Constructors - 1

- I costruttori sono metodi che permettono di creare array numpy
- Abbiamo già visto il costruttore principale: array()
- Altri costruttori:

empty(shape[, dtype, order, like])	Return a new array of given shape and type, without initializing entries.
empty_like(prototype[, dtype, order, subok, ...])	Return a new array with the same shape and type as a given array.
eye(N[, M, k, dtype, order, like])	Return a 2-D array with ones on the diagonal and zeros elsewhere.
identity(n[, dtype, like])	Return the identity array.
ones(shape[, dtype, order, like])	Return a new array of given shape and type, filled with ones.
ones_like(a[, dtype, order, subok, shape])	Return an array of ones with the same shape and type as a given array.
zeros(shape[, dtype, order, like])	Return a new array of given shape and type, filled with zeros.
zeros_like(a[, dtype, order, subok, shape])	Return an array of zeros with the same shape and type as a given array.
full(shape, fill_value[, dtype, order, like])	Return a new array of given shape and type, filled with *fill_value*.
full_like(a, fill_value[, dtype, order, ...])	Return a full array with the same shape and type as a given array.

Esempi

```
import numpy as np

empty_arr = np.empty((20), float)
print(empty_arr)

i33 = np.eye(3)
print(i33)

empty_like = np.empty_like(i33)
print(empty_like)

all_ones = np.ones(5)
print(all_ones)
all_zeros = np.zeros(5)
print(all_zeros)
fill_pi = np.full((3,3), np.pi)
print(fill_pi)
```

```
[0.  0.  0.3 1.  0.  0.  1.  1.  1.  1.
1.  1.  1.  0.  0.  1.  0.5 0.
 0.  1. ]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0.]
[[3.14159265 3.14159265 3.14159265]
 [3.14159265 3.14159265 3.14159265]
 [3.14159265 3.14159265 3.14159265]]
```

Constructors - 2

array(object[, dtype, copy, order, subok, ...])	Create an array.
asarray(a[, dtype, order, like])	Convert the input to an array.
asanyarray(a[, dtype, order, like])	Convert the input to an ndarray, but pass ndarray subclasses through.
ascontiguousarray(a[, dtype, like])	Return a contiguous array (ndim >= 1) in memory (C order).
asmatrix(data[, dtype])	Interpret the input as a matrix.
copy(a[, order, subok])	Return an array copy of the given object.
frombuffer(buffer[, dtype, count, offset, like])	Interpret a buffer as a 1-dimensional array.
fromfile(file[, dtype, count, sep, offset, like])	Construct an array from data in a text or binary file.
fromfunction(function, shape, *[, dtype, like])	Construct an array by executing a function over each coordinate.
fromiter(iter, dtype[, count, like])	Create a new 1-dimensional array from an iterable object.
fromstring(string[, dtype, count, like])	A new 1-D array initialized from text data in a string.
loadtxt(fname[, dtype, comments, delimiter, ...])	Load data from a text file.

Esempio - genfromtxt

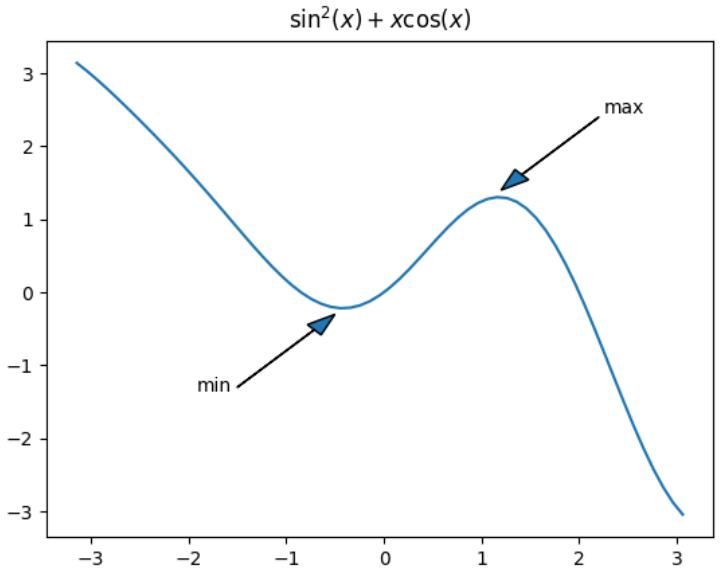
```
import numpy as np
dt = np.dtype([(‘sex', 'U10'), (‘height', 'f8') , (‘weight', 'f8')])
dataset = np.genfromtxt('weight-height.csv',
                        dtype=dt,
                        skip_header=1,
                        delimiter=",",
                        converters={0: lambda s:s[1:-1]},
                        names=["Sesso", "Altezza", "Peso"])
bmi = dataset["Peso"] * lb_to_kg / (dataset["Altezza"] * in_to_mt)**2
```

Constructors - 3

arange([start,] stop[, step,][, dtype, like])	Return evenly spaced values within a given interval.
linspace(start, stop[, num, endpoint, ...])	Return evenly spaced numbers over a specified interval.
logspace(start, stop[, num, endpoint, base, ...])	Return numbers spaced evenly on a log scale.
geomspace(start, stop[, num, endpoint, ...])	Return numbers spaced evenly on a log scale (a geometric progression).

Esempio

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

x = np.arange(-np.pi, np.pi, .1)
y = np.sin(x)*np.sin(x)+x*np.cos(x)


plt.plot(x, y)
plt.title(r"$\sin^2(x)+x\cos(x)$")
plt.arrow(2.2, 2.4, -1, -1,
          length_includes_head=True,
          head_width=0.2)
plt.text(2.25, 2.45, "max")
plt.arrow(-1.5, -1.3, 1, 1,
          length_includes_head=True,
          head_width=0.2)
plt.text(-1.55, -1.35, "min", ha='right')
plt.show()
```



Esercizio 1 – Media Mobile

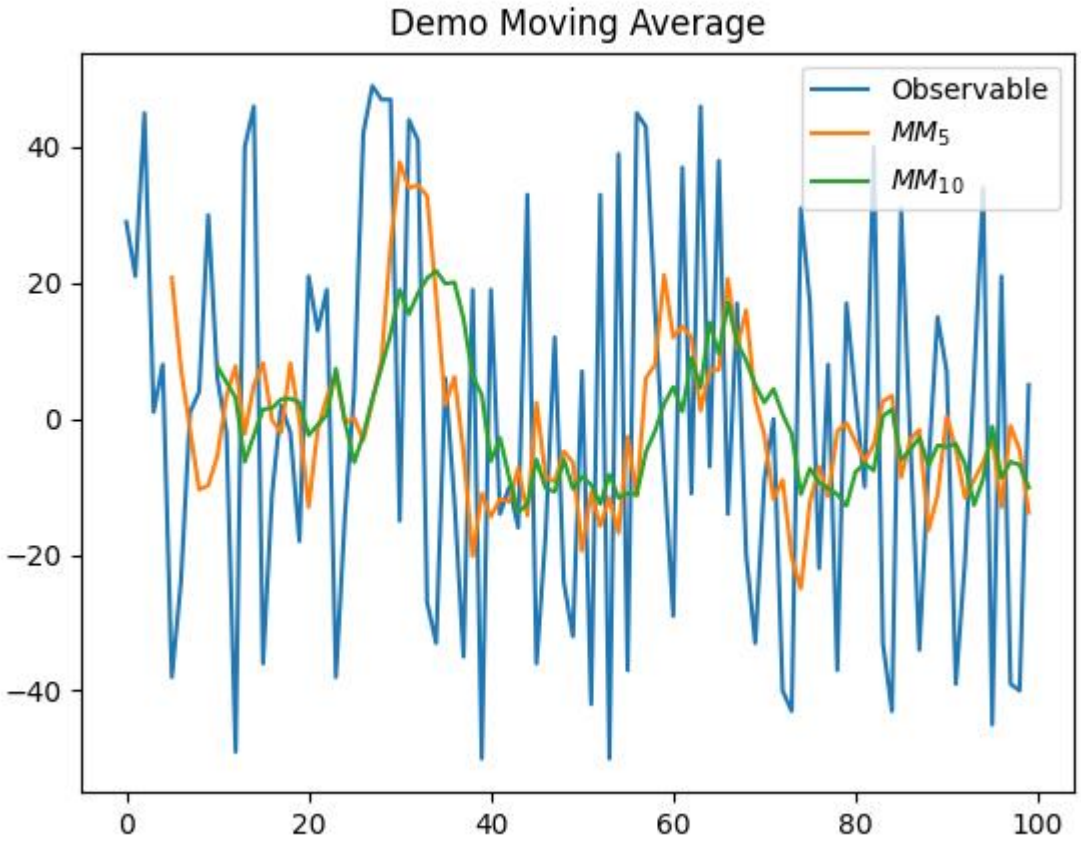
- `numpy` non ha una funzione che calcoli la *media mobile*
- Per fare questo costruiamone una usando le proprietà e gli operatori di `numpy`:
- Vi ricordo che la formula per il calcolo della media mobile su una serie storica $\{y_t\}$ per $t \in [1, T]$ su un periodo $m \in \mathbb{N}^+$ è data da:

$$\underbrace{\overline{y_t}}_{m} = \frac{1}{m} \sum_{i=0}^{m-1} y_{t-i} \;\; \forall t \in [m, T]$$

Esercizio 1 - Svolgimento

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

def moving_average(a, k):
    ret = [a[t-k:t].sum() for t in range(k, len(a))]
    return np.array(ret)/k

min_value = -50
max_value = 50
obs = 100
range1 = 5
range2 = 10

time_series = np.random.randint(min_value, max_value, obs)
mm1_time_series = moving_average(time_series, range1)
mm2_time_series = moving_average(time_series, range2)

plt.title("Demo Moving Average")
plt.plot(range(100), time_series)
plt.plot(range(range1, 100), mm1_time_series)
plt.plot(range(range2, 100), mm2_time_series)
plt.legend(["Observable", f"$MM_{range1}$", f"$MM_{{{range2}}}$"])
plt.show()
```

Esercizio 1 - Risultati



Materiale

- https://numpy.org
- https://python.org
- https://matplotlib.org
- https://github.com/rougier/scientific-visualization-book
- https://data-flairing.training

pandas

pandas

- *pandas* è una libreria Python contenente strutture e strumenti di dati di alto livello che sono stati creati per aiutare i programmatori Python a eseguire analisi sui dati.
- Lo scopo finale di pandas è di aiutare a scoprire rapidamente le informazioni nei dati, informazioni intese come significato sottostante.
- L'utilizzo di *pandas* nell'ambiente del *datascientist* Python è **fondamentale**
- *pandas* è già inclusa nell'ambiente Anaconda

pandas

- Oggetti `Series` ed `DataFrame` veloci ed efficienti per la manipolazione dei dati con indicizzazione integrata
- Allineamento intelligente dei dati tramite indici ed etichette
- Gestione integrata dei dati mancanti
- Funzionalità per la conversione di dati disordinati in dati ordinati (riordino)
- Strumenti integrati per la lettura e la scrittura di dati tra strutture e file di dati in memoria, database e servizi Web
- Capacità di elaborare i dati memorizzati in molti formati comuni come CSV, Excel, HDF5 e JSON

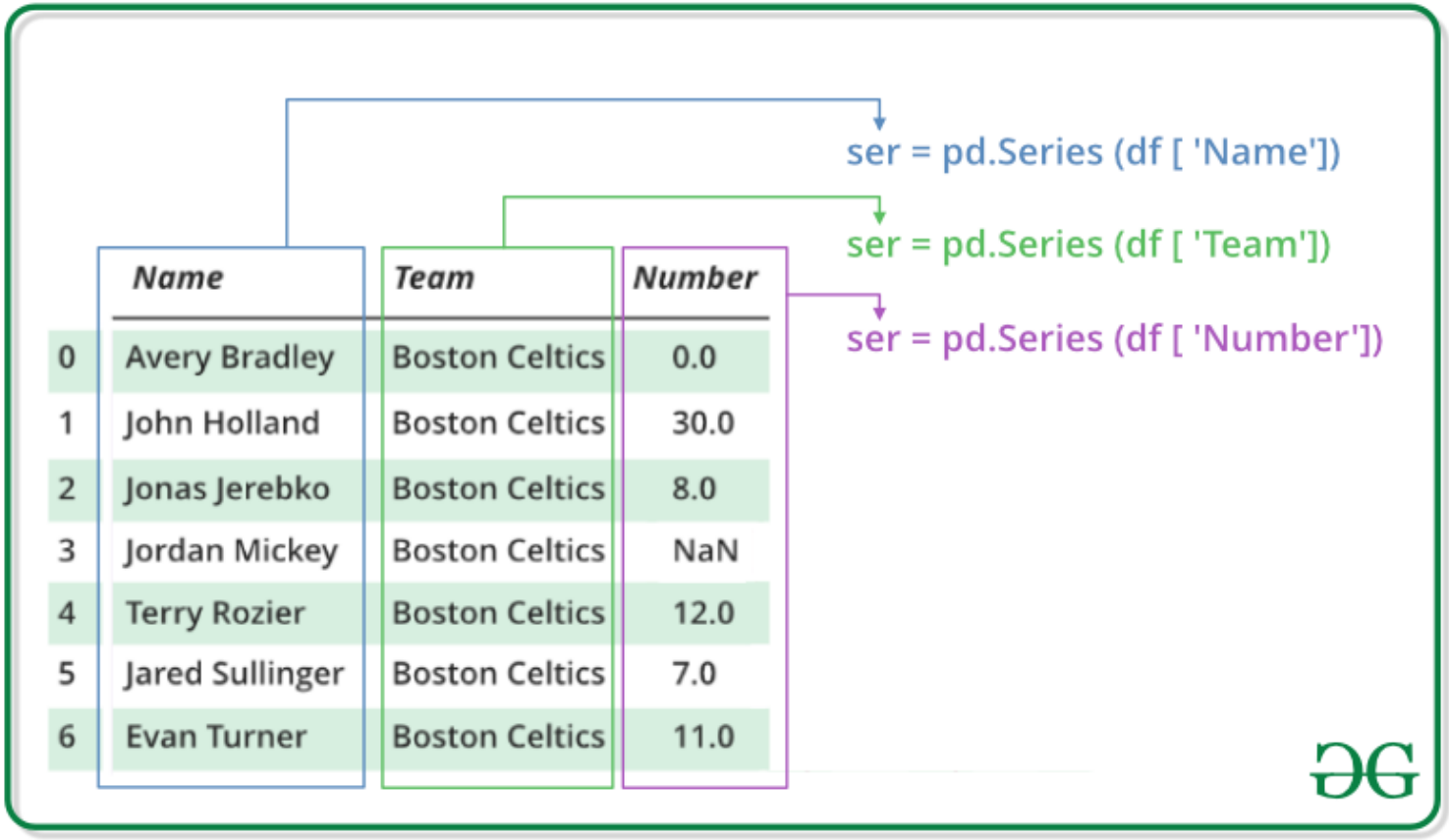
pandas

- Smart slicing basato su etichette, indicizzazione elaborata e sottoinsieme di set di dati di grandi dimensioni
- Le colonne possono essere inserite ed eliminate dalle strutture di dati per la mutabilità delle dimensioni
- Aggregazione o trasformazione di dati con una potente funzione di raggruppamento per eseguire divisioni di applicazione divisa su set di dati

pandas

- Core
  - I due componenti principali di pandas sono `Series` e `DataFrame`.
  - `Series` è essenzialmente una colonna e un `DataFrame` è una tabella bidimensionale composta da una raccolta di `Series`

Series



	Name	Team	Number
0	Avery Bradley	Boston Celtics	0.0
1	John Holland	Boston Celtics	30.0
2	Jonas Jerebko	Boston Celtics	8.0
3	Jordan Mickey	Boston Celtics	NaN
4	Terry Rozier	Boston Celtics	12.0
5	Jared Sullinger	Boston Celtics	7.0
6	Evan Turner	Boston Celtics	11.0

Series

- Una `Series` pandas è un array etichettabile unidimensionale in grado di contenere dati di qualsiasi tipo (intero, stringa, float, oggetti Python, ecc.).
- Le etichette degli assi sono chiamate collettivamente *indici*.
- Una `Series` non è altro che una colonna in un foglio Excel.

Costruttori

- I dati possono essere molte cose diverse:
  - un dict Python
  - un ndarray
  - un valore scalare (come 5)
- L'indice passato è un elenco di etichette degli assi.
  - Pertanto, questo si differenzia in alcuni casi a seconda di quali dati sono:
    - ndarray
    - dict
    - valore scalare

```
s = pd.Series(data, index=index)
```

Series

- Creare una Series

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
array_np = np.array([1, 2, 3])
ser = pd.Series(array_np)
print(ser)
```

```
0    1
1    2
2    3
dtype: int32
```

In [3]:
```
list_py = ['a', 1, True]
ser = pd.Series(list_py)
ser = ser.rename("NOME")
print(ser)
```

```
0       a
1       1
2    True
Name: NOME, dtype: object
```

In [4]:
```
dict = {'A' : 10,
        'B' : 20,
        'C' : 30}

ser = pd.Series(dict)

print(ser)
```

```
A    10
B    20
C    30
dtype: int64
```

Series

- Accesso
  - L'accesso ad un elemento della serie è possibile usare l'indice in qualunque forma

```
dict = {'A': 10,
        'B': 20,
        'C': 30}
ser = pd.Series(dict)
print(ser['A'])
```

```
list_py = ['a', 1, True]
ser = pd.Series(list_py)
print(ser[2])
```

Series

- `Series.iloc[n]`
  - Ritorna la posizione relativa e non quella della etichetta
- `Series.loc[n]`
  - Ritorna la posizione indicizzata dall'etichetta
- `Series[n]`
  - Ritorna la posizione indicizzata dalla posizione assoluta

`.iloc[]`

- `.iloc[]` è principalmente basato sulla posizione intera (da 0 a lunghezza-1 dell'asse), ma può anche essere utilizzato con un array booleano.
- Valore ammessi:
  - Intero
  - Una lista di interi
  - Un oggetto slice (es. 1:3)
  - Una array booleano
  - Una *callable function* con un argomento (la serie)

`.loc[]`

- Accede a un gruppo di righe e colonne per etichetta o per una matrice booleana.
- `.loc[]` è principalmente basato su etichette, ma può anche essere utilizzato con un array booleano.
- Gli input consentiti sono:
  - Una etichetta singola
  - Una lista di etichette
  - Uno slice (es. 'a':'c')
  - Un array booleano
  - Una funzione *callable*

.iloc[] vs .loc[]

	loc	iloc
A value	A single label or integer e.g. `loc[A]` or `loc[1]`	A single integer e.g. `iloc[1]`
A list	A list of labels e.g. `loc[[A, B]]`	A list of integers e.g. `iloc[[1,2,3]]`
Slicing	e.g. `loc[A:B]`, A and B are included	e.g. `iloc[n:m]`, n is included, m is excluded
Conditions	A bool Series or list	A bool list
Callable function	`loc[lambda x: x[2]]`	`iloc[lambda x: x[2]]`

Series

```
list_py = [2, 3, 4, 5, 6, 7, 8]
ser = pd.Series(list_py)
ser2 = ser[4:]
print(ser2)
print("INDICE [5]")
print(ser2[5])
print("INDICE loc[5]")
print(ser2.loc[5])
print("INDICE iloc[1]")
print(ser2.iloc[1])
print("1) INDICE [5:6]")
print(ser2[5:6])
print("2) INDICE loc[5:6]")
print(ser2.loc[5:6])
print("3) INDICE iloc[1:2]")
print(ser2.iloc[1:2])
```

```
4  6
5  7
6  8

INDICE [5]
7
INDICE loc[5]
7
INDICE iloc[1]
7
1) INDICE [5:6]
Series([], dtype: int64)
2) INDICE loc[5:6]
5  7
6  8

3) INDICE iloc[1:2]
5  7
```

Series

- Operazioni su Series

```
In [39]:
1 data = pd.Series([5, 2, 3,7], index=['a', 'b', 'c', 'd'])
2 data1 = pd.Series([1, 6, 4, 9], index=['a', 'b', 'd', 'e'])
```

Usando la versione funzionale è possibile descrivere cosa fare in caso di chiavi mancanti

```
In [40]:
1 data.add(data1, fill_value=0)

Out[40]:
a     6.0
b     8.0
c     3.0
d    11.0
e     9.0
dtype: float64
```

```
In [41]:
1 data.sub(data1, fill_value=0)

Out[41]:
a    4.0
b   -4.0
c    3.0
d    3.0
e   -9.0
dtype: float64
```

Usando la versione operazionale le chiavi mancanti producono un NaN

```
In [42]:
1 data-data1

Out[42]:
a    4.0
b   -4.0
c    NaN
d    3.0
e    NaN
dtype: float64
```

Pandas vs Numpy

- Le Series *pandas* anche se a prima vista molto simili agli array di numpy sono in realtà assai diversi
- Ad esempio un Array numpy è un oggetto omogeneo mentre la Series pandas è eterogeneo:

Pandas vs Numpy

```
import numpy as np
import pandas as pd

vett1 = [1, 1.4, True, 'ciao']

np_vett1 = np.array(vett1)
print(f"Tipo dati array numpy: {np_vett1.dtype}")
for i in range(len(np_vett1)):
    print(f"Dato originale {type(vett1[i])}" +
          f" tipo derivato numpy {type(np_vett1[i])}")

pd_vett1 = pd.Series(vett1)
print(f"Tipo dati array pandas: {pd_vett1.dtype}")

for i in range(len(pd_vett1)):
    print(f"Dato originale {type(vett1[i])}" +
          f" tipo derivato pandas {type(pd_vett1[i])}")
```

Pandas vs Numpy

```
Tipo dati array numpy: <U32
Dato originale <class 'int'> tipo derivato numpy <class 'numpy.str_'>
Dato originale <class 'float'> tipo derivato numpy <class 'numpy.str_'>
Dato originale <class 'bool'> tipo derivato numpy <class 'numpy.str_'>
Dato originale <class 'str'> tipo derivato numpy <class 'numpy.str_’>

Tipo dati array pandas: object
Dato originale <class 'int'> tipo derivato pandas <class 'int'>
Dato originale <class 'float'> tipo derivato pandas <class 'float'>
Dato originale <class 'bool'> tipo derivato pandas <class 'bool'>
Dato originale <class 'str'> tipo derivato pandas <class 'str'>
```

Series

- Conversioni

In [77]:
```
1 data = pd.Series([6, 5.2, '7', 7])
2 data1 = pd.to_numeric(data)
3 print("PRIMA", data.dtypes, "DOPO", data1.dtypes)
```

```
PRIMA object DOPO float64
```

In [78]:
```
1 data2 = data1.astype(int)
2 print("PRIMA", data1.dtypes, "DOPO", data2.dtypes)
```

```
PRIMA float64 DOPO int32
```

In [81]:
```
1 data3 = pd.Series(["2020-01-01", "2020-01-02", "2020-01-03", "NP"])
2 data4 = pd.to_datetime(data3, errors='coerce')
```

In [82]:
```
1 data4
```

Out[82]:
```
0   2020-01-01
1   2020-01-02
2   2020-01-03
3          NaT
dtype: datetime64[ns]
```