Lezione 6: Pandas

30 Giugno 2025

Pandas: ancora più flessibilità

- Le strutture dati che introduce possono essere viste come delle estensioni agli array di Numpy, aggiungendo
  - un nuovo metodo di indicizzazione, pensato per lavorare con dati tabulari
  - supporto per nuovi tipi
  - supporto per le operazioni sulle serie di stringhe
  - supporto per molti tipi di file da cui caricare dati
  - molto altro…

Limiti degli array strutturati (1/2)

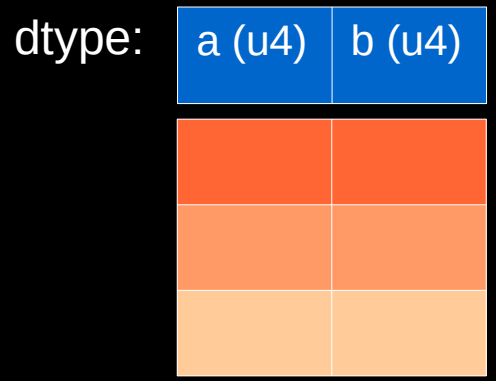
- Gli array strutturati di numpy hanno un dtype strutturato che non può essere aggiornato
  - Non potete semplicemente “aggiungere una colonna”: bisogna ricostruire con un nuovo dtype
  - Per aggiungere una riga, bisogna usare `np.concatenate`, facendo una copia
- Le “righe” non hanno un’etichetta, possiamo solo selezionarle in base alla posizione

Limiti degli array strutturati (2/2)

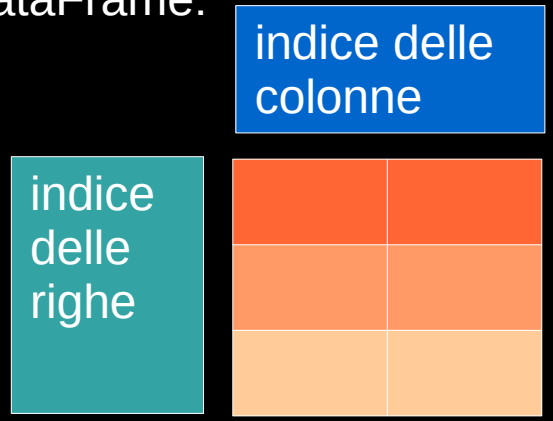
```
>>> a = np.array([(1,2), (4,6)],
…     dtype=[('a', 'u4'), ('b', 'u4')])
>>> a['c'] = a['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: no field of name c
>>> np.concatenate((a, a))
array([(1, 2), (4, 6), (1, 2), (4, 6)],
dtype=[('a', '<u4'), ('b', '<u4')])
```

I DataFrame di Pandas

Array strutturato:



DataFrame:



Gli indici del DataFrame possono essere visti come chiavi di due dizionari: uno per le colonne e uno per le righe

Installazione

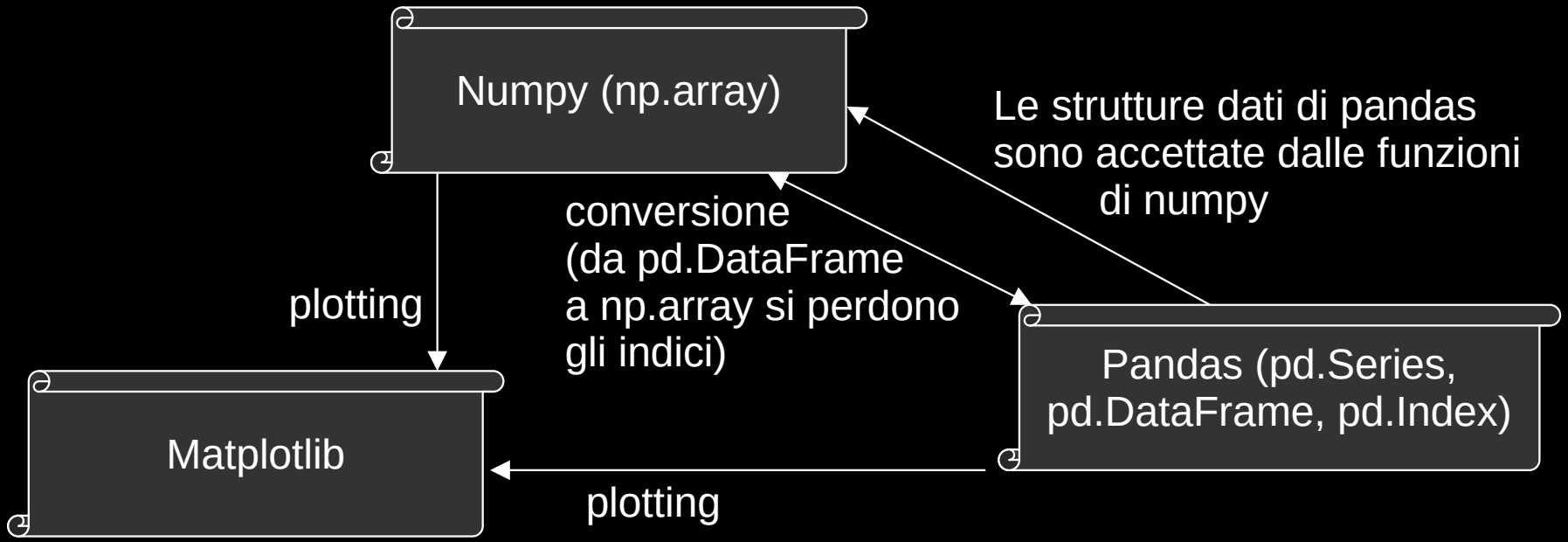
- Come al solito, possiamo usare pip o apt

```
$ pip install pandas
$ sudo apt install python3-pandas
```

- Useremo sempre l'alias pd per pandas

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

Integrazione delle librerie



Funzioni per caricare da file

Pandas offre molte funzioni per caricare DataFrame da diversi tipi di file.
Per vederle, digitare `pd.read_<TAB>` sulla shell di Python

- pd.read_csv → valori separati da “,” o altro
  - Pandas riconosce quando sono presenti i nomi delle colonne alla prima riga
- pd.read_excel → File di excel (estensione .xls)
  - Funziona anche con i file di LibreOffice (estensione .ods)
- pd.read_json
- pd.read_xml
- pd.read_html → Tabelle delle pagine web
  - Restituisce una lista di DataFrame, nell’ordine in cui le tabelle sono presentate nella pagina

Funzioni per scrivere su un file

- Dopo aver finito di lavorare con il nostro DataFrame, chiamiamolo `df`
  - Per scrivere su un file e salvare `df`, pandas offre un'altra famiglia di funzioni (`df.to_<TAB>`)
  - in `df.to_csv(nomefile)` si può specificare il separatore, se scrivere l'indice delle colonne, se scrivere l'indice delle righe, ecc…

Carichiamo un DataFrame

```
>>> g = pd.read_html("https://crawl.develz.org/tournament/0.32/combo-scoreboard.html")
>>> len(g) # in realtà ne abbiamo caricate 3
3
>>> g = g[2] # prendiamo la terza
>>> g.info()
class 'pandas.core.frame.DataFrame'>
RangeIndex: 665 entries, 0 to 664
Data columns (total 10 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Score     665 non-null    int64
 1   Player    665 non-null    object
 2   Combo     665 non-null    object
 3   God       617 non-null    object
 4   Title     665 non-null    object
 5   XL        665 non-null    int64
 6   Turns     665 non-null    int64
 7   Duration  665 non-null    object
 8   Runes     413 non-null    float64
 9   Date      665 non-null    object
dtypes: float64(1), int64(3), object(6)
memory usage: 52.1+ KB
```

Finito il torneo, sono stati calcolati i massimi punteggi per ogni combinazioine (colonna Combo) razza-background

Esempio: la combo Minotaur Fighter avrà l'abbreviazione "MiFi" nella colonna Combo

Accedere ai valori di un DataFrame

- Abbiamo molti modi per accedere ai valori di una tabella. Partiamo da quello che appare più famigliare:
  - `g['Combo']`: stiamo usando l'indice `g.columns` come se fosse un insieme di chiavi di dizionario
    - Abbiamo ottenuto una Series! (prossima slide)
  - Possiamo creare nuove colonne con l'assegnamento:

    ```
    >>> g['levelprogress'] = g['XL']/27
    ```
  - Possiamo cancellare le colonne con il solito comando `del`:

    ```
    >>> del g['God']
    ```

pandas.Series

- Una series è molto simile ad un array monodimensionale di numpy
- Ha un solo indice

```
>>> a = pd.Series([1,2,3,4])
>>> a.index = list("abcd")
```

numpy array:

Series:

Operazioni su Series (1/3)

```
print(a[1::-1]) # slices
print(a * 2) # broadcasting di 2
```

- Se per fare un'operazione tra due array di numpy questa veniva applicata sugli elementi con la stessa posizione, per le Series questa viene applicata sugli elementi con la stessa etichetta nell'indice:

```
print(a + a[1::-1])
```

a	b	c	d
2.0	4.0	NaN	NaN

Per vedere quali valori sono NaN, usare `a.isna()`

Operazioni su Series (2/3)

- Così come abbiamo usato le posizioni, possiamo usare anche le etichette dell'indice

```
>>> a['b']
2
>>> a['c':'b':-1] # stop è incluso!
c    3
b    2
dtype: int64
```

Solo quando uso gli indici nelle slice. Se uso le slice con le posizioni si ha il solito comportamento (stop escluso)

Operazioni su Series (3/3)

- Ci sono le solite indicizzazioni avanzate come Numpy, con l'unica aggiunta dell'utilizzo dell'indice

```
>>> a[['a','c','b']] == a[[0,2,1]]
a    True
c    True
b    True
dtype: bool
>>> a[a<3] # ricordare che (a<3) è una Series di bool
a    1
b    2
dtype: int64
```

Torniamo alla tabella

- Fino ad ora abbiamo usato l'indice delle colonne, `g.columns`
- Prima di usare l'indice delle righe (`g.index`), selezioniamo una colonna che sostituisca l'indice che Pandas ha creato come default:

```
>>> g.index
RangeIndex(start=0, stop=665, step=1)
>>> g = g.set_index('Date')
```

Ci siamo quasi...

- Se controlliamo il tipo di dato (dtype) di g.index, vedremo che è object:
  - object è un oggetto generico, quindi non offre molte funzionalità
  - ma noi abbiamo data-ora come indice, e vogliamo sfruttare le funzionalità relative a questo tipo di dato

```
>>> g.index = g.index.astype("datetime64[ns]")
>>> # pandas ha riconosciuto un formato data-ora in tutte le
>>> # etichette dell'indice, altrimenti avrebbe mostrato un errore
```

Altre modalità di accesso per la tabella

- Possiamo accedere alle righe usando le slices:

```
# tutte le partite che hanno ottenuto il punteggio
# tra il 10 e il 12 Settembre
print(g['2024-09-10':'2024-09-12'])
# dalla riga 5 alla riga 7
print(g[5:8])
```

E se sulle righe avessimo avuto un indice di interi?

- Oppure l'indicizzazione avanzata con array (o series) di booleani

```
print(g[g['Score'] > 40000000])
# equivalentemente:
print(g.query('Score > 40000000'))
```

Troppa confusione

- Il modo di indicizzare che abbiamo usato accetta troppi oggetti diversi tra le parentesi quadre:
  - Slices di posizioni/etichette (ambiguo...), ottenendo le righe in un DataFrame
  - Etichette, liste di etichette, ottenendo le colonne
  - Array di booleani, ottenendo nuovamente le righe…
- Oltre ad essere confusionario per noi, pandas deve ogni volta distinguere tra molti casi possibili, rallentando gli accessi

Il problema, in sintesi

- Vogliamo distinguere due casi:
  - Quando stiamo selezionando sulle colonne
  - Quando stiamo selezionando sulle righe
- Per ciascuno dei due casi, vogliamo ulteriormente distinguere:
  - Quando usiamo le posizioni
  - Quando usiamo le etichette degli indici

Soluzione proposta: loc e iloc

- Per evitare i problemi, sono consigliate le seguenti modalità di accesso:
  - g.loc[riga (indice)]
  - g.loc[riga (indice), colonna (indice)]
  - g.iloc[riga (posizioni)]
  - g.iloc[riga (posizioni), colonna (posizioni)]
- E gli array di booleani? Non dovrebbero essere un altro caso a parte?
  - No, gli array di booleani possono stare ovunque (Nessuno farebbe mai un indice di booleani… non avrebbe senso)
  - Basta che abbiano la lunghezza giusta

loc e iloc: esempi

- Punteggi migliori ottenuti ad Agosto:

```
print(g.loc['2024-08', 'Score'])
```

Notare che non è servito specificare la slice per i giorni del mese

- Combo (terza colonna) con cui sono stati ottenuti punteggi maggiori di `40000000`

```
print(g.iloc[(g['Score'] > 40000000).values, 2])
```

Combo separate!

- In g abbiamo la colonna “Combo”, che consiste nella coppia (razza, background)
- Essendo pandas molto abile nel trattare con i dati testuali, possiamo separarla in 2 colonne:

```
>>> g['Race'] = g['Combo'].str[:2]
>>> g['Back'] = g['Combo'].str[2:]
>>> del g['Combo']
```

Abilità sbloccate

- Ora possiamo ricavare i punteggi massimi per razza e per background:

```
print(g.groupby("Race")["Score"].max())
print(g.groupby("Back")["Score"].max())
```

- Ricordando che questa è una tabella di punteggi massimi per combo, proviamo a creare una nuova tabella

Tabella pivot (1/3)

- Creiamo una tabella che ha come indice delle righe le razze, mentre i backgrounds compongono l'indice delle colonne
- Abbiamo la garanzia che data una combo, può corrispondere al più un solo punteggio massimo (esistono le combo che non sono mai state giocate)
  - Significa che in ogni cella `a.loc[razza, back]` posso inserire il punteggio massimo per quella combo

```
a = g.pivot(index='Race', columns='Back',
            values='Score')
```

Tabella pivot (2/3)

- Con il DataFrame appena creato, posso ottenere gli stessi risultati di prima:

```
print(a.max(axis=1)) # per razza
print(a.max()) # per background (axis = 0)
```

- Ma anche qualcosa in più, come le combinazioni migliori per razza o per background:

```
print(a.idxmax(axis=1))
print(a.idxmax())
```

Usando l'asse 0, si calcola il massimo per ogni colonna. Con 1, per ogni riga. Il parametro axis può essere trovato in molte funzioni di Numpy, oltre che a Pandas

Tabella pivot (3/3)

- Ma cosa succede se ad una coppia riga-colonna corrispondono più valori?
- pd.pivot(…) Vi darà un errore…
- Questi valori multipli possono essere raccolti da Pandas in una Sequenza, per poi applicare una funzione di aggregazione (max, len, min, ...) e ottenere un valore unico
- Bisogna usare `pd.pivot_table` per usare le funzioni di aggregazione

Dalla demo aggiornata con Pandas

```
def tobest(data):
    return data.pivot_table(columns='race',
           index='back', values='score',
           aggfunc='min')

def tocount(data):
    return data[['race',
    'back']].pivot_table(columns='race',
    index='back', aggfunc=len, fill_value=0)
```

Prima di passare alla demo...

- Se avete un DataFrame `df`, potete ottenere un DataFrame trasposto (scambiare righe e colonne) con `df.T`
- L'operazione inversa di `df.set_index` è `df.reset_index`, che sposta l'indice delle rige su una colonna, oppure lo resetta senza creare colonne con `drop=True`
- Si può convertire un DataFrame in un array bidimensionale di numpy:

```
d = df.to_numpy()
```

- Si può creare un DataFrame da un dizionario:

```
f = pd.DataFrame({'a' : list('qwert'), 'b': range(5)})
```