

ADVANCED PYTHON





RICHIAMI DI PYTHON



LISTE

- Manipolazione attraverso un'espressione
 - Itera su tutta la collezione e ne visita gli elementi
 - Su ogni elemento è possibile valutare un'espressione booleana che determina se l'elemento deve essere aggiunto al risultato



STRINGHE

- Sono array di caratteri
 - In quanto tali si possono usare operatori di `slicing`
- Ricerca testo con operatore `in` / `not in`
- Concatenazione con operatore `+`
- Caratteri di `escape`
- Funzioni principali:
 - `upper()` , `lower()` , `strip()` , `replace()` , `split()` , `format()` , `isXXX()`



LAMBDA

- Funzione «anonima»
 - Può prevedere parametri di input
 - Ha un body composto da un'unica istruzione
 - La sua potenza è ben chiara quando essa è usata all'interno di un'altra funzione
 - In quanto «cattura» eventuali variabili locali

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

```
>>> x = 10
>>> l = lambda a: a * x
>>> print(l(2))
20
>>> x = 20
>>> print(l(2))
40
```



REGULAR EXPRESSIONS

- Un'espressione regolare consente la ricerca di specifiche «classi» di caratteri all'interno di un testo
- Utilizzo
 - `import re`
- Funzioni più importanti

Funzione	Descrizione
<code>findall()</code>	Restituisce una lista che contiene tutti i risultati di una ricerca (match)
<code>search()</code>	Restituisce un oggetto <code>Match</code> se trova corrispondenze
<code>split()</code>	Restituisce una lista di stringhe separata utilizzando i risultati di una ricerca
<code>sub()</code>	Sostituisce ogni match con una stringa



REGULAR EXPRESSIONS

- È preferibile «compilare» le espressioni regolari
- Caratteri e classi
 - Sequenze di caratteri alfanumerici
 - Qualsiasi carattere eccetto il fine riga (.)
 - Alternative (|)
 - Carattere di escape (\ - da non confondere con il carattere di escape nelle stringhe)
 - Insiemi ([])
 - Esclusione(^)
 - Classi particolari (\d \D \w \W \s \S)
 - Quantificatori (* + ? {})



REGULAR EXPRESSIONS

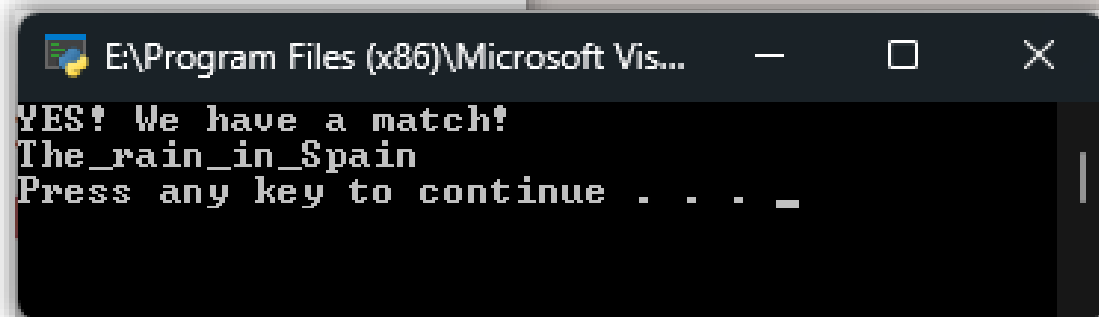
```
import re

#Check if the string starts with "The" and ends with "Spain":

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)

if x:
    print("YES! We have a match!")
else:
    print("No match")

# Replace all spaces with underscore
print(re.sub("\s", "_", txt))
```



E:\Program Files (x86)\Microsoft Vis... — □ ×

```
YES! We have a match!
The_rain_in_Spain
Press any key to continue . . . _
```




NUMPY



NUMPY

- Package fondamentale in ambito scientifico
- Fornisce
 - Array multidimensionali
 - Un vasto assortimento di routines per operazioni performanti su array
 - Matematiche, logiche, manipolazione di forme, ordinamento, selezione, I/O, trasformazioni di Fourier, algebra lineare di base, operazioni statistiche, simulazioni random...



ARRAYS

- L'oggetto principale è un array multidimensionale che contiene dati omogenei
 - È gestito come una tabella di elementi, indicizzata da una tupla di interi non negativi
 - Le dimensioni sono dette **assi**
 - Un punto 3D ha, ad esempio, un solo asse: [1,2,1]
 - La quantità di elementi in un asse ne rappresenta la **lunghezza**



ARRAYS

- La classe principale è `ndarray`, che viene anche usata con l'alias `array`

Attributo	Funzione
<code>ndarray.ndim</code>	Numero di assi (dimensioni) dell'array
<code>ndarray.shape</code>	Tupla che indica la lunghezza in ogni asse
<code>ndarray.size</code>	Il numero totale di elementi nell'array (il prodotto dei valori di shape)
<code>ndarray.dtype</code>	Oggetto che descrive il tipo degli elementi. Possono essere usati tutti i tipi di Python oltre a nuovi tipi introdotti da numpy
<code>ndarray.itemsize</code>	La dimensione in byte di ogni elemento
<code>ndarray.data</code>	Il buffer che contiene gli elementi attualmente nell'array



ARRAYS

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3,5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int32'
>>> a.itemsize
4
>>> a.size
15
```



CREAZIONE DI ARRAY

- Funzione `array()`
 - gli elementi vanno passati con il formato classico degli array (`[]`)
 - le diverse dimensioni vanno passate *come array o tuple*
 - È possibile specificare il tipo usando un parametro `dtype`

```
>>> a = np.array([[1,2],[3,4]], dtype=complex)
>>> a
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
>>> a = np.array([(1,2),(3,4)], dtype=complex)
>>> a
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```



CREAZIONE DI ARRAY

- Funzione **zero()**
 - Crea un array con tutti gli elementi inizializzati a 0
- Funzione **ones()**
 - Crea un array con tutti gli elementi inizializzati a 1
- Funzione **np.random.random()**
 - Crea un array con valori casuali
- Entrambe le funzioni per default creano array di **float64**, a meno che non sia specificato un tipo diverso con il parametro **dtype**

```
>>> a = np.zeros((3,4))
>>> a
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> a = np.ones((3,4), dtype=np.int64)
>>> a
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]], dtype=int64)
```

```
>>> a = np.random.random((3,4))
>>> a
array([[0.60267297, 0.42559004, 0.2388137 , 0.31598706],
       [0.09308972, 0.1910928 , 0.14234325, 0.77500509],
       [0.76116149, 0.97088096, 0.70210109, 0.25530193]])
```



CREAZIONE DI SEQUENZE

- Funzione `arange()`
 - Simile alla funzione `range()` del Python, ma restituisce **arrays** invece di **liste**
- Funzione `linspace()`
 - Consente di creare in maniera più agevole dati float, in quanto consente di specificare il totale degli elementi da produrre e quindi determina lo step automaticamente
 - Utile per la valutazione di funzioni

```
>>> import numpy as np
>>> from numpy import pi
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 2, .3)
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
>>> np.linspace(0, 2, 9)
array([0. , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace(0, 2 * pi, 360)
>>> f = np.sin(x)
```




STAMPA DI ARRAY

- L'ultimo asse è stampato da sinistra a destra
- Gli assi dal secondo in poi sono stampati dall'alto in basso
 - Separando gli assi con un ritorno a capo
- Se l'array è troppo grande, automaticamente la parte centrale è omessa
 - Sostituendola con un'ellissi
 - L'intero array può essere stampato con l'opzione
`set_printoptions(threshold=sys.maxsize)`

```
>>> print(np.arange(10000))
[  0   1   2 ... 9997 9998 9999]
>>> print(np.arange(10000).reshape(100, 100))
[[  0   1   2 ...  97  98  99]
 [100 101 102 ... 197 198 199]
 [200 201 202 ... 297 298 299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```



OPERAZIONI DI BASE

- Gli operatori aritmetici sugli array sono applicati ai singoli elementi (*elementwise*)
 - Restituiscono un nuovo array con il risultato
 - A differenza di altri linguaggi, anche il prodotto è *elementwise*
 - Per il prodotto tra matrici si usa l'operatore `@` o la funzione/metodo `dot`
- Sono presenti gli operatori aritmetici con risultato in-place (es. `+=`)
- Le operazioni tra array contenenti tipi diversi producono come risultato un **array con elementi del tipo più preciso**
 - **Nel caso di incompatibilità di tipo (perdita di informazioni) si ottiene un errore**



OPERAZIONI DI BASE

```
>>> A = np.array([[1,1],[0,1]])
>>> B = np.array([[2,0],[3,4]])
>>> A * B
array([[2, 0],
       [0, 4]])
>>> A + B
array([[3, 1],
       [3, 5]])
```

```
>>> A @ B
array([[5, 4],
       [3, 4]])
>>> A.dot(B)
array([[5, 4],
       [3, 4]])
```

```
>>> A += B
>>> A
array([[3, 1],
       [3, 5]])
```



OPERAZIONI DI BASE

- La classe array contiene numerosi metodi per le operazioni di aggregazione
 - Somma, minimo, massimo, etc.
- Anche per singolo asse o cumulative su tutti gli assi

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a.sum()
66
>>> a.sum(axis = 0) # somma per colonna
array([12, 15, 18, 21])
>>> a.sum(axis = 1) # somma per riga
array([ 6, 22, 38])
>>> a.cumsum(axis = 1) # somma cumulativa per riga
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```



FUNZIONALITÀ

- Indicizzazione
 - Con l'operatore []
- Slicing
 - Con l'operatore [:]
- Iterazione
 - Direttamente iterabili ad esempio in un'istruzione for

```
>>> a = np.arange(10) ** 3
>>> a
array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729], dtype=int32)
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64], dtype=int32)
>>> a[2:6:2] # start da pos. 2 fino a 6 | ogni 2 elementi
array([ 8, 64], dtype=int32)
```

```
>>> for i in a:
...     print(i)
...
0
1
8
27
64
125
216
343
512
729
```



FUNZIONALITÀ

- Gli array multidimensionali hanno un solo indice rappresentato da una tupla di elementi
 - Per ogni dimensione possono essere specificate indicazioni di slicing
- La specifica di un'ellissi (...) rappresenta tutti gli assi per cui si vuole una completa indicizzazione
- Le iterazioni su array multidimensionali vengono fatte rispetto al primo asse
 - A meno che non si utilizzi il metodo `flat()` che «appiattisce» la matrice

```
>>> def f(x, y):  
...     return x + y  
...  
>>> b = np.fromfunction(f, (5,4), dtype=int)  
>>> b  
array([[0, 1, 2, 3],  
       [1, 2, 3, 4],  
       [2, 3, 4, 5],  
       [3, 4, 5, 6],  
       [4, 5, 6, 7]])  
>>> b[2,3]  
5  
>>> b[0:5, 1]  
array([1, 2, 3, 4, 5])  
>>> b[:, 1]  
array([1, 2, 3, 4, 5])  
>>> b[1::2, :]  
array([[1, 2, 3, 4],  
       [3, 4, 5, 6]])
```



MASK INDEX

- Per l'indicizzazione può essere usata un'espressione booleana

```
>>> a = np.arange(100)
>>> print(a[a % 2 == 0])
[ 0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94
 96 98]
```

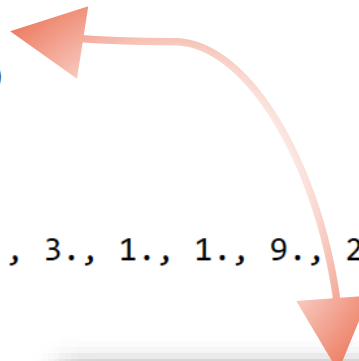
```
>>> a[b]=0
>>> a
array([ 0,  1,  0,  3,  0,  5,  0,  7,  0,  9,  0, 11,  0, 13,  0, 15,  0,
        17,  0, 19,  0, 21,  0, 23,  0, 25,  0, 27,  0, 29,  0, 31,  0, 33,
         0, 35,  0, 37,  0, 39,  0, 41,  0, 43,  0, 45,  0, 47,  0, 49,  0,
        51,  0, 53,  0, 55,  0, 57,  0, 59,  0, 61,  0, 63,  0, 65,  0, 67,
         0, 69,  0, 71,  0, 73,  0, 75,  0, 77,  0, 79,  0, 81,  0, 83,  0,
        85,  0, 87,  0, 89,  0, 91,  0, 93,  0, 95,  0, 97,  0, 99])
```



MANIPOLAZIONE

- Attributo **shape**
 - Ottiene il numero di elementi su ogni asse
- Funzione **ravel()**
 - Restituisce una matrice «*appiattita*»
- Funzione **reshape()**
 - Restituisce un nuovo array modificando il numero di elementi per asse
- Funzione **resize()**
 - Come **reshape()**, ma modifica l'array al quale è applicata
- Attributo **T**
 - Restituisce la matrice trasposta

```
>>> a = np.floor(10 * np.random.random((3,4)))
>>> a
array([[4., 2., 8., 7.],
       [9., 3., 1., 1.],
       [9., 2., 6., 6.]])
>>> a.shape
(3, 4)
>>> a.ravel()
array([4., 2., 8., 7., 9., 3., 1., 1., 9., 2., 6., 6.])
>>> a.reshape(6,2)
array([[4., 2.],
       [8., 7.],
       [9., 3.],
       [1., 1.],
       [9., 2.],
       [6., 6.]])
```



```
>>> a.T
array([[4., 9., 9.],
       [2., 3., 2.],
       [8., 1., 6.],
       [7., 1., 6.]])
```




STACK

- Due array possono essere combinati in stack
 - Orizzontalmente con `hstack()`
 - Verticalmente con `vstack()`
 - In colonna con `column_stack()`
 - In riga con `row_stack()`

```
>>> from numpy import newaxis
>>> a = np.random.random(4)
>>> b = np.random.random(4)
>>> a
array([0.25754657, 0.56433727, 0.70034728, 0.46833978])
>>> b
array([0.52124615, 0.83525478, 0.84492717, 0.5255661 ])
>>> np.hstack((a,b))
array([0.25754657, 0.56433727, 0.70034728, 0.46833978, 0.52124615,
       0.83525478, 0.84492717, 0.5255661 ])
>>> np.vstack((a,b))
array([[0.25754657, 0.56433727, 0.70034728, 0.46833978],
       [0.52124615, 0.83525478, 0.84492717, 0.5255661 ]])
>>> np.column_stack((a,b))
array([[0.25754657, 0.52124615],
       [0.56433727, 0.83525478],
       [0.70034728, 0.84492717],
       [0.46833978, 0.5255661 ]])
>>> np.row_stack((a,b))
array([[0.25754657, 0.56433727, 0.70034728, 0.46833978],
       [0.52124615, 0.83525478, 0.84492717, 0.5255661 ]])
```



VIEW (SHALLOW COPY)

- Una vista (shallow copy) è un riferimento ad un array
 - L'attributo base recupera l'array originale
 - Modifiche di shape non influenzano l'array originale
 - Modifiche agli elementi influenzano l'array originale
- L'operazione di slicing produce una vista

```
>>> a = np.array((1,2,3,4))
>>> c = a.view()
>>> c is a
False
>>> c.base is a
True
>>> c.flags.owndata
False
>>> c.shape = 2,2
>>> c
array([[1, 2],
       [3, 4]])
>>> a
array([1, 2, 3, 4])
```

```
>>> c[1,1] = 10
>>> c
array([[ 1,  2],
       [ 3, 10]])
>>> a
array([ 1,  2,  3, 10])
```



DEEP COPY

- L'operazione di copia profonda crea un altro array a partire da quello originale
 - La copia non condivide alcuna informazione con l'array originale
 - Tutte le operazioni sul nuovo array non influenzano l'array originale
 - L'operazione di copia è utile quando l'array originale viene ridotto con uno slicing per operare solo su quest'ultimo

```
>>> d = a.copy()
>>> d is a
False
>>> d.base is a
False
>>> d.flags.owndata
True
>>> d[1] = 10
>>> a
array([ 1,  2,  3, 10])
>>> d
array([ 1, 10,  3, 10])
```

```
>>> a = np.arange(10000).reshape(100,100)
>>> b = a[:100].copy()
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> b
array([[ 0,  1,  2, ..., 97, 98, 99],
       [100, 101, 102, ..., 197, 198, 199],
       [200, 201, 202, ..., 297, 298, 299],
       ...,
       [9700, 9701, 9702, ..., 9797, 9798, 9799],
       [9800, 9801, 9802, ..., 9897, 9898, 9899],
       [9900, 9901, 9902, ..., 9997, 9998, 9999]])
```

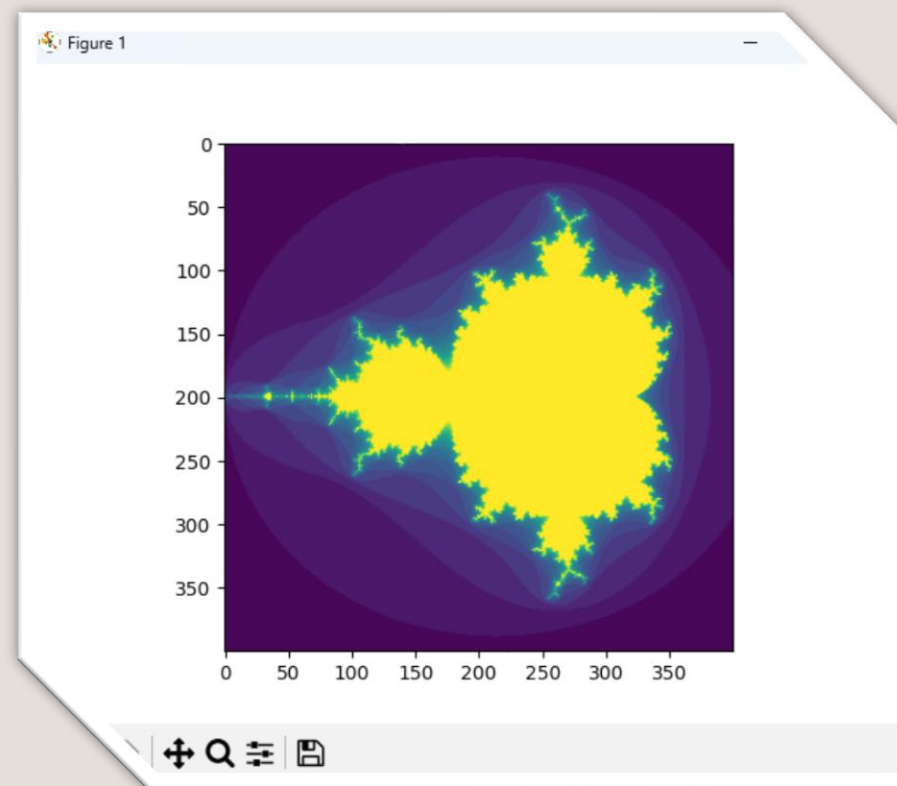


Wow!

```
import numpy as np
import matplotlib.pyplot as plt
def mandelbrot( h,w, maxit=20 ):
    """Returns an image of the Mandelbrot fractal of size (h,w)."""
    y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
    c = x+y*1j
    z = c
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
        z = z**2 + c
        diverge = z*np.conj(z) > 2**2 # who is diverging
        div_now = diverge & (divtime==maxit) # who is diverging now
        divtime[div_now] = i # note when
        z[diverge] = 2 # avoid diverging too much

    return divtime
plt.imshow(mandelbrot(400,400))
plt.show()
```





ALGEBRA LINEARE

- Funzione `inv()`
 - Restituisce l'inversa di una matrice
- Funzione `eye()`
 - Restituisce la matrice identità
- Funzione `dot` o operatore `@`
 - Restituisce la matrice prodotto
- Funzione `trace()`
 - Restituisce la traccia di una matrice
- Funzione `eig()`
 - Restituisce l'autovettore di una matrice

```
>>> i = np.eye(2)
>>> i
array([[1., 0.],
       [0., 1.]])
>>> j = np.array([[0,-1],[1,0]])
>>> j @ j
array([[ -1,  0],
       [ 0, -1]])
>>> y = np.array([[5],[7]])
>>> np.linalg.eig(j)
(array([0.+1.j, 0.-1.j]), array([[0.70710678+0.j, 0.70710678-0.j],
                                [0. -0.70710678j, 0. +0.70710678j]]))
```



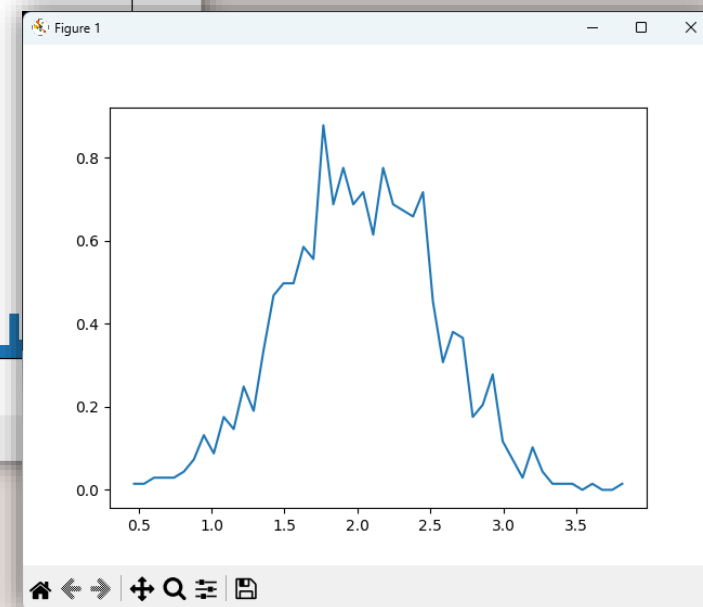
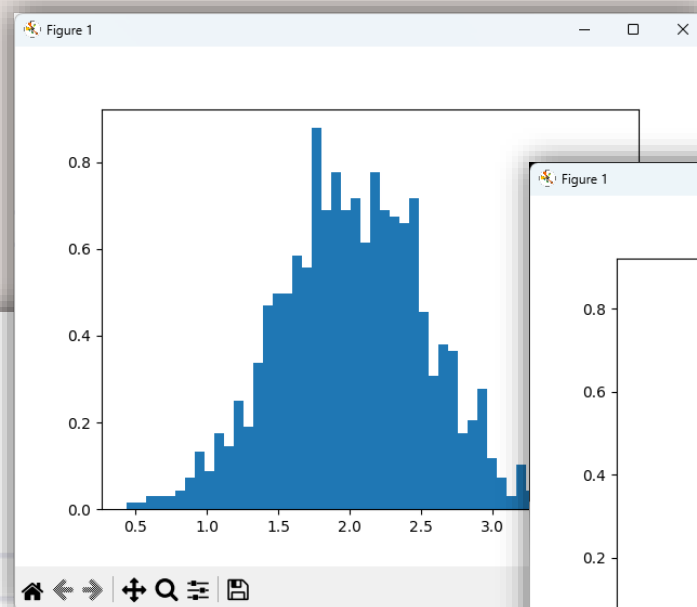
ASPETTI PARTICOLARI

- Funzione `histogram()`
 - Simile alla funzione di `matplotlib` ma genera solo i dati

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

mu, sigma = 2, 0.5
v = np.random.normal(mu, sigma, 1000)
plt.hist(v, bins=50, density=1)
plt.show()

(n, bins) = np.histogram(v, bins=50, density=True)
plt.plot(0.5 * (bins[1:] + bins[:-1]), n)
plt.show()
```





TIPI DI DATO

Numpy type	C type	Description
<i>np.bool_</i>	bool	Boolean (True or False) stored as a byte
<i>np.byte</i>	signed char	Platform-defined
<i>np.ubyte</i>	unsigned char	Platform-defined
<i>np.short</i>	short	Platform-defined
<i>np.ushort</i>	unsigned short	Platform-defined
<i>np.intc</i>	int	Platform-defined
<i>np.uintc</i>	unsigned int	Platform-defined
<i>np.int_</i>	long	Platform-defined
<i>np.uint</i>	unsigned long	Platform-defined
<i>np.longlong</i>	long long	Platform-defined
<i>np.ulonglong</i>	unsigned long long	Platform-defined
<i>np.half</i> / <i>np.float16</i>		Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
<i>np.single</i>	float	Platform-defined single precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
<i>np.double</i>	double	Platform-defined double precision float: typically sign bit, 11 bits exponent, 52 bits mantissa.
<i>np.longdouble</i>	long double	Platform-defined extended-precision float
<i>np.csingle</i>	float complex	Complex number, represented by two single-precision floats (real and imaginary components)
<i>np.cdouble</i>	double complex	Complex number, represented by two double-precision floats (real and imaginary components).
<i>np.clongdouble</i>	long double complex	Complex number, represented by two extended-precision floats (real and imaginary components).

Numpy type	C type	Description
<i>np.int8</i>	int8_t	Byte (-128 to 127)
<i>np.int16</i>	int16_t	Integer (-32768 to 32767)
<i>np.int32</i>	int32_t	Integer (-2147483648 to 2147483647)
<i>np.int64</i>	int64_t	Integer (-9223372036854775808 to 9223372036854775807)
<i>np.uint8</i>	uint8_t	Unsigned integer (0 to 255)
<i>np.uint16</i>	uint16_t	Unsigned integer (0 to 65535)
<i>np.uint32</i>	uint32_t	Unsigned integer (0 to 4294967295)
<i>np.uint64</i>	uint64_t	Unsigned integer (0 to 18446744073709551615)
<i>np.intp</i>	intptr_t	Integer used for indexing, typically the same as <i>ssize_t</i>
<i>np.uintp</i>	uintptr_t	Integer large enough to hold a pointer
<i>np.float32</i>	float	
<i>np.float64</i> / <i>np.float_</i>	double	Note that this matches the precision of the builtin python <i>float</i> .
<i>np.complex64</i>	float complex	Complex number, represented by two 32-bit floats (real and imaginary components)
<i>np.complex128</i> / <i>np.complex_</i>	double complex	Note that this matches the precision of the builtin python <i>complex</i> .

Conversione di Tipo:

- Funzione **astype()**
- Attributo **dtype**



FUNZIONE GENFROMTXT()

- Unico parametro obbligatorio è la sorgente
 - Stringa, lista di stringhe, generatore, file aperto con `read()`
- Riconosce file di testo o archivi
 - Archivi `gzip` o `bz2`
 - Il tipo di archivio è determinato dall'estensione
- L'argomento `delimiter` consente di specificare un delimitatore per la separazione dei valori
- L'argomento `autostrip` consente di eliminare automaticamente spazi iniziali e finali
- L'argomento `comment` consente di specificare un delimitatore di testo da non considerare
- Gli argomenti `skip_header` e `skip_footer` consentono di saltare righe iniziali e finali
- L'argomento `usecols` consente di specificare le colonne di interesse (tramite nome o indice)



FUNZIONE GENFROMTXT()

```
import numpy as np
import re

def floatconvert(x):
    x = re.sub(b"\\.", b"", x)
    x = re.sub(b",", b".", x)
    return float(x)

with open("statistiche.csv") as f:
    print(
        np.genfromtxt(
            f,
            skip_header=1,
            delimiter=";",
            dtype=(int, "|U5", float, float, int, float),
            usecols=(2, 3, 4, 5, 7, 8),
            converters={
                4: floatconvert,
                5: floatconvert,
                7: floatconvert,
                8: floatconvert,
            },
        )
    )
```

	A	B	C	D	E	F	G	H	I	J	K	L
1	Codice Re	Codice Pr	Codice Co	Denominazione	Superficie	Superficie	Classi di s	Popolazio	Densità abitativa (abitanti per Km2)			
2	1	1	1001	Agliè	1.314,64	13,15	2	2.644	201,12			
3	1	1	1002	Airasca	1.573,95	15,74	2	3.819	242,64			
4	1	1	1003	Ala di Stur	4.633,22	46,33	3	462	9,97			
5	1	1	1004	Albiano d'	1.173,16	11,73	2	1.791	152,66			
6	1	1	1005	Alice Supè	737,97	7,38	1	701	94,99			
7	1	1	1006	Almese	1.787,59	17,88	2	6.303	352,6			
8	1	1	1007	Alpette	562,62	5,63	1	277	49,23			
9	1	1	1008	Alpignanc	1.191,95	11,92	2	16.893	1.417,26			
10	1	1	1009	Andezenc	748,61	7,49	1	1.966	262,62			
11	1	1	1010	Andrate	930,87	9,31	1	512	55			
12	1	1	1011	Angrogna	3.887,88	38,88	3	870	22,38			
13	1	1	1012	Arignano	816,7	8,17	1	1.039	127,22			
14	1	1	1013	Avigliana	2.321,75	23,22	2	12.129	522,41			
15	1	1	1014	Azeglio	996,07	9,96	1	1.347	135,23			
16	1	1	1015	Bairo	708,6	7,09	1	816	115,16			
17	1	1	1016	Bellinzona	1.388,66	13,89	2	2.164	213,88			
18	1	1	1017	Bellinzona	1.388,66	13,89	2	2.164	213,88			
19	1	1	1018	Bellinzona	1.388,66	13,89	2	2.164	213,88			
20	1	1	1019	Bellinzona	1.388,66	13,89	2	2.164	213,88			
21	1	1	1020	Bellinzona	1.388,66	13,89	2	2.164	213,88			
22	1	1	1021	Bellinzona	1.388,66	13,89	2	2.164	213,88			
23	1	1	1022	Bellinzona	1.388,66	13,89	2	2.164	213,88			
24	1	1	1023	Bellinzona	1.388,66	13,89	2	2.164	213,88			

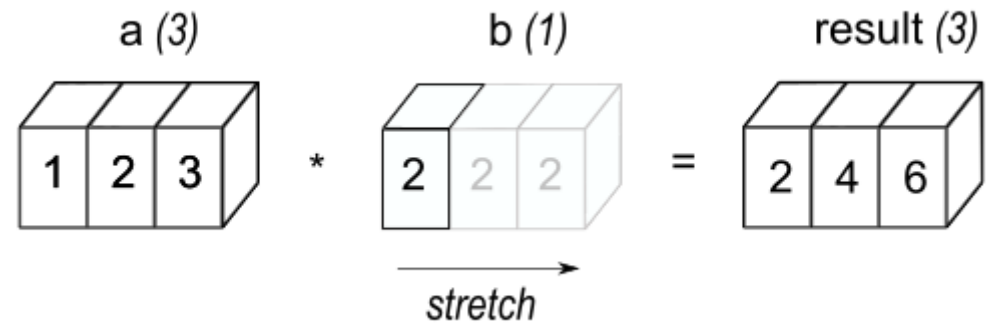
```
E:\Program Files (x86)\Microsoft Visual Studio\Shared\Pyt...
[< 1001, 'Agliè', 1314.64, 13.15, 2644, 201.12>
< 1002, 'Airas', 1573.95, 15.74, 3819, 242.64>
< 1003, 'Ala d', 4633.22, 46.33, 462, 9.97> ...
<107021, 'Trata', 3100.29, 31., 1107, 35.71>
<107022, 'Villa', 9138.9, 91.39, 3655, 39.99>
<107023, 'Villa', 3642.89, 36.43, 1097, 30.11>]
Press any key to continue . . .
```



BROADCASTING

- Funzionalità interna per l'effettuazione di operazioni che coinvolgono elementi con diverse dimensioni

```
>>> a = np.array([1.0, 2.0, 3.0])  
>>> b = 2.0  
>>> a * b  
array([2., 4., 6.])
```





ARRAY STRUTTURATI

- Array organizzati in maniera tale che i dati interni non siano scalari, ma composti da sequenze campo/valore

```
>>> x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],  
...              dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')])  
>>> x  
array([('Rex', 9, 81.), ('Fido', 3, 27.)],  
      dtype=[('name', 'U10'), ('age', '<i4'), ('weight', '<f4')])
```

```
>>> x[1]  
('Fido', 3, 27.0)
```



ARRAY STRUTTURATI

- L'accesso alle posizioni avviene normalmente e come risultato si ottiene un elemento strutturato chiave/valore

```
>>> x[1]
('Fido', 3, 27.0)
```

- L'accesso può avvenire attraverso il nome del campo

```
>>> x['age']
array([9, 3], dtype=int32)
>>> x['age'] = 5
>>> x
array([('Rex', 5, 81.), ('Fido', 5, 27.)],
      dtype=[('name', 'U10'), ('age', '<i4'), ('weight', '<f4')])
```

- I dati strutturati corrispondono alle **struct** del **C** e ne condividono la rappresentazione in memoria



TIPI DI DATO STRUTTURATI

- Sequenze di bytes di una determinata lunghezza interpretate come una collezione di campi
 - Ognuno di essi ha un nome, un tipo e un offset di bytes all'interno della struttura
- Il tipo può contenere qualsiasi **datatype** di **numpy** inclusi altri tipi strutturati
- L'offset interno generalmente è determinato da **numpy**
 - Ma può essere specificato anche manualmente



CREAZIONE DI DATI STRUTTURATI

- Funzione `numpy.dtype()`
 - Usa diverse modalità
 - Una lista di tuple, dove ogni tupla rappresenta un campo
 - Una stringa con specifiche di tipo separate da virgola
 - Un dizionario di campi in un array
 - Un dizionario di nomi di campi



CREAZIONE DI DATI STRUTTURATI

```
>>> # a list of tuples, one tuple per field
... np.dtype([('x', 'f4'), ('y', np.float32), ('z', 'f4', (2, 2))])
dtype([('x', '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
>>> # a nonnamed field is replaced by fX notation
... np.dtype([('x', 'f4'), ('', np.float32), ('z', 'f4', (2, 2))])
dtype([('x', '<f4'), ('f1', '<f4'), ('z', '<f4', (2, 2))])
```



CREAZIONE DI DATI STRUTTURATI

```
>>> # a list of tuples, one tuple per field
... np.dtype([('x', 'f4'), ('y', np.float32), ('z', 'f4', (2, 2))])
dtype([('x', '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
>>> # a nonnamed field is replaced by fX notation
... np.dtype([('x', 'f4'), ('', np.float32), ('z', 'f4', (2, 2))])
dtype([('x', '<f4'), ('f1', '<f4'), ('z', '<f4', (2, 2))])
```

```
>>> ar = np.array((1, 2, (3,4)), dtype=t)
>>> ar
array((1., 2., [[3., 4.], [3., 4.]]),
      dtype=[('x', '<f4'), ('f1', '<f4'), ('z', '<f4', (2, 2))])
```




CREAZIONE DI DATI STRUTTURATI

```
>>> # a string of comma-separated dtype specifications
... t = np.dtype('i8, f4, S3')
>>> t
dtype([('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')])
>>> a = np.array((1,2,3), dtype=t)
>>> a
array((1, 2., b'3'), dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')])
```



CREAZIONE DI DATI STRUTTURATI

```
>>> # a dictionary of parameter arrays
... t = np.dtype({'names': ['col1', 'col2'], 'formats': ['i4', 'f4']})
>>> t
dtype([('col1', '<i4'), ('col2', '<f4')])
>>> a = np.array([1,2], dtype=t)
>>> a
array([(1, 1.), (2, 2.)], dtype=[('col1', '<i4'), ('col2', '<f4')])

dtype({'names': ['col1', 'col2'], 'formats': ['<i4', '<f4'], 'offsets': [0, 4],
      ↪ 'itemsize': 12})
```



CREAZIONE DI DATI STRUTTURATI

```
>>> # a dictionary of field names
... t = np.dtype({'col1': ('i1', 0), 'col2': ('f4', 1)})
>>> t
dtype([('col1', 'i1'), ('col2', '<f4')])
```

```
>>> a = np.ones(3, dtype=t)
>>> a
array([(1, 1.), (1, 1.), (1, 1.)], dtype=[('col1', 'i1'), ('col2', '<f4')])
>>> a['col1'] = 2
>>> a
array([(2, 1.), (2, 1.), (2, 1.)], dtype=[('col1', 'i1'), ('col2', '<f4')])
>>> a['col1'][0] = 3
>>> a
array([(3, 1.), (2, 1.), (2, 1.)], dtype=[('col1', 'i1'), ('col2', '<f4')])
```



RECORD ARRAYS

- Sottoclasse di array
 - `numpy.recarray` nel submodule `numpy.rec`
 - `numpy.record` datatype
 - Consente l'accesso agli scalari interni all'array attraverso un attributo



RECORD ARRAYS - CREAZIONE

```
>>> recordarr = np.rec.array([(1, 2., 'Hello'), (2, 3., "World")],  
... dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])  
>>> recordarr.bar  
array([2., 3.], dtype=float32)  
>>> recordarr[1:2]  
rec.array([(2, 3., b'World')],  
          dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')])  
>>> recordarr[1:2].foo  
array([2])  
>>> recordarr[1].baz  
b'World'
```

```
>>> arr = np.array([(1, 2., 'Hello'), (2, 3., "World")],  
... dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])  
>>> recordarr = np.rec.array(arr)  
>>> recordarr  
rec.array([(1, 2., b'Hello'), (2, 3., b'World')],  
          dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')])
```



RECORD ARRAYS – CREAZIONE DA VIEW

```
>>> arr = np.array([(1, 2., 'Hello'), (2, 3., "World")],  
... dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'a10')])  
...  
>>> recordarr = arr.view(dtype=np.dtype((np.record, arr.dtype)),  
... type=np.recarray)  
>>> recordarr  
rec.array([(1, 2., b'Hello'), (2, 3., b'World')],  
          dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')])
```



RECORD ARRAYS – FUNZIONI HELPER

- `numpy.lib.recfunzioms.append_fields(
 base, names, data, dtypes=None, fill_value=-1,
 use-mask=True, asrecarray=False)`
 - Aggiunge nuovi campi ad un array esistente
- `numpy.lib.recfunctions.apply_along_fields(func, arr)`
 - Applica una funzione di riduzione sui campi dell'array strutturato
- `numpy.lib.recfunctions.assign_fields_by_name(
 dst, src, zero_unassigned=True)`
 - Assegna valori da un array strutturato ad un altro



RECORD ARRAYS – FUNZIONI HELPER

- `numpy.lib.recfunctions.drop_fields(
 base, drop_names, usemask=True, asrecarray=False)`
 - Restituisce un nuovo array senza i campi definiti in `drop_names`
- `numpy.lib.recfunctions.find_duplicates(
 a, key=None, ignoremask=True, return_index=False)`
 - Cerca i duplicate in un array strutturato rispetto ad una determinata chiave
- `numpy.lib.recfunctions.flatten_descr(ndtype)`
 - Appiattisce la descrizione di un tipo di dati strutturato



RECORD ARRAYS – ESEMPI HELPER

```
>>> from numpy.lib import recfunctions as rfn
>>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8, 11), (10, 11, 12)],
...              dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
>>> rfn.apply_along_fields(np.mean, b)
array([ 2.66666667,  5.33333333,  8.66666667, 11.
])
>>> rfn.apply_along_fields(np.mean, b[['x', 'z']])
array([ 3. ,  5.5,  9. , 11. ])
```

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = [('a', int)]
>>> a = np.ma.array([1, 1, 1, 2, 2, 3, 3],
...                 mask=[0, 0, 1, 0, 0, 0, 1]).view(ndtype)
>>> rfn.find_duplicates(a, ignoremask=True, return_index=True)
(masked_array(data=[(1,), (1,), (2,), (2,)],
               mask=[(False,), (False,), (False,), (False,)],
               fill_value=999999,
               dtype=[('a', '<i8')]), array([0, 1, 3, 4]))
```

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
...              dtype=[('a', np.int64), ('b', [('ba', np.double), ('bb', np.int64)])])
>>> rfn.drop_fields(a, 'a')
array([(2., 3.), (5., 6.)],
      dtype=[('b', [('ba', '<f8'), ('bb', '<i8')])])
>>> rfn.drop_fields(a, 'ba')
array([(1, (3,)), (4, (6,))], dtype=[('a', '<i8'), ('b', [('bb', '<i8')])])
>>> rfn.drop_fields(a, ['ba', 'bb'])
array([(1,), (4,)], dtype=[('a', '<i8')])
```

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = np.dtype([('a', '<i4'), ('b', [('ba', '<f8'), ('bb', '<i4')])])
>>> rfn.flatten_descr(ndtype)
(('a', dtype('int32')), ('ba', dtype('float64')), ('bb', dtype('int32')))
```



CUSTOM CONTAINERS

- Meccanismo di **dispatch** (v. 1.16)
 - Approccio per la scrittura di array n-dimensionali compatibili con le API di **numpy**
- La creazione di containers custom consente di incapsulare la logica di gestione di array all'interno di una classe che è vista da **numpy** come un array standard
 - Metodi
 - **__array__**
 - Consente l'ottenimento di un array **numpy** utilizzabile come tutti gli array standard



CUSTOM CONTAINERS

```
>>> class DiagonalArray:
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self):
...         return self._i * np.eye(self._N)
...
>>> a = DiagonalArray(5, 11)
>>> a
DiagonalArray(N=5, value=11)
```

```
>>> np.asarray(a)
array([[11.,  0.,  0.,  0.,  0.],
       [ 0., 11.,  0.,  0.,  0.],
       [ 0.,  0., 11.,  0.,  0.],
       [ 0.,  0.,  0., 11.,  0.],
       [ 0.,  0.,  0.,  0., 11.]])
```

```
>>> arr = np.multiply(a, 2)
>>> arr
array([[22.,  0.,  0.,  0.,  0.],
       [ 0., 22.,  0.,  0.,  0.],
       [ 0.,  0., 22.,  0.,  0.],
       [ 0.,  0.,  0., 22.,  0.],
       [ 0.,  0.,  0.,  0., 22.]])
>>> type(arr)
<class 'numpy.ndarray'>
```



CUSTOM CONTAINERS

- Per il mantenimento del tipo durante il passaggio ad una funzione di libreria
 - Metodo `__array_ufunc__`
 - Per funzioni identificate da numpy come ufunc (numpy.multiply, numpy.sin, ...)
 - Metodo `__array_function__`
 - Per le altre funzioni del Python



NDARRAY SUBCLASSING

- Una sottoclasse di `ndarray` si costruisce in 3 modi diversi
 - Chiamata diretta al costruttore
 - `MySubClass(params)`
 - View casting
 - Cast da un `ndarray` esistente verso la nostra classe
 - Creazione tramite template
 - Come la precedente usata per supportare le operazioni di slicing



NDARRAY SUBCLASSING

- View casting
 - Meccanismo standard per ottenere un `ndarray` di una determinata sottoclasse

```
>>> class C(np.ndarray): pass
...
>>> arr = np.zeros((3,))
>>> arr
array([0., 0., 0.])
>>> type(arr)
<class 'numpy.ndarray'>
>>> c_arr = arr.view(C)
>>> type(c_arr)
<class '__main__.C'>
```



NDARRAY SUBCLASSING

- Creazione da template
 - Usata, ad esempio, quando occorre avere uno slice di un array

```
>>> v = c_arr[1:]  
>>> type(v)  
<class '__main__.C'>  
>>> v is c_arr  
False
```



NDARRAY SUBCLASSING

- Differenze tra view cast e template
 - View cast significa creare una nuova istanza di un array custom a partire da ogni potenziale sottoclasse di `ndarray`
 - Creare da un template significa creare una nuova istanza a partire da una esistente
 - Consentendo di effettuare operazioni su particolari attributi



IMPLICAZIONI NEL SUBCLASSING

- Non solo dobbiamo gestire la costruzione diretta ma
 - Dobbiamo gestire la costruzione tramite view casting e tramite template
 - Questo significa che
 - Dobbiamo ridefinire `ndarray.__new__()`
 - Dobbiamo ridefinire `__array_finalize__()`



METODI `__new__` E `__init__`

- Il metodo `__new__` viene richiamato prima del metodo `__init__` (se presente)

```
>>> class C(object):
...     def __new__(cls, *args):
...         print('__new__')
...         print('\tclasse: ', cls)
...         print('\targomenti: ', args)
...         return object.__new__(cls)
...
>>> class C(object):
...     def __new__(cls, *args):
...         print('__new__')
...         print('\tclasse: ', cls)
...         print('\targomenti: ', args)
...         return object.__new__(cls)
...     def __init__(self, *args):
...         print('__init__')
...         print('\tself è di tipo ', type(self))
...         print('\targomenti: ', args)
```

```
>>> C('Hello, World!')
__new__
    classe:  <class '__main__.C'>
    argomenti:  ('Hello, World!',)
__init__
    self è di tipo  <class '__main__.C'>
    argomenti:  ('Hello, World!',)
```



METODI `__new__` E `__init__`

- Perché usare `__new__()` invece di `__init__()`?

```
>>> class D(C):
...     def __new__(cls, *args):
...         print('D.__new__')
...         print('\tclasse: ', cls)
...         print('\targomenti: ', args)
...         return C.__new__(C, *args)
...     def __init__(self, *args):
...         print('D.__init__')
```

Da notare che il metodo `__init__()` di D non è invocato

```
>>> o = D("Hello, World!")
D.__new__
      classe: <class '__main__.D'>
      argomenti: ('Hello, World!')
__new__
      classe: <class '__main__.C'>
      argomenti: ('Hello, World!')
>>> type(o)
<class '__main__.C'>
```



METODI `__new__` E `__init__`

- Perché usare `__new__()` invece di `__init__()`?

```
>>> class D(C):
...     def __new__(cls, *args):
...         print('D.__new__')
...         print('\tclasse: ', cls)
...         print('\targomenti: ', args)
...         return C.__new__(D, *args)
...     def __init__(self, *args):
...         print('D.__init__')
```

Nel metodo `__new__()` occorre usare la subclass

```
>>> o = D("Hello, World!")
D.__new__
      classe: <class '__main__.D'>
      argomenti: ('Hello, World!')
__new__
      classe: <class '__main__.D'>
      argomenti: ('Hello, World!')
D.__init__
>>> type(o)
<class '__main__.D'>
```



SUBCLASSING

ABBIAMO PERÒ UN PROBLEMA!

Un `ndarray` può costruire la classe in questo modo, attraverso i suoi operatori di slicing e view, ma il metodo `__new__()` di `ndarray` non conosce affatto i nostri modi di gestire gli attributi, perché spesso non abbiamo la stessa firma!



__ARRAY_FINALIZE__()

- Meccanismo usato da **numpy** per consentire alle sottoclassi di gestire i diversi modi per ottenere una nuova istanza
 - Nel caso di costruzione tramite costruttore o tramite view casting basterà ridefinire `__new__()` e `__init__()`
 - Nel caso di costruzione tramite template dobbiamo ridefinire `__array_finalize__()`
 - Quindi nelle sottoclassi deleghiamo al metodo `__array_finalize__()` la costruzione in tutti e tre i casi



__ARRAY_FINALIZE__()

```
>>> class C(np.ndarray):
...     def __new__(cls, *args, **kwargs):
...         print('__new__ con classe %s' % cls)
...         return super(C, cls).__new__(cls, *args, **kwargs)
...     def __init__(self, *args, **kwargs):
...         print('__init__ con classe %s' % self.__class__)
...     def __array_finalize__(self, obj):
...         print('__array_finalize__')
...         print('\tself type: %s' % type(self))
...         print('\tobj type: %s' % type(obj))
```

```
>>> c = C((10,))
__new__ con classe %s <class '__main__.C'>
__array_finalize__
self type: <class '__main__.C'>
obj type: <class 'NoneType'>
__init__ con classe <class '__main__.C'>
```

```
>>> a = np.arange(10)
>>> c_a = a.view(C)
__array_finalize__
self type: <class '__main__.C'>
obj type: <class 'numpy.ndarray'>
```

```
>>> cv = c[1:]
__array_finalize__
self type: <class '__main__.C'>
obj type: <class '__main__.C'>
```



PANDAS



SERIES

- Array monodimensionale etichettato e denominato (**name**)
 - In grado di gestire qualsiasi tipo di dato (un unico dtype)
 - L'asse delle etichette è referenziato come **index**

```
import numpy as np
import pandas as pd

s = pd.Series(np.random.random(5), index=["a", "b", "c", "d", "e"])
print(s)
print(s.index)

a = pd.Series(np.random.random(5))
print(a)
print(a.index)
```

```
E:\Program Files (x86)\Microsoft Visual Studio...
a    0.923647
b    0.586965
c    0.317488
d    0.827166
e    0.590847
dtype: float64
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
0    0.438624
1    0.429851
2    0.971796
3    0.290307
4    0.886055
dtype: float64
RangeIndex(start=0, stop=5, step=1)
Press any key to continue . . . _
```



SERIES

- Le serie possono essere istanziate

- tramite dizionari

- se è indicato anche un indice, saranno restituiti tutti i valori specificati nell'indice

- tramite scalare

```
>>> pd.Series(10, index=["a", "b", "c"])
a    10
b    10
c    10
dtype: int64
```

```
>>> d = {"a": 1, "b": 2, "c": 3}
>>> pd.Series(d)
a    1
b    2
c    3
dtype: int64
>>> pd.Series(d, index=["a", "c", "d", "b"])
a    1.0
c    3.0
d    NaN
b    2.0
dtype: float64
```



SERIES

- Le serie sono simili a `ndarray`
- Le serie sono simili a dizionari

```
>>> a = pd.Series((1,2,3))
>>> a[a > a.median()]
2      3
dtype: int64
```

```
>>> s = pd.Series(10, index=["a","b","c"])
>>> s["b"]
10
>>> s["c"] = 15
>>> s
a      10
b      10
c      15
dtype: int64
```



DATAFRAME

- Simile ad uno spreadsheet o a una tabella SQL
 - o ad un dizionario di oggetti **Series**
- Oggetto di uso più comune in **pandas**
- Accetta diversi tipi di input
 - Dizionario di **ndarray** monodimensionali, liste, dicts, **Series**
 - Un **ndarray** a due dimensioni
 - Un **ndarray** strutturato o di tipo record
 - Una serie
 - Un altro **DataFrame**
- Contestualmente alla costruzione possiamo specificare etichette di riga (index) o di colonna (columns) per garantire che il risultato risponda alla configurazione desiderata



DATAFRAME

- Costruzione tramite serie

```
>>> d = {  
...     "uno": pd.Series([1,2,3], index=["a","b","d"]),  
...     "due": pd.Series([1,2,3,4], index=["a","b","c","d"])}  
>>> df = pd.DataFrame(d)  
>>> df  
   uno  due  
a  1.0    1  
b  2.0    2  
c  NaN    3  
d  3.0    4
```



DATAFRAME

- Costruzione tramite `ndarray` / list

```
>>> d = {"uno": [1., 2., 3., 4.], "due": [4., 5., 6., 7.]}
>>> pd.DataFrame(d)
   uno  due
0  1.0  4.0
1  2.0  5.0
2  3.0  6.0
3  4.0  7.0
```



DATAFRAME

- Costruzione da array strutturato o record

```
>>> d = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
>>> d[:] = [(1,2,"Hello"), (2,3,"World")]
>>> pd.DataFrame(d)
```

	A	B	C
0	1	2.0	b'Hello'
1	2	3.0	b'World'



DATAFRAME

- Costruzione da lista di dizionari e da dizionario di tuple

```
data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]
print(pd.DataFrame(data2))
print(pd.DataFrame(data2, index=["first", "second"]))
print(pd.DataFrame(data2, columns=["a", "b"]))

print(
    pd.DataFrame(
        {
            ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
            ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
            ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
            ("b", "a"): {("A", "C"): 7, ("A", "B"): 8},
            ("b", "b"): {("A", "D"): 9, ("A", "B"): 10},
        }
    )
)
```

```
E:\Program Files (x86)\Mic...
a  b  c
0  1  2 NaN
1  5 10 20.0

   a  b  c
first 1  2 NaN
second 5 10 20.0

   a  b
0  1  2
1  5 10

   a  b
A B 1.0 4.0 5.0 8.0 10.0
C 2.0 3.0 6.0 7.0 NaN
D NaN NaN NaN NaN 9.0
Press any key to continue . . .
```




DATAFRAME – LAVORARE CON LE COLONNE

- Possiamo gestire i **DataFrame** come delle **Series** «etichettate»
- Le colonne possono essere eliminate con l'operatore **del** o tramite il metodo **pop()**
- L'aggiunta di una colonna avviene attraverso la definizione del suo nome nel dataframe
 - Uno scalare riempie tutta la colonna
 - Una serie viene uniformata all'indice del dataframe

```
>>> d = pd.DataFrame(data2)
>>> d["new col"] = "new value"
>>> d
```

	a	b	c	new col
0	1	2	NaN	new value
1	5	10	20.0	new value

```
>>> d["s_col"] = pd.Series([10,20,30,40])
>>> d
```

	a	b	c	new col	s_col
0	1	2	NaN	new value	10
1	5	10	20.0	new value	20



FUNZIONALITÀ DI BASE

- Testa e coda
 - `head()` / `tail()`
- Conversione ad array
 - Attributo `array` (`index.array` per gli indici)
 - Metodo `to_numpy()` o `numpy.asarray()`
 - Per conversione verso numpy
- Operazioni «accelerate»
 - È possibile usare sia `numexpr` che `bottleneck`
 - `pd.set_option("compute.use_bottleneck", False)`
 - `pd.set_option("compute.use_numexpr", True)`



FUNZIONALITÀ DI BASE

- Confronti flessibili
 - Metodi `eq()`, `ne()`, `lt()`, `gt()`, `le()`, `ge()`
 - Restituiscono un oggetto dello stesso tipo dell'operando di sinistra
- Riduzioni booleane
 - Metodi `any()`, `all()`, `bool()` e proprietà `empty`
- Combinazioni
 - Metodo `combine()`
 - Combiner function

```
In [76]: def combiner(x, y):
.....:     return np.where(pd.isna(x), y, x)
.....:
```

```
In [77]: df1.combine(df2, combiner)
```

```
Out[77]:
```

	A	B
0	1.0	NaN
1	2.0	2.0
2	3.0	3.0
3	5.0	4.0
4	3.0	6.0
5	7.0	8.0

```
In [46]: df.gt(df2)
```

```
Out[46]:
```

	one	two	three
a	False	False	False
b	False	False	False
c	False	False	False
d	False	False	False

```
In [48]: (df > 0).all()
```

```
Out[48]:
```

one	False
two	True
three	False

dtype: bool

```
In [49]: (df > 0).any()
```

```
Out[49]:
```

one	True
two	True
three	True

dtype: bool

```
In [47]: df2.ne(df)
```

```
Out[47]:
```

	one	two	three
a	False	False	True
b	False	False	False
c	False	False	False
d	True	False	False



FUNZIONALITÀ DI BASE

- Aggregazione
 - Metodi `mean()`, `sum()`, `quantile()`
- Cumulazione
 - Metodi `cumsum()`, `cumprod()`
- Sommarizzazione
 - Metodo `describe()`
 - Calcola varie statistiche
- Histogramming
 - Metodo `value_counts()`

```
In [78]: df
Out[78]:
```

	one	two	three
a	1.394981	1.772517	NaN
b	0.343054	1.912123	-0.050390
c	0.695246	1.478369	1.227435
d	NaN	0.279344	-0.613172

```
In [79]: df.mean(0)
Out[79]:
```

one	0.811094
two	1.360588
three	0.187958

dtype: float64

```
In [80]: df.mean(1)
Out[80]:
```

a	1.583749
b	0.734929
c	1.133683
d	-0.166914

dtype: float64

```
In [81]: df.sum(0, skipna=False)
Out[81]:
```

one	NaN
two	5.442353
three	NaN

dtype: float64

```
In [82]: df.sum(axis=1, skipna=True)
Out[82]:
```

a	3.167498
b	2.204786
c	3.401050
d	-0.333828

```
In [118]: data = np.random.randint(0, 7, size=50)
```

```
In [119]: data
Out[119]:
```

```
array([6, 6, 2, 3, 5, 3, 2, 5, 4, 5, 4, 3, 4, 5, 0, 2, 0, 4, 2, 0, 3, 2,
       2, 5, 6, 5, 3, 4, 6, 4, 3, 5, 6, 4, 3, 6, 2, 6, 6, 2, 3, 4, 2, 1,
       6, 2, 6, 1, 5, 4])
```

```
In [120]: s = pd.Series(data)
```

```
In [121]: s.value_counts()
```

```
Out[121]:
```

6	10
2	10
4	9
3	8
5	8
0	3
1	2

```
In [99]: frame.describe()
```

```
Out[99]:
```

	a	b	c	d	e
count	500.000000	500.000000	500.000000	500.000000	500.000000
mean	0.033387	0.030045	-0.043719	-0.051686	0.005979
std	1.017152	0.978743	1.025270	1.015988	1.006695
min	-3.000951	-2.637901	-3.303099	-3.159200	-3.188821
25%	-0.647623	-0.576449	-0.712369	-0.691338	-0.691115
50%	0.047578	-0.021499	-0.023888	-0.032652	-0.025363
75%	0.729907	0.775880	0.618896	0.670047	0.649748
max	2.740139	2.752332	3.004229	2.728702	3.240991



I/O

- Le API di I/O prevedono
 - un set di **reader** che restituiscono un oggetto dopo la lettura di un documento
 - un set di **writer** che scrivono su un documento
- I formati gestiti nativamente sono tantissimi

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	to_orc
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq



I/O

```
c = pd.read_csv("statistiche.csv", sep=";", encoding = 'unicode_escape')  
print(c)
```

```
E:\Program Files (x86)\Microsoft Visual Studio\Shared\Python39_64\...  
Codice Regione ... Densità abitativa (abitanti per Km2)  
0 1 ... 201,12  
1 1 ... 242,64  
2 1 ... 9,97  
3 1 ... 152,66  
4 1 ... 94,99  
... ... ... ...  
8087 20 ... 73,94  
8088 20 ... 130,79  
8089 20 ... 35,71  
8090 20 ... 39,99  
8091 20 ... 30,11  
[8092 rows x 9 columns]  
Press any key to continue . . . _
```



INDICIZZAZIONE GERARCHICA

- **Pandas** supporta indici gerarchici (**multilivello**) e nomi di colonne gerarchici
 - Indici multilivello sono detti anche **multiindici**
- Indice multilivello
 - Tre liste
 - Nomi di livelli
 - Etichette possibili per ogni livello
 - Liste dei vali effettivi per ogni elemento del frame



INDICIZZAZIONE GERARCHICA

- MultilIndex
 - Funzione `from_tuples()`
- Stack
 - Appiattimento di un indice multilivello
 - Con l'aggiunta di colonne multilivello
 - Il frame è più «alto» e più «stretto»
- Pivot
 - Converte un frame in un altro, usando l'indice di una colonna come nuovo indice



DATI MANCANTI

- Funzione **dropna()**
 - Rimuove parzialmente o interamente le colonne o le righe non valide, pulendo il frame
- Funzioni **isnull()** e **notnull()**
 - Per la selezione di colonne nulle o non nulle
- Funzione **fillna()**
 - Inserisce un valore nei dati assenti
- Funzione **replace()**
 - Sostituisce una lista di valori con un'altra



COMBINARE I DATI

- Unione
 - Funzione `merge()`
- Concatenamento
 - Funzione `concat()`
- Cancellazione duplicati
 - Funzione `duplicated()`
- Ordinamento e classificazione
 - Funzioni `sort_index()`, `sort_values()`



COMBINARE I DATI

- Unicità, conteggio e appartenenza
 - Funzione `unique()`
 - Funzione `value_counts()`
 - Funzione `isin()`



TRASFORMAZIONE

- Operazioni aritmetiche
- Aggregazione
 - Suddivisione
 - Frammentazione sulla base di chiavi
 - Aggregazione
 - Su ogni frammento viene eseguita una funzione
 - Ricombinazione
 - I risultati vengono ricombinati in un nuovo oggetto



MAPPATURA

- Funzione **map()**
 - Applica una funzione arbitraria con un solo parametro ad ogni elemento di una colonna
 - Generalmente è una funzione lambda
 - Altamente inefficiente perché interpretata



TABULAZIONE INCROCIATA

- Funzione **crosstab()**
 - Calcola le frequenze dei gruppi e restituisce un frame in cui
 - Le righe sono i valori di variabili categoriche («fattori»)
 - Le colonne sono i valori di frequenza