effettua un `cut` sulla base dei quantili.

13.1.2.17 Statistiche descrittive

Per le frequenze si utilizza il metodo `value_counts`

```
>>> agecl.value_counts()
medio      4
vecchio    3
giovane    2
anziano    1
vetusto    0
Name: count, dtype: int64
```

13.1.2.18 Ottenere dummy variables

Si usa la funzione `get_dummies`, che restituisce un `DataFrame`

```
>>> 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 il metodo `str.getdummies` delle `Series`, utile per creare dummy a partire da una Series specificando 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 medainte specificando `prefix`.

13.1.2.19 `MultiIndex`, indexing gerarchico nelle serie, reshape

L'indexing gerarchico (implementato mediante la classe `MultiIndex`) è una feature di pandas che permette di gestire dati multidimensionali, riconducendoli ad una tabella a due dimensioni. È importante nel *reshape* dei dati (funzioni `stack`/`unstack`) e nelle operazioni basate su gruppo (formare pivot table). Qui vediamo il multiindexing nelle `Series`, lasciamo quello dei `DataFrame` per più tardi

```
>>> 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)],
           )
```

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
```

Un esempio di reshape con una `Series` e `MultiIndex`, che diviene un `DataFrame`

```
>>> 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       0.206587
    2       0.124627
    3       0.791500
b   1       0.657671
    3       0.222644
c   1       0.662914
    2       0.438563
d   2       0.510118
    3       0.561622
dtype: float64
>>> df2 = df.unstack() # wide
>>> df3 = df2.stack()  # tornare a long
```

13.1.3 DataFrame

È una 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` che condividono lo stesso indice.

13.1.3.1 Definizione e attributi

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, mediante dict
>>> 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"))

>>> # 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
```

Gli attributi principali sono `shape`, `index`, `columns`, `values` e i loro `name`:

```
>>> 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 sono 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
```

13.1.3.2 Accedere a colonne e righe

Il subsetting avviene mediante le quadre e gli attributi `loc` e `iloc`. Direi che il meglio sia:

- abituarsi ad utilizzare `loc`
- specificare i criteri (per riga e colonne) separati da virgole tra parentesi;
- se su riga o colonna si prende tutto, specificare una slice vuota `:`;
- per avere un equivalente R sugli indici di riga (che partano da 1, ma da valutare sta cosa ..) si può specificare nella definizione del `DataFrame`

  ```
  index = np.arange(5) + 1
  ```

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, index = list("abcdef")) # mettiamo indice stringa..

>>> # colonne: indice stringa viene interpretato come colonna
>>> df.state    # singola colonna, meglio per l'uso interattivo
a      Ohio
b      Ohio
c      Ohio
d    Nevada
e    Nevada
f    Nevada
Name: state, dtype: object
>>> df["state"] # singola colonna, meglio per la programmazione
a      Ohio
b      Ohio
c      Ohio
d    Nevada
e    Nevada
f    Nevada
Name: state, dtype: object
>>> df[["state", "pop"]] # due colonne
    state  pop
a    Ohio  1.5
b    Ohio  1.7
c    Ohio  3.6
d  Nevada  2.4
```

```
e  Nevada  2.9
f  Nevada  3.2

>>> # righe 1: indici logici e numerici (slices) vengono interpretati
>>> # di riga
>>> df[df.state == 'Ohio']     # logici
  state  year  pop
a  Ohio  2000  1.5
b  Ohio  2001  1.7
c  Ohio  2002  3.6
>>> df[:2]                     # slice
  state  year  pop
a  Ohio  2000  1.5
b  Ohio  2001  1.7

>>> # righe 2: mediante loc e iloc un solo indice interpretato di riga
>>> df.loc['a'] # indice stringa
state    Ohio
year     2000
pop       1.5
Name: a, dtype: object
>>> df.iloc[3]  # numerico con iloc
state    Nevada
year       2001
pop         2.4
Name: d, dtype: object

>>> # righe e colonne
>>> # df[df.state == 'Ohio', "pop"]   # questo R style non funziona ma
>>> df.loc[df.state == 'Ohio', "pop"] # righe su logico e scelta colonne
a    1.5
b    1.7
c    3.6
Name: pop, dtype: float64
>>> df.loc['a', 'pop']              # uso di nomi di riga e colonna
1.5
>>> df.loc['a',['pop', 'year']]     # più righe di una colonna
pop      1.5
year    2000
Name: a, dtype: object
>>> df.iloc[3, 1]                   # solo indici numerici in [] di iloc
2001
>>> df.iloc[3, [1,2]]               # selezionare piu colonne con iloc
year    2001
pop      2.4
Name: d, dtype: object
>>> df.iloc[3]['year']              # numero di riga e nome colonna
2001
>>> # Quest'ultima sintassi ][ meglio evitare direi (definire indici di riga numerici
```

```
>>> # sia loc che iloc funzionano con slice
>>> df.loc[:"d", "year":"pop"]
   year  pop
a  2000  1.5
b  2001  1.7
c  2002  3.6
d  2001  2.4
>>> df.iloc[:, :3]
    state  year  pop
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

>>> # array booleani utilizzabili con loc, non con iloc
>>> df.loc[df.year==2001]
    state  year  pop
b    Ohio  2001  1.7
d  Nevada  2001  2.4
```

13.1.3.3 Creare e modificare colonne

Le colonne possono esser create mediante assegnazione a nomi colonna inesistenti (di default le colonne vengono inserite alla fine, usare il metodo `insert` se si desidera altrimento). Possono essere modificate mediante assegnazione a nomi esistenti:

```
>>> ## ------------------------------------------------
>>> ## Creazione colonna: torna comoda la sintassi df['variabile']
>>> ## focalizziamoci qui sul valore a destra dell'=
>>> ## ------------------------------------------------

>>> # su valore puntuale funziona broadcasting
>>> df['constant'] = 3
>>> # è possibile fare cose anche del tipo
>>> df[['foo', 'bar']] = [0, 1]

>>> # lista (diciamo) e array numpy: lunghezza deve matchare
>>> df['list'] = range(6)
>>> rng = np.random.default_rng(4515)
>>> df['nparray']  = rng.random(6)

>>> # assegnazione di una pd.Series rispetta gli indici
>>> df['series']  = pd.Series([10,20,30,40], index = ['a', 'x', 'y', 'z'])
>>> df
    state  year  pop  constant  foo  bar  list   nparray  series
a    Ohio  2000  1.5         3    0    1     0  0.511587    10.0
b    Ohio  2001  1.7         3    0    1     1  0.071141     NaN
```

```
c    Ohio  2002  3.6         3    0    1     2  0.573725     NaN
d  Nevada  2001  2.4         3    0    1     3  0.885854     NaN
e  Nevada  2002  2.9         3    0    1     4  0.942855     NaN
f  Nevada  2003  3.2         3    0    1     5  0.875534     NaN


>>> # ------------------------------------------------------------
>>> # Modifica a parti specifiche meglio utilizzare loc o iloc
>>> # ------------------------------------------------------------

>>> # punto specifico
>>> df.loc["a", "constant"] = 5

>>> # riga con indice numerico usare iloc (evitiamo
>>> # di scrivere sulla prima colonna se no state diventa
>>> # di dtype "object" ossia generico e non più stringa)
>>> df.iloc[1, 1:] = 2
>>> df
    state  year  pop  constant  foo  bar  list   nparray  series
a    Ohio  2000  1.5         5    0    1     0  0.511587    10.0
b    Ohio     2  2.0         2    2    2     2  2.000000     2.0
c    Ohio  2002  3.6         3    0    1     2  0.573725     NaN
d  Nevada  2001  2.4         3    0    1     3  0.885854     NaN
e  Nevada  2002  2.9         3    0    1     4  0.942855     NaN
f  Nevada  2003  3.2         3    0    1     5  0.875534     NaN
```

13.1.3.4 Eliminare colonne e righe

Per le colonne in maniera definitiva si usa `del`

```
>>> df
    state  year  pop  constant  foo  bar  list   nparray  series
a    Ohio  2000  1.5         5    0    1     0  0.511587    10.0
b    Ohio     2  2.0         2    2    2     2  2.000000     2.0
c    Ohio  2002  3.6         3    0    1     2  0.573725     NaN
d  Nevada  2001  2.4         3    0    1     3  0.885854     NaN
e  Nevada  2002  2.9         3    0    1     4  0.942855     NaN
f  Nevada  2003  3.2         3    0    1     5  0.875534     NaN
>>> del df['constant']
>>> df
    state  year  pop  foo  bar  list   nparray  series
a    Ohio  2000  1.5    0    1     0  0.511587    10.0
b    Ohio     2  2.0    2    2     2  2.000000     2.0
c    Ohio  2002  3.6    0    1     2  0.573725     NaN
d  Nevada  2001  2.4    0    1     3  0.885854     NaN
e  Nevada  2002  2.9    0    1     4  0.942855     NaN
f  Nevada  2003  3.2    0    1     5  0.875534     NaN
```

Altrimenti per righe e colonne si usa sempre il metodo `drop`, specificando tra parentesi gli assi (ma avendo cura di assegnare il risultato):

```
>>> df.drop(index = 'a')
    state  year  pop  foo  bar  list   nparray  series
b    Ohio     2  2.0    2    2     2  2.000000     2.0
c    Ohio  2002  3.6    0    1     2  0.573725     NaN
d  Nevada  2001  2.4    0    1     3  0.885854     NaN
e  Nevada  2002  2.9    0    1     4  0.942855     NaN
f  Nevada  2003  3.2    0    1     5  0.875534     NaN
>>> df.drop(columns = ['year', 'series'])
    state  pop  foo  bar  list   nparray
a    Ohio  1.5    0    1     0  0.511587
b    Ohio  2.0    2    2     2  2.000000
c    Ohio  3.6    0    1     2  0.573725
d  Nevada  2.4    0    1     3  0.885854
e  Nevada  2.9    0    1     4  0.942855
f  Nevada  3.2    0    1     5  0.875534
>>> df
    state  year  pop  foo  bar  list   nparray  series
a    Ohio  2000  1.5    0    1     0  0.511587    10.0
b    Ohio     2  2.0    2    2     2  2.000000     2.0
c    Ohio  2002  3.6    0    1     2  0.573725     NaN
d  Nevada  2001  2.4    0    1     3  0.885854     NaN
e  Nevada  2002  2.9    0    1     4  0.942855     NaN
f  Nevada  2003  3.2    0    1     5  0.875534     NaN
```

13.1.3.5 Rinominare colonne/indici

Si usa il metodo `rename`, che può prendere dict e funzioni e applicarle a righe o colonne

```
>>> # utilizzo di un dict sulle colonne, str.upper sugli index/righe
>>> ft = {'foo': 'new1', 'bar': 'new2'}
>>> df = df.rename(index = str.upper, columns = ft)
>>> df
    state  year  pop  new1  new2  list   nparray  series
A    Ohio  2000  1.5     0     1     0  0.511587    10.0
B    Ohio     2  2.0     2     2     2  2.000000     2.0
C    Ohio  2002  3.6     0     1     2  0.573725     NaN
D  Nevada  2001  2.4     0     1     3  0.885854     NaN
E  Nevada  2002  2.9     0     1     4  0.942855     NaN
F  Nevada  2003  3.2     0     1     5  0.875534     NaN
```

13.1.3.6 Reindexing

Nei data frame serve per selezionare/ordinare righe e colonne.

```
>>> 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

>>> # reindexing di riga, creato missing perché b
>>> # non disponibile nei dati di partenza
>>> df2 = df.reindex(['a', 'b', 'c', 'd'])
>>> df2
   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

>>> # reindexing di colonna, utilizzare columns
>>> # qui si cancella Ohio, non richiesto
>>> states = ['Texas', 'Utah', 'California']
>>> df.reindex(columns = states)
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

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)

```
>>> df.loc[["a", "d", "c"], ["California", "Texas"]]
   California  Texas
a           2      1
d           8      7
c           5      4
```

13.1.3.7 `MultiIndex` e `DataFrame`

Con un dataframe, 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
```

13.1.3.8 Creare indici a partire dai dati e viceversa

Se si vuol creare gli indici di riga a partire da una/più colonne di un `DataFrame` si usa `set_index` (specificando tra parentesi la variabile/lista di variabili):

```
>>> 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
>>> df2 = df.set_index(['c', 'd'])
>>> df2
       a  b
c   d
```

```
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

Per non eliminare le colonne specificate come index si usa il parametro `drop = False`.
Viceversa per creare colonne a partire dagli indici si usa `reset_index`

```
>>> 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
```

13.2 Data management con `DataFrame`

13.2.1 Sorting

Se si vuole ordinare una dataframe sulla base degli indici (di riga) o nomi colonna si usa il metodo `sort_index`, per i valori `sort_values`

```
>>> 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")
... )
>>> x
0      a|b
1      b|c
2        a
3    a|b|c
dtype: object

>>> # ordinare le righe per indice
>>> 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 il DataFrame per il valore assunto da uno o più
```

```
>>> 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
```

Per ordinare le colonne per nome variabile, sempre:

```
>>> # ordinare le colonne 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
```

13.2.2 Applicazione di funzioni

13.2.2.1 A righe e colonne

Per applicare una funzione ad ogni riga o colonna di un DataFrame si può utilizzare il metodo `apply`. Un esempio:

```
>>> df = pd.DataFrame({"a" : [1, 2, 3], "b" : [4, 5, 6]})
>>> df
   a  b
0  1  4
1  2  5
2  3  6

>>> # funzione che collassa ad un numero
>>> def f(x):
...     return(x.mean())
...
>>> # applica a colonne di default
>>> df.apply(f)
a    2.0
b    5.0
dtype: float64
>>> # applica a riga (intende effettua la funzione
>>> # ciclando su 1, le colonne)
>>> df.apply(f, axis = 1)
0    2.5
1    3.5
2    4.5
```

```
dtype: float64

>>> # funzione che restituisce una serie (esempio banale ..)
>>> 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
```

13.2.2.2 Funzioni element-wise ad una colonna

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]})
>>> def gt2(x):
...     return x > 2
...
>>> # mapping di funzione
>>> df["bgt2"] = df["b"].map(gt2)

>>> # 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["brecoded"] = df.b.replace(ft)

>>> df
   a  b   bgt2 agroup  brecoded
0  a  1  False    a-b         1
1  b  2  False    a-b         2
2  c  3   True    c-d         3
3  d  4   True    c-d         3
```

13.2.2.3 Funzioni element-wise a tutto il `DataFrame`

Si utilizza il metodo `applymap` (che applica il metodo `map` delle Series a tutte le righe/colonne)

```
>>> rng = np.random.default_rng(665)
>>> df = pd.DataFrame(rng.standard_normal((2, 2)))

>>> def sign(x):
...     return -1 if x < 0 else 1
...
>>> df.applymap(sign)
   0  1
0  1 -1
1 -1 -1
```

Analogamente al caso delle `Series` accetta anche un dict per *recode* (con tutti i casi specificati).

13.2.3 Merge

Si effettua mediante la funzione `merge`

Sulla base di variabili Il merge funziona di default sulla base delle colonne presenti in entrambi i dataset, qui `key`. Altriment specificare i parametri `on` (se i due dataframe hanno variabili di merge con lo stesso nome) oppure ordinatamente `left_on` e `right_on`.
Se non si specifica nulla ritorna le righe dove la chiave è presente in entrambi i dataset (*inner join*):

```
>>> 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)
  key  data1 data2
0   a      0     x
1   b      1     y
2   b      2     y
```

Se si vogliono altri tipi di merge rispetto l'inner si possono specificare nel parametro `how` (es `left`, `right` ed `outer`)

```
>>> # tiene tutto il dataset di sx e lo integra con quello di dx
>>> pd.merge(df1, df2, how = 'left')
  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
>>> # tiene tutto il dataset di sx e lo integra con quello di dx
>>> pd.merge(df1, df2, how = 'right')
  key  data1 data2
0   a    0.0     x
1   b    1.0     y
2   b    2.0     y
3   d    NaN     z
>>> # outer tiene tutto
>>> pd.merge(df1, df2, how = 'outer')
  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
```

Sulla base di indici Se si desidera che la chiave di merge siano gli indici (di riga) dei dataframe bisogna passare `left_index = True` o `right_index = True` per fare il merge.

13.2.4 Binding (concatenating)

Si effettua mediante la funzione `concat`, alla doc della quale si rimanda per altro:

```
>>> 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
```

13.2.5 Dati mancanti

Nel caso di `DataFrame` si potrebbe voler effettuare diverse operazioni. Anche qui 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

>>> # eliminare le righe che ...
>>> df.dropna()             # ... contengono anche un solo NA
     0    1    2
0  1.0  6.5  3.0
>>> df.dropna(how = 'all') # ... sono tutte NA
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0

>>> # eliminare le colonne che ...
>>> df[4] = np.nan
>>> df.dropna(axis = "columns", how = "all") # contengono tutti 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
```

Per riempire i dati mancanti invece si usa `fillna`.

13.2.6 Test di appartenenza: `in`, `isin`

`in` applicato ad un DataFrame testa la presenza di una colonna avente un determinato nome, `isin`

```
>>> df = pd.DataFrame({"a": [1, 2, 3],
...                    "b": list("xyz"),
...                    "c": [2,3,4]},
...                    index = list("lmn"))
>>> df
   a  b  c
```

```
l  1  x  2
m  2  y  3
n  3  z  4

>>> # in per nomi colonna
>>> 'a' in df
True
>>> 'l' in df
False

>>> # isin per valori
>>> df.isin([2, "z"])  # lista: cerca in tutte le colonne
       a      b      c
l  False  False   True
m   True  False  False
n  False   True  False
>>> ~df.isin(["z"])    # come negare una ricerca
      a      b     c
l  True   True  True
m  True   True  True
n  True  False  True
>>> df.isin({"c" : [3]})  # dict: cerca in alcune colonne specificate
       a      b      c
l  False  False  False
m  False  False   True
n  False  False  False

>>> # se si forniscono una Series o un DataFrame
>>> # anche gli indici devono matchare .. esempio
>>> # eventualmente da fare
```

13.2.7 Gestione duplicati

A livello di `DataFrame`:

- il metodo `duplicated` ritorna un vettore di booleani per indicare se una riga è duplicata di un altra
- `drop_duplicates` ritorna un data frame senza duplicati.

Alcune opzioni:

- questi metodi utilizzano tutte le colonne disponibili: alternativamente specificare una lista di colonne come parametro `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]})
```

```
>>> df
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4

>>> # ricerca ed eliminazione complessiva
>>> df.duplicated()
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
>>> df.drop_duplicates()
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4

>>> # cercare duplicati solo sulla base di una colonna
>>> df["v1"] = range(7)
>>> df
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
5  two   4   5
6  two   4   6
>>> df.drop_duplicates(subset = ["k1"])
    k1  k2  v1
0  one   1   0
1  two   1   1

>>> # tenere gli ultimi elementi, non i primi
>>> df.drop_duplicates(subset = ["k1"], keep = 'last')
    k1  k2  v1
4  one   3   4
6  two   4   6
```

13.2.8 Reshape

13.2.8.1 Mediante indexing e 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 in particolar modo può esser comodo dare un nome agli indici. In seguito un esempio di base.

```
>>> df = pd.DataFrame(dict(
...     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

>>> # porre in formato long: uso di stack
>>> df.stack()
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(dropna = False)
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()

>>> # porre in formato wide: uso di unstack
>>> df_long.unstack() # default: level = 1 (seconda colonna di indici)
```

```
          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)  # fare l'unstack usando la prima
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') # stessa cosa ma sfruttando nomi
state  Ohio  Colorado
one     0.0       3.0
two     1.0       NaN
three   2.0       5.0
four    NaN       6.0
```

13.2.8.2 Mediante pivot e melt

Porre in wide Quando 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
```

13.2.9 Bootstrap, permutazioni, subsampling

Per generare gli indici da usare nell'estrazione si usa `np.random.integers`, per applicarli `iloc`. Un esempio didattico a seguire. Alternativamente si può utilizzare il metodo `sample` dei `DataFrame` che ammette l'opzione `replace`

```
>>> rng = np.random.default_rng(6235)

>>> df = pd.DataFrame(dict(
...     x = list("abcdef"),
...     y = rng.standard_normal(6)
... ))

>>> nrows = df.shape[0]
>>> boot_reps = 5
>>> rep_n = 10

>>> # lista di indici per ciascuna ripetizione bootstrap
```

```
>>> indexes = []
>>> for idrep in range(boot_reps):
...     indexes.append(rng.integers(6, size = rep_n))
...
>>> # dare un occhio
>>> indexes
[array([4, 0, 5, 0, 5, 3, 0, 5, 1, 3]), array([3, 4, 3, 5, 5, 1, 3, 5, 2, 3]), array([2, 0, 1,

>>> # lista di dataframe con le estrazioni
>>> dfs = []
>>> for i in indexes:
...     dfs.append(df.iloc[i])
...
>>> # statistica (media di y)
>>> def boot_f(df):
...     return df.y.mean()
...
>>> res = []
>>> for x in dfs:
...     res.append(boot_f(x))
...
>>> res
[0.29995932166929806, 0.576792941726796, 0.38232247814089265, -0.13974435347512554, 0.554319863
```

13.3 Statistica descrittiva

13.3.1 Univariata

Per le statistiche descrittive per un `DataFrame` ci sono le funzioni riportate in tabella 13.2.

13.3.2 Bivariata

I metodi `corr` e `cov` permettono di ottenere la correlazione e covarianza per gli elementi di un dataframe.

13.3.3 Stratificata

Si applica il classico split-apply-combine ossia:

1. si splitta la struttura dati (serie o data frame) sulla base di chiavi fornite; lo splitting è effettuato su un particolare asse;
2. viene applicata una funzione ad ogni chunk di dati;
3. viene riassemblato il tutto: la forma dell'oggetto di ritorno dipende dalle elaborazioni fatte al secondo step.

13.3.3.1 Splitting (grouping)

Per effettuare lo splitting si usa `groupby` alla quale si può fornire alternativamente mediante liste/array, dict/series che forniscono il mapping o funzione

Metodo	Descrizione
`count`	numero di dati non mancanti
`describe`	set di statistiche descrittive per ciascuna colonna
`min`, `max`	minimo e massimo
`argmin`, `argmax`	indici (locations, interi) dove si trovano minimo/massimo rispettivamente
`idxmin`, `idxmax`	indici di massimo o minimo
`quantile`	quantili
`sum`	somma dei valori
`mean`	media
`median`	mediana
`mad`	MAD
`prod`	Prodotto di tutti i valori
`var`	Varianza campionaria
`std`	Deviazione standard campionaria
`skew`	Skewness campionaria
`kurt`	Kurtosis campionaria
`cumsum`	somma cumulata
`cummin`, `cummax`	minimo e massimo cumulato
`cumprod`	produttoria cumulata
`diff`	differenze prime (per serie storiche)
`pct_change`	variazione percentuale

Tabella 13.2: Metodi per analisi descrittiva

Funzionamento di base Un esempio:

```
>>> df = pd.DataFrame({'key1' : list('xyyzzzwwww'),
...                    'key2' : ['one', 'two'] * 5,
...                    'data1' : rng.random(10),
...                    'data2' : rng.random(10)},
...                    index = list("abcdefghil")
... )
>>> df
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
b    y  two  0.980548  0.347376
c    y  one  0.135042  0.491588
d    z  two  0.375093  0.955909
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070
g    w  one  0.727694  0.841709
h    w  two  0.355830  0.922741
i    w  one  0.344559  0.687357
l    w  two  0.558278  0.459548

>>> # Serie
>>> data1_spl = df['data1'].groupby(df['key1'])
>>> data1_spl
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7f2d118d1c10>
>>> data1_spl.size()
```

```
key1
w    4
x    1
y    2
z    3
Name: data1, dtype: int64

>>> # DataFrame: uso di nomi
>>> df_spl = df.groupby('key1')
>>> df_spl.mean(numeric_only = True) # necessario se no essendoci key2 da errore
         data1     data2
key1
w     0.496590  0.727839
x     0.167832  0.083391
y     0.557795  0.419482
z     0.667791  0.573986

>>> # Con più chiavi di grouping viene applicato un indexing doppio/gerarchico
>>> df_spl = df.groupby(['key1', 'key2'])
>>> df_spl.mean()
              data1     data2
key1 key2
w    one   0.536126  0.764533
     two   0.457054  0.691145
x    one   0.167832  0.083391
y    one   0.135042  0.491588
     two   0.980548  0.347376
z    one   0.631076  0.238977
     two   0.686148  0.741490

>>> # Uso di un dict col mapping sulla base dell'index
>>> # (series funzionano similmente)
>>> groups = {
...     "a" : 'g1',
...     "b" : 'g1',
...     "c" : 'g1',
...     "d" : 'g1',
...     "e" : 'g2',
...     "f" : 'g2',
...     "g" : 'g2',
...     "h" : 'g2',
...     "i" : 'g2',
...     "l" : 'g2'
... }
>>> df_spl = df.groupby(groups)
>>> df_spl.count()
    key1  key2  data1  data2
g1     4     4      4      4
g2     6     6      6      6
```

```
>>> # Uso di una funzione con mapping sull'index
>>> def ltf(x):
...       return(x < "f")
...
>>> df_spl = df.groupby(ltf)
>>> df_spl.count()
       key1  key2  data1  data2
False     5     5      5      5
True      5     5      5      5

>>> # 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

>>> df2 = df.set_index('key2')
>>> df2.groupby(level = 0).count()
      key1  data1  data2
key2
one      5      5      5
two      5      5      5
>>> df2.groupby(level = 'key2').count()
      key1  data1  data2
key2
one      5      5      5
two      5      5      5
```

Mischiare grouping sulla base di funzioni, array, dict o Series è possibile dato che tutto è convertito internamento ad array

```
>>> df.groupby([ltf, 'key2']).count()
            key1  data1  data2
      key2
False one      2      2      2
      two      3      3      3
True  one      3      3      3
      two      2      2      2
```

Gli oggetti creati sono della classe `GroupBy` sono iterabili (quindi funzionano bene con i `for`) e dispongono di un set di metodi già pronti che vedremo nella prossima sezione. Prima vediamo

Subset di `DataFrame` grouped Una volta fatto il subset di un dataframe risulta comodo poter scegliere su quali variabili elaborare senza dover rifare lo splitting. Si fa cosi:

```
# grouping
df_spl = df.groupby('key1')

# subsetting as usual
df_spl['data1']
```

```
df_spl[['data1', 'data2']]

# applicando funzioni di sommarizzazione si ottengono
# gli stessi risultati di
# df['data1'].groupby(df['key1'])
# df[['data1', 'data2']].groupby(df['key1'])
# ma in questo modo si di effettuarlo a monte
#  e dover  rieffettuare il grouping ogni volta.
```

Se il dataset è molto grande e si analizzano poche colonne conviene fare lo splitting di queste ultime, viceversa conviene splittare tutto il dataset e procedere ad analisi dei subset in un secondo momento.

13.3.3.2 Iterazione sui gruppi

Un oggetto `GroupBy` è iterabile quindi funziona bene con i `for`: al suo interno genera una sequenza di tuple da due elementi contenenti nome del gruppo (chiave) e pezzo di dati il pezzo di dati

```
>>> for name, group in df.groupby('key1'):
...   print(name)
...   print(type(group))
...   print(group)
...
w
<class 'pandas.core.frame.DataFrame'>
  key1 key2     data1     data2
g    w  one  0.727694  0.841709
h    w  two  0.355830  0.922741
i    w  one  0.344559  0.687357
l    w  two  0.558278  0.459548
x
<class 'pandas.core.frame.DataFrame'>
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
y
<class 'pandas.core.frame.DataFrame'>
  key1 key2     data1     data2
b    y  two  0.980548  0.347376
c    y  one  0.135042  0.491588
z
<class 'pandas.core.frame.DataFrame'>
  key1 key2     data1     data2
d    z  two  0.375093  0.955909
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070
```

Nel caso di keys multiple, il primo elemento della tupla sarà una tupla con i valori chiave:

```
>>> for (key1, key2), group in df.groupby(['key1', 'key2']):
...   print(key1, key2)
```

```
...     print(group)
...
w one
  key1 key2     data1     data2
g    w  one  0.727694  0.841709
i    w  one  0.344559  0.687357
w two
  key1 key2     data1     data2
h    w  two  0.355830  0.922741
l    w  two  0.558278  0.459548
x one
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
y one
  key1 key2     data1     data2
c    y  one  0.135042  0.491588
y two
  key1 key2     data1     data2
b    y  two  0.980548  0.347376
z one
  key1 key2     data1     data2
e    z  one  0.631076  0.238977
z two
  key1 key2     data1     data2
d    z  two  0.375093  0.955909
f    z  two  0.997204  0.527070
```

Si può decidere di fare quello che si vuole sull'iteratore (ad esempio anche di trasformarlo in un dict per avervi facile accesso)

```
>>> pieces = dict(list(df.groupby('key1')))
>>> pieces
{'w':   key1 key2     data1     data2
g    w  one  0.727694  0.841709
h    w  two  0.355830  0.922741
i    w  one  0.344559  0.687357
l    w  two  0.558278  0.459548, 'x':   key1 key2     data1     data2
a    x  one  0.167832  0.083391, 'y':   key1 key2     data1     data2
b    y  two  0.980548  0.347376
c    y  one  0.135042  0.491588, 'z':   key1 key2     data1     data2
d    z  two  0.375093  0.955909
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070}
>>> pieces['x']
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
```

13.3.3.3 Data aggregation

Vediamo post aggregazione come creare sintesi a partire dai un gruppo di dati: sono disponibili innanzitutto metodi builtin, si possono specificare funzioni cu-

Metodo	Descrizione
`any`, `all`	Vero se qualcuno o tutti sono veri (nel senso di Python)
`count`	numero di non NA
`cummin`, `cummax`	minimo e massimo cumulato
`cumsum`	somma cumulata
`cumprod`	produttoria cumulata
`first`, `last`	primo, ultimo
`mean`	media
`median`	mediana
`min`, `max`	minimo, massimo
`nth`	n-esimo elemento con dati ordinati
`prod`	prodotto
`quantile`	quantile
`rank`	ranghi
`size`	dimensione del gruppo
`sum`	somma
`std`, `var`	deviazione standard e varianza campionaria

Tabella 13.3: Metodi ottimizzati grouped

stom, e analisi un po più custom rispetto alle standard (es una funzione a tutto il dataset).

Uso di metodi builtin L' oggetto `GroupBy` frutto del metodo omonimo ha un set di funzioni pronte per esser applicate: 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 13.3 sono riportati alcuni metodi notevoli per `DataFrame`.

Applicazione di funzioni custom Per utilizzare delle funzioni di aggregazione custom (ovvero funzioni che a partire da un insieme di dati ci forniscono un unico valore) bisogna passarle al metodo `aggregate` (o la sua abbreviazione `agg`) di un oggetto `GroupBy`

```
>>> def range(x):
...     return(x.max() - x.min())
...
>>> data = ["data1", "data2"]
>>> df.groupby('key1')[data].agg(range)
         data1     data2
key1
w     0.383135  0.463193
x     0.000000  0.000000
y     0.845506  0.144213
z     0.622111  0.716932
```

Aggregazioni più elaborate Si potrebbe voler aggregare usando:

- molteplici funzioni sugli stessi dati (ad esempio per ogni colonna media e deviazione standard);
- diverse funzioni su dati differenti (ad esempio per la prima colonna media per la seconda deviazione standard).

Il dataset che utilizziamo è:

```
>>> df2 = pd.DataFrame({
...     'name': ['Luca', 'Silvio', 'Luisa', 'Andrea', 'Giovanni'],
...     'age' : [21, 22, 23, 24, 25],
...     'income' : np.arange(5),
...     'group' : list("aaabb")
... })
>>> df2
       name  age  income group
0      Luca   21       0     a
1    Silvio   22       1     a
2     Luisa   23       2     a
3    Andrea   24       3     b
4  Giovanni   25       4     b

>>> df_spl = df2.groupby('group')

>>> # lista di funzioni e/o stringhe con nomi di metodi ottimizzati
>>> def range(x):
...     return(x.max() - x.min())
...
>>> analyzed = ["income", "age"]
>>> df_spl[analyzed].agg(['mean', range])
      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[analyzed].agg(analyses)
      income         age
       Media Range Media Range
group
a        1.0     2  22.0     2
b        3.5     1  24.5     1

>>> # dict con variabili analizzate e analisi effettuate
>>> analysis = {'age' : [np.max, np.std], 'income' : np.std}
>>> df_spl.agg(analysis)
       age              income
      amax       std       std
group
```

```
a       23  1.000000  1.000000
b       25  0.707107  0.707107
```

Ritornare dati aggregati senza indexing 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`.

```
>>> df
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
b    y  two  0.980548  0.347376
c    y  one  0.135042  0.491588
d    z  two  0.375093  0.955909
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070
g    w  one  0.727694  0.841709
h    w  two  0.355830  0.922741
i    w  one  0.344559  0.687357
l    w  two  0.558278  0.459548
>>> keys = ['key1', 'key2']

>>> # aggregazione con indici
>>> df_spl = df.groupby(keys)
>>> df_spl.mean(numeric_only = True)
              data1     data2
key1 key2
w    one   0.536126  0.764533
     two   0.457054  0.691145
x    one   0.167832  0.083391
y    one   0.135042  0.491588
     two   0.980548  0.347376
z    one   0.631076  0.238977
     two   0.686148  0.741490

>>> # aggregazione flat
>>> df_spl = df.groupby(keys, as_index = False)
>>> df_spl.mean(numeric_only = True)
  key1 key2     data1     data2
0    w  one  0.536126  0.764533
1    w  two  0.457054  0.691145
2    x  one  0.167832  0.083391
3    y  one  0.135042  0.491588
4    y  two  0.980548  0.347376
5    z  one  0.631076  0.238977
6    z  two  0.686148  0.741490
```

13.3.3.4 Operazioni e trasformazioni group-wise

Le aggregazioni effettuate mediante `aggregate` sino ad ora (da un gruppo di dati a un numero) sono solo una tipologia particolare di operazioni applicabili ad un gruppo di osservazioni. Nel seguito si introducono i metodi `transform` e `apply` che permettono di fare molti altri tipi di operazioni.

`transform` applica una specifica funzione ad un gruppo di valori e sistema l'output: 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 piazzato come appropriato). Calcoliamo lo scarto dalla media di gruppo:

```
>>> def demean(x):
...     return(x - x.mean())
...
>>> df_spl.transform(demean)
      data1     data2
a  0.000000  0.000000
b  0.000000  0.000000
c  0.000000  0.000000
d -0.311056  0.214419
e  0.000000  0.000000
f  0.311056 -0.214419
g  0.191567  0.077176
h -0.101224  0.231597
i -0.191567 -0.077176
l  0.101224 -0.231597
```

`apply` È il metodo più generale di `GroupBy`: splitta l'oggetto manipolato in pezzi, applica la funzione e cerca di ricomporre i risultati. I parametri della funzione vanno passati in `apply`, dopo la funzione, 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,]
...

>>> df
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
b    y  two  0.980548  0.347376
c    y  one  0.135042  0.491588
d    z  two  0.375093  0.955909
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070
g    w  one  0.727694  0.841709
h    w  two  0.355830  0.922741
i    w  one  0.344559  0.687357
```

```
l    w  two  0.558278  0.459548

>>> # applicare a tutto il dataframe
>>> report(df)
  key1 key2     data1     data2
f    z  two  0.997204  0.527070
b    y  two  0.980548  0.347376

>>> # applicare l'estrazione per strato di key2
>>> df_spl = df.groupby('key2')
>>> df_spl.apply(report)
       key1 key2     data1     data2
key2
one  g    w  one  0.727694  0.841709
     e    z  one  0.631076  0.238977
two  f    z  two  0.997204  0.527070
     b    y  two  0.980548  0.347376

>>> # uso di parametri della funzione report, scegliamo di fare
>>> # l'estrazione per data2
>>> df_spl.apply(report, var = 'data2')
       key1 key2     data1     data2
key2
one  g    w  one  0.727694  0.841709
     i    w  one  0.344559  0.687357
two  d    z  two  0.375093  0.955909
     h    w  two  0.355830  0.922741
```

Qualora si vogliano la struttura flat senza indici specificare `group_keys = False` in `groupby`

```
>>> df_spl = df.groupby('key2', group_keys = False)
>>> df_spl.apply(report)
  key1 key2     data1     data2
g    w  one  0.727694  0.841709
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070
b    y  two  0.980548  0.347376
```

13.3.3.5 Tabelle pivot

Sono tabelle bivariate con statistiche stratificate per gruppi formati da righe e colonne spesso disponibili in fogli di elaborazione dati. Possono essere riprodotte con i metodi di cui sopra oppure velocemente col metodo `pivot_table` dei DataFrame, che come default usa `mean` come funzione di aggregazione. Vediamo un esempio alternativo con il calcolo delle frequenze (`count` o `len` a seconda dei valori missing) e aggiungendo i margini per il calcolo

```
>>> df
  key1 key2     data1     data2
a    x  one  0.167832  0.083391
```

```
b    y  two  0.980548  0.347376
c    y  one  0.135042  0.491588
d    z  two  0.375093  0.955909
e    z  one  0.631076  0.238977
f    z  two  0.997204  0.527070
g    w  one  0.727694  0.841709
h    w  two  0.355830  0.922741
i    w  one  0.344559  0.687357
l    w  two  0.558278  0.459548
>>> 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     0.536126  0.457054  0.457054
x     0.167832       NaN  0.167832
y     0.135042  0.980548  0.557795
z     0.631076  0.686148  0.631076
All   0.344559  0.558278  0.466685
```

13.3.3.6 Crosstabulazioni

Sono una speciale tipo di tabella pivot dove si applica il conteggio degli elementi in ciascun gruppo. Vi è una funzione apposta, `crosstab`, per velocizzare ulteriormente

```
>>> pd.crosstab(df.key1, df.key2, margins = True)
key2  one  two  All
key1
w       2    2    4
x       1    0    1
y       1    1    2
z       1    2    3
All     5    5   10
```

13.4 Importazione/esportazione dati

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 xlsx
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

# Alcuni binari notevoli
df.to_excel('path.xlsx')
df.to_feather('path.feather')
df.to_pickle('path.pkl')
df.to_stata('path.dta')
```

Utile 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}
```

Capitolo 14

Grafici

14.1 matplotlib

L'importazione avviene con le seguenti convenzioni

```
import matplotlib as mpl
import matplotlib.pyplot as plt

# altre librerie/utilities necessarie
import numpy as np
import pylbmisc as lb
```

dove `plt` è quello che si usa più spesso. Vi sono due stili diplotting in matplotlib, uno classico state-based che imita matlab e uno object oriented. Preferiamo questo secondo. In sessioni interattive con `ipython`, prima di iniziare a fare grafici diamo che permetterà al tutto di essere interattivo

```
%matplotlib
```

14.1.1 Setup della figura

Una immagine è contenuta in un oggetto `Figure`. Ogni figure ha 1 o più subplots, che sono assi/riferimenti cartesiani nei quali è possibile plotttare

```
# Figura 1x1
fig, ax = plt.subplots()

# Figura 2x3
fig, ax = plt.subplots(2, 3)  # ax è ora un array
                              # es ax[0,2] è ultimo grafico della prima riga
```

Il maggior lavoro si farà con i metodi messi a disposizione da `ax`/`axes`; alcuni metodi utili sono riportati in tabella 14.1. Un esempio minimale con risultato in figura 14.1 segue

```
fig, ax = plt.subplots()
x = np.linspace(0, 10, 100)
ax.plot(x, np.sin(x), label = 'sin(x)')
```

Metodo	Descrizione
`plot`	linee/scatterplot
`hist`	istogramma
`scatter`	scatterplot più flessibile/lento
`errorbar`	intervalli di confidenza
`set`	imposta tutti i seguenti in una unica chiamata
`set_title`	imposta il titolo
`set_xlabel`, `set_xlabel`,	imposta il titolo degli assi
`set_xticks`, `set_yticks`	imposta la posizione dei ticks sugli assi
`set_xticklabels`, `set_yticklabels`	imposta l'etichetta dei ticks
`set_xlim`, `set_ylim`	impostare i limiti degli assi/zoom figura
`legend`	aggiunta di legenda
`text`	aggiunta di testo

Tabella 14.1: Metodi utili di `ax`

```
ax.plot(x, np.cos(x), label = 'cos(x)')
ax.legend()

# salvare su file: il tipo di file è preso dall'estensione
# fig.savefig("/tmp/first_plot.png")
lb.fig.dump(fig, label = 'first_plot')
```

Maggior controllo nei subplots 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 14.2).

```
rng = np.random.default_rng(123)
fig = plt.figure()
ax1 = fig.add_axes([0,       0,   1,   1]) # assi standard
ax2 = fig.add_axes([0.65, 0.65, 0.3, 0.3]) # assino piccolino
# assino piccolino a 0.65, 0.65 della figura e di estensione 0.3x0.3
ax1.plot(np.sin(np.linspace(0, 10)))
ax2.scatter(rng.standard_normal(100), rng.standard_normal(100))
lb.fig.dump(fig, label = 'custom_subplots')
```

Impostare una griglia di grafici custom Si usa `plt.GridSpec` e poi `add_subplot` fornendo la griglia (con slicing) per generare gli `ax` (figura 14.3)

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
fig  = plt.figure()
ax1 = fig.add_subplot(grid[0, 0])
ax2 = fig.add_subplot(grid[0, 1:])
ax3 = fig.add_subplot(grid[1, :2])
ax4 = fig.add_subplot(grid[1, 2])
lb.fig.dump(fig, label = 'custom_grid')
```

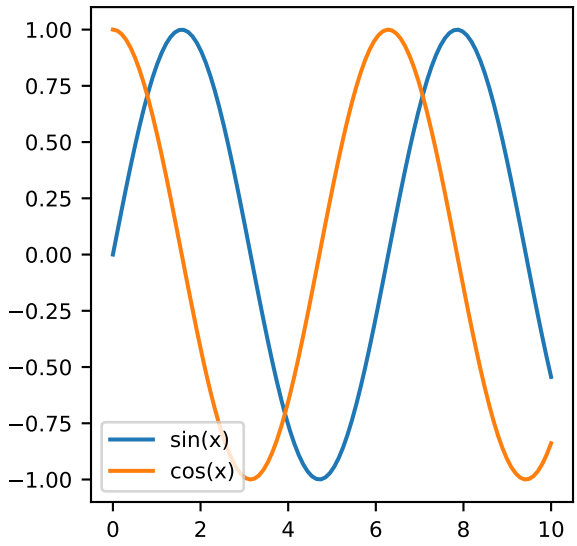

Figura 14.1: First plot

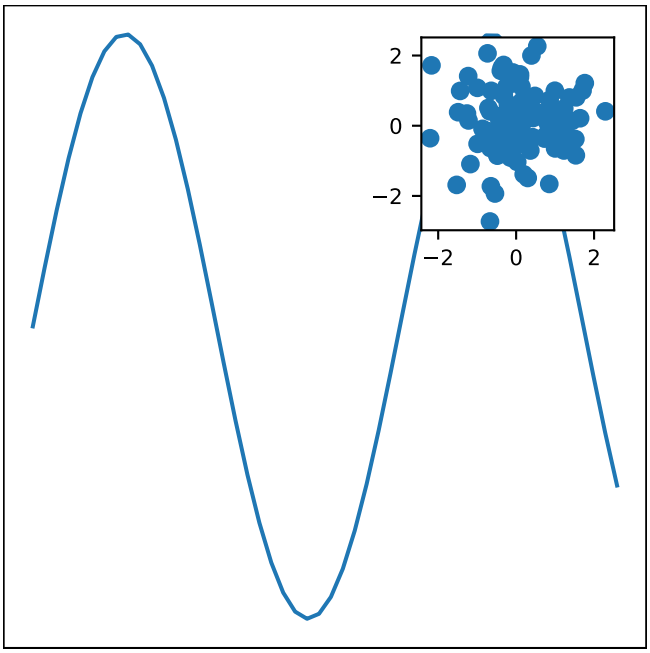

Figura 14.2: Custom subplots

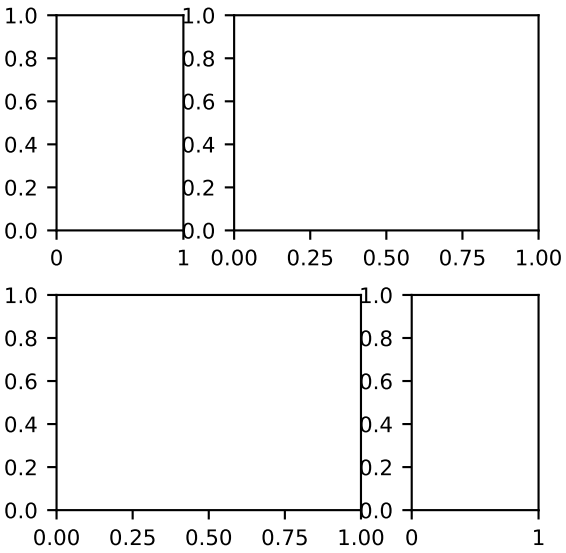

Figura 14.3: Custom grid

Gestire lo spazio bianco tra subplots 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)
```

14.1.2 Configurazioni

Ogni volta che matplotlib si carica definisce delle runtime configuration (rc) che valgono per ciascun plot che creiamo. Queste configurazioni possono essere modificate mediante la funzione `plt.rc` oppure salvandole nel file `.config/matplotlib/matplotlibrc`. Le configurazioni possibili sono listate nel dict `plt.rcParams`

```
plt.rcParams # tutte
plt.rcParams["figure.figsize"]
```

Ad esempio per 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)
```

Se si vuole impostare come valore default, in `.config/matplotlib/matplotlibrc` editare

```
figure.figsize:    3.346 , 3.346  # figure size in inches
```

È possibile ripristinare i valori di default con la funzione `plt.rcdefaults()`.

14.1.2.1 Cambiare stile

Sono disponibili stili di grafico diversi che servono per avere un array di configurazioni già pronto.

```
plt.style.available # listare stili disponibili
```

Per impostare lo stile dei grafici per tutto il resto della sessione si :

```
plt.style.use('default')
```

Per impostare lo stile temporaneamente per una serie di grafici si può usare un context manager:

```
with plt.style.context('stylename'):
    make_a_plot()
```

Parte III

Integrazione

Capitolo 15

Integrazione con R

15.1 Equivalenti di R

15.1.1 Stampa di codice

Il modulo `inspect` permette la stampa del codice sorgente di oggetti (moduli, classi, funzioni ecc):

```
>>> import re
>>> from inspect import getsource
>>> print(getsource(re.compile))
def compile(pattern, flags=0):
    "Compile a regular expression pattern, returning a Pattern object."
    return _compile(pattern, flags)
```

15.1.2 Stampa di dati

Usare il modulo `pprint`

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> pprint.pprint(stuff)
['spam', 'eggs', 'lumberjack', 'knights', 'ni']
```

15.1.3 match.arg

```
>>> def match_arg(arg, choices):
...     res = [expanded for expanded in choices \
...            if expanded.startswith(arg)]
...     l = len(res)
...     if l == 0:
...         raise ValueError("Parameter ", arg, "must be one of ", choices)
...     elif l > 1:
...         raise ValueError(arg, "matches multiple choices from ", choices)
...     else:
...         return res[0]
...
```

```
>>> a = match_arg("foo", ["foobar", "foos", "asdomar"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in match_arg
ValueError: ('foo', 'matches multiple choices from ', ['foobar', 'foos', 'asdomar'])
>>> a = match_arg("foob", ["foobar", "foos", "asdomar"])
>>> print(a)
foobar
```

15.1.4 `on.exit`

Per eseguire codice al termine di una funzione, qualunque cosa accada usare il `try finally`

15.2 Chiamare R da Python: `rpy2`

Si installa

```
pip install --user rpy2
```

Quattro moduli principali:

- `rpy2.robjects`: 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
```

15.2.1 Importazione di pacchetti

```
>>> base = importr('base')
>>> utils = importr('utils')
>>> stats = importr('stats')
>>> lme4 = importr("lme4")
```

15.2.2 Ottenimento di dati

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
```

15.2.3 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
```

15.2.4 Creazione di vettori

```
>>> ints = ro.IntVector([2, 1, 3])
>>> print(ints)
[1] 2 1 3

>>> floats = ro.FloatVector([1.1, 2.2, 3.3])
```

15.2.5 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)
...
```

15.2.6 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 16

PythonTeX

16.1 Installazione e utilizzo

Disponibile in Debian nel pacchetto `texlive-extra-utils`, una volta installato e composto il `file.tex` con gli environments e comandi di cui sotto, per compilarlo la sequenza è

```
pdflatex file.tex   # qui pythontex estrae il codice py in un file temporaneo
pythontex file.tex  # qui esegue il codice e salva gli output
pdflatex file.tex   # qui si mette tutto assieme
```

16.2 Comandi inline ed environment forniti

16.2.1 Comandi inline

I comandi inline presentano la struttura:

`\nomecomando[sessione_opzionale]{codice}`

I principali sono tre, il più utile `\py`:

- `\pyc` (con `c` per code) è usato eseguire ma non visualizzare (equivalente a pycode, se si vuole stampare usare `print`)
- `\py` usato per eseguire codice (anche statement di `print` volendo) e stampare il risultato finale, convertendo in stringa se necessario. Serve tipicamente per richiamare dati/risultati. Se dentro `py` si usa chiama una funzione verrà convertito il valore di `return` (quindi se non restituisce verrà stampato `None`). Il quello che ritorna deve essere valido codice LaTeX, che verrà stampato paro paro;
- `\pyv` (con `v` per verbatim) è usato per visualizzare codice ben formattato, ma non eseguirlo.

16.2.2 Environments

Gli environment hanno una struttura del genere

```
\begin{nomeenvironment}[sessione_opz][formattazione_opz]
  ...
\end{nomeenvironment}
```

Sono disponibili i seguenti quattro, di cui `pycode` è il più utile nel lavoro:

- `pycode` esegue il codice presente ma non lo stampa a video. Stampa solo ciò che è stampato esplicitamente mediante `print`, che deve essere codice LaTeXvalido. Tornano in questo caso comodo le stringhe raw di Python che permettono di non porre la doppia backslash per averne una;

- `pyblock` esegue il codice e lo stampa a video. Il codice stampato con `print` non è automaticamente incluso ma si usano le macro `\printpythontex` dove si vuole che avvenga la stampa

- `pyconsole` simula una sessione a terminale con input e output (differentemente da `pyblock` e `pycode` però non permette l'utilizzo di virtual environments)

- `pyverbatim` visualizza codice senza eseguirlo

16.2.3 Note sugli environments

È meglio:

- lasciare sempre una linea bianca all'inizio e alla fine di un blocco, non stare a ridosso del `begin end` di latex, per esigenze di interprete.

- evitare di mischiare codice che deve essere eseguito sequenzialmente in environment successivi di tipo diverso: non è possibile definire in `pyconsole` e usare in `pycode` o viceversa. È invece possibile definire in `pycode` e usare in `pyblock` (e viceversa)

16.2.4 Uso di funzioni python per la stampa in latex

Una idea (minimale, si potrebbe aggiungere molto altro) che stampa codice latex e che quindi è pensata per essere usata con `pyc` è la seguente:

```
def tabellina(n):
    print(r"\begin{table} \centering \begin{tabular}{cc} \hline")
    for i in range(1, 10 + 1):
        print(i, "&", n*i, r"\\")
    print(r"\hline \end{tabular}")
    print(r"\caption{Tabellina del %d.}" % n)
    print(r"\label{tab:tabellina_del_%d}" % n)
    print(r"\end{table}")
```

Questa stampa già tutto lei quindi la si può usare con `tabellina(7)` entro `\pyc`, come fatto per tabella 16.1.

1	7
2	14
3	21
4	28
5	35
6	42
7	49
8	56
9	63
10	70

Tabella 16.1: Tabellina del 7.

16.2.5 Definizioni di comandi LaTeX che usano python

Una idea a seguire, se in `pycode` o `pyblock` definiamo

```
def pow(x, y):
    return x ** y
```

e in seguito in LaTeXdefiniamo

```
\newcommand{\pow}[2]{\py{pow(#1, #2)}}
```

poi, sempre in LaTeX si potrà usare `\pow{2}{3}`, anche in espressioni matematiche col dollaro, infatti $2^3 = 8$.

16.3 Plot di grafici

Si crea l'immagine in un `pycode` (non visualizzato) e poi la si necessario inserire il grafico in LaTeXnormalmente mediante `includegraphics` che potrà essere stampato con `print` (oppure incluso a mano sotto al codice python, in quello latex).
Sotto mettiamo un esempio minimale da includere in `pycode`; se invece si vuole stampare anche il codice si userà `pyblock` + `\printpythontex` (come avviene in questo documento).

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np


# Data for plotting
t = np.arange(0.0, 2.0, 0.01)
s = 1 + np.sin(2 * np.pi * t)

fig, ax = plt.subplots()
ax.plot(t, s)
ax.set(xlabel = 'time (s)', ylabel='voltage (mV)',
       title = 'About as simple as it gets, folks')
```

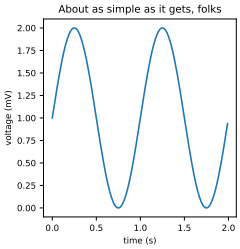

Figura 16.1: Immagine test.

luca	1	2
andrea	3	4
fabio	5	6

Tabella 16.2: Test tabella

```
fig.savefig("/tmp/test.pdf")

# statement di print con includegraphics e il path qui
# usiamo una stringa raw (r) per evitare di dover dare \\ per avere un \
print(r"""
\begin{figure}
  \centering
   \includegraphics[scale=0.4]{/tmp/test}
   \caption{Immagine test.}
   \label{fig:testimg}
\end{figure}
""")
```

16.4 Tabelle

In questo può tornare utile la libreria python `PyLaTeX`. Un esempio minimale (sempre in un `pycode` semplifica, stampando solo il risultato finale) segue mediante l'oggetto `Tabular`, di interesse anche `LongTable` e `LongTabu`.

```
from pylatex import Tabular

data = [["luca", 1, 2], ["andrea", 3, 4], ["fabio", 5,6]]

tab = Tabular("|lcc|")
for row in data:
    tab.add_row(row)

# sempre stringhe raw per semplificare la vita
print(r'\begin{table}\centering',
      tab.dumps(),
      r'\caption{Test tabella} \label{tab:tabletest} \end{table}')
```

Inline	Environment
`R`, `Rc`	`Rcode`
`Rb`	`Rblock`
`Rv`	`Rverbatim`

Tabella 16.3: Integrazione con R

16.5 Valutazione condizionale di codice

Non so come farla a livello di LaTeX, teniamo semplice.

```
dothings = True
if dothings:
  pass
```

16.6 Sessioni

Senza che si specifichi nulla tutto il codice viene eseguito nella sessione `default` sequenzialmente. È possibile specificare che un dato chunk sia eseguito in una data sessione:

- sessioni differenti vengono eseguite in parallelo e sono tra loro indipendenti;
- sessioni differenti possono usare linguaggi differenti;
- è possibile porre codice disponibile a tutte le sessioni (es dei comandi della famiglia `py`) con l'environment `pythontexcustomcode`. Un esempio con librerie comuni segue

  ```
  \begin{pythontexcustomcode}{py}
  import pandas as pd
  \end{pythontexcustomcode}
  ```

16.7 Integrazione con R

Attivando il pacchetto con

```
\usepackage[usefamily=R]{pythontex}
```

si abilitano i comandi di 16.3, che funzionano similmente alle controparti python. Viceversa caricarlo con

```
\usepackage[usefamily=Rcon]{pythontex}
```

abilita l'environment `Rconsole` che emula la sessione interattiva, ed `Rconcode` che esegue il codice ma non prettyprinta nulla.