

# Preparazione dei Dati in Python con Pandas (parte 1)

## Programmazione di Applicazioni Data Intensive

Laurea in Ingegneria e Scienze Informatiche  
DISI – Università di Bologna, Cesena

Proff. Gianluca Moro, Roberto Pasolini  
*nome.cognome@unibo.it*



## Outline

### Prima lezione

- Dati relazionali
  - tipi di dato
- Serie pandas
  - struttura e creazione
  - selezione di dati
  - operazioni
- DataFrame pandas
  - creazione, import da CSV
  - selezione di dati
  - statistiche
  - ordinamento

### Seconda lezione

- Operazioni avanzate sui DF
  - join
  - indici a più livelli
  - pivoting
  - raggruppamento
- Database relazionali
  - Python DB API
  - esecuzione query da pandas
- Creazione di grafici con matplotlib



## Preparazione dei Dati

- L'analisi dei dati è un'attività scientifica per comprendere e predire fenomeni di interesse aziendale, sociale, di ricerca ...
- Un processo di analisi di dati prevede diversi passaggi:
  - raccolta dei dati da una o più sorgenti (database, servizi Web, ...)
  - comprensione della struttura e del significato dei dati
  - trasformazione e pulizia dei dati in una forma utile alle fasi successive
  - estrazione di conoscenza dai dati (statistiche, modelli predittivi, ...)
  - validazione e interpretazione della conoscenza estratta
  - deployment della conoscenza (es. predizioni) in applicazioni
- I primi tre punti costituiscono la fase di preparazione dei dati, che sarà l'oggetto di questa lezione



## Preparazione dei Dati in Python

- Le caratteristiche di base di Python lo rendono adatto per operazioni di analisi (in particolare estrazione e pulizia)
  - semplice da imparare ed usare
  - utilizzabile sia in modo interattivo che per script e programmi completi
- Negli anni sono state sviluppate diverse librerie che rendono Python un ambiente completo di analisi dati
  - Python è sempre più usato in sostituzione di software ad hoc come R e Matlab, in particolare da chi ha esperienza nella programmazione
- Abbiamo visto come usare NumPy per la rappresentazione e l'elaborazione di dati in forma di vettori e matrici



## Dati Tabulari

- Dati oggetto d'analisi sono comunemente reperiti o convertiti in forma **relazionale** o **tabulare**, cioè in forma di tabella dove
  - ogni **riga** rappresenta un'**osservazione o istanza**, ossia uno degli elementi su cui si compie l'analisi (una persona, un prodotto, ...)
  - ogni **colonna** è una **variabile, attributo o feature** che caratterizza ciascun oggetto; ciascuna contiene valori di un tipo specifico
- Esempi di dati in forma relazionale sono ad es. tabelle, viste e risultati di query in un database relazionale (SQL)

nome	cognome	data di nascita	sex	titolo di studio	num. acquisti
Mario	Rossi	18/6/82	M	laurea	8
Maria	Verdi	9/3/88	F	diploma	12
...	...	...	...	...	...



## Tipi di Dati: Nominali e Ordinali

- Colonne contenenti **testo arbitrario** (es. nome, cognome, testo di una recensione, ...) sono di solito ignorate nell'analisi
  - salvo usare metodi per il trattamento dei testi che vedremo poi ...
- Una colonna di tipo **nominale** (o **categorico**) contiene valori a scelta in un insieme finito
  - es. sesso (M/F), regione di residenza, categoria di prodotto, ...
- Una colonna di tipo **ordinale** è simile, ma definisce inoltre una **relazione d'ordine** tra i valori possibili
  - un valore può essere **maggiore o minore** di un altro
  - es. titolo di studio (diploma < laurea triennale < laurea magistrale)

nome	cognome	data di nascita	sex	titolo di studio	num. acquisti
Mario	Rossi	18/6/82	M	laurea	8
Maria	Verdi	9/3/88	F	diploma	12
...	...	...	...	...	...



## Tipi di Dati: Intervallo e Ratio

4. In colonne di tipo **intervallo**, oltre ad esistere una relazione d'ordine, è possibile **quantificare le differenze tra valori**
- su questi valori hanno senso somma, sottrazione e calcolo della media
  - es. date, temperature in °C e Fahrenheit, ...
5. Le colonne di tipo **ratio** sono quelle in cui, in aggiunta alle proprietà delle intervallo, **esiste un valore "zero" significativo**
- solo su questi valori hanno senso anche moltiplicazione e divisione, ad es. dire che un valore è il doppio di un altro
  - es. quantità fisiche (massa, durata, ...), numero di acquisti, prezzo, fatturato, ...
  - temperatura in gradi Kelvin (zero assoluto)

nome	cognome	data di nascita	sex	titolo di studio	num. acquisti
Mario	Rossi	18/6/82	M	laurea	8
Maria	Verdi	9/3/88	F	diploma	12
...	...	...	...	...	...



## Attributi e Proprietà Numeriche

valore che si presenta con maggiore frequenza

Tipo di attributo		Descrizione	Esempi	Operazioni
Categorico (qualitativo)	nominale	I suoi valori sono semplicemente nomi diversi, ovvero forniscono soltanto l'informazione sufficiente a distinguere un oggetto da un altro: <b>= e ≠</b>	codici postali, matricole, colore degli occhi, sesso	Moda, entropia, contingenza, correlazione, test $\chi^2$
	ordinale	I suoi valori forniscono informazione sufficiente per ordinare gli oggetti: <b>&lt;, ≤, ≥ e &gt;</b>	Durezza di minerali, valutazioni di qualità non numeriche (sufficiente, buono, ottimo)	Mediana, percentili, correlazioni di rango, ...
Numerico (quantitativo)	intervallo	Sono significative somme e sottrazioni, quindi <b>esiste</b> un'unità di misura: <b>+ e -</b>	Date di calendario, temperature in centigradi e fahrenheit	Media, deviazione standard, correlazione di Pearson, test t e F
	Rapporto ("ratio")	Sono significative anche le operazioni di prodotto e divisione: <b>* e /</b>	Temperature in kelvin, quantità monetarie, conteggi, masse, lunghezze, ...	Media geometrica, media armonica, variazione percentuale

La contingenza fornisce informazioni sull'indipendenza o dipendenza tra variabili, ma non sulla forza del legame. La correlazione, invece, fornisce informazioni sulla forza e direzione del legame lineare.



## Formato CSV (*Comma Separated Values*)

- CSV è il formato più comune per rappresentare dati tabulari su semplici file di testo
- Ogni riga del file costituisce una riga della tabella, le colonne in ogni riga sono separate da un carattere specifico
  - spesso virgola, punto e virgola oppure tabulazione; in quest'ultimo caso i file sono talvolta detti TSV (*Tab Separated Values*)
- La prima riga del file di solito contiene i nomi delle colonne  
 Nome;Cognome;Data nascita;Sesso;Titolo studio;N. acquisti  
 Mario;Rossi;18/6/82;M;laurea;8  
 Maria;Verdi;9/3/88;F;diploma;12
- In CSV non sono riportati esplicitamente i tipi delle colonne: sono dedotti o vanno specificati quando si carica il file



## pandas

- **pandas** è una libreria Python di uso comune che offre svariate strutture dati e funzionalità per l'analisi di dati strutturati
  - strutture dati “smart” per tabelle relazionali, serie temporali, ...
  - selezione dati in stile NumPy per etichetta, posizione, condizione, ...
  - operazioni avanzate di ordinamento, raggruppamento, *join*, ...
  - import/export dati da/a file CSV, Excel, database relazionale, ...
  - funzioni per la generazione rapida di grafici (basati su matplotlib)
- Per convenzione il package **pandas** si importa con nome “pd”  

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



# Serie

- Una **serie** (**Series**) è un **vettore di valori dello stesso tipo**, a ciascuno dei quali è **associata un'etichetta**
  - i tipi supportati, sia per i valori che per le etichette, sono quelli di NumPy (**float64**, **int64**, ... <https://docs.scipy.org/doc/numpy/user/basics.types.html>)
  - una serie è in pratica un **ndarray** a una dimensione (un vettore) con un'etichetta associata ad ogni elemento
- L'**indice** di una serie (attributo **index**) è la **sequenza delle etichette** associate ai valori
  - anche l'indice è un **ndarray**, che può essere di tipo diverso dai valori
  - le etichette sono spesso **identificatori univoci** di tipo numerico o stringa
    - ad es. per dati estratti da un DB possono essere le chiavi primarie
    - *(non è obbligatorio che siano univoche, ma molte operazioni richiedono che lo siano)*

Applicazioni Data Intensive - G. Moro, R. Pasolini - DISI, Università di Bologna, Cesena

11



## Esempi di Serie

- Una serie può rappresentare ad esempio:
  1. il **numero di abitanti** (in migliaia) di ciascuna **città**
  2. il **numero di ordini effettuati** da ciascun **utente** di un ecommerce
  3. il **valore di chiusura giornaliero** di un titolo azionario
- In tutti i casi abbiamo:
  - un insieme di **valori** che rappresentano i **dati** oggetto dell'analisi, di cui possiamo ad es. estrarre una media
  - le **etichette** che identificano ciascun singolo valore e indicano a cosa si riferisce
- È in pratica simile ad un dizionario Python (etichette = chiavi), ma con molte più funzioni...

1

Roma	2.617
Milano	1.242
Napoli	962
...	...

2

alice82	8
bob86	12
caro190	7
...	...

3

2018-09-01	535,43
2018-09-02	532,12
2018-09-03	536,28
...	...



## Creare una Serie

- Il costruttore di `Series` accetta i valori della serie e come attributo `index` opzionale le etichette corrispondenti
  - se non specificato, l'indice è la sequenza di interi da 0 a N-1, così che ogni elemento sia etichettato dalla posizione (come in liste e array)

```
>>> ser = pd.Series([ 4 , 7 , -5 , 3 ],
...                  index=["d", "b", "a", "c"])
```

```
>>> ser
```

```
d      4
b      7
a     -5
c      3
```

```
dtype: int64
```

In fondo alla serie è stampato il tipo dei valori (in seguito sarà omesso per brevità)

d	4
b	7
a	-5
c	3

- I valori e le etichette sono passati come sequenze di valori: sono validi tuple, liste e array NumPy



## Tipi di Dato

- Quando si crea una serie, come per gli `ndarray`, è possibile specificare il tipo di dato col parametro `dtype`
- I tipi di dati utilizzati più comunemente sono quelli numerici
  - i tipi `np.floatN` memorizzano numeri a virgola mobile
  - `np.intN`, `np.uintN` memorizzano numeri interi con o senza segno
  - $N$  è il numero di bit usati, pari a 8, 16, 32 o 64
- Altri tipi di dato includono
  - `bool`: valori booleani
  - `datetime64`, `timedelta64`: timestamp e intervalli di tempo
  - `object`: generici oggetti Python, usato principalmente per stringhe
- Usare i tipi di dati appropriati è importante per ottimizzare l'uso di memoria, in particolare su dataset molto grandi



## Dati Categoricali

- Una serie di tipo categorico (un valore in un insieme finito) può essere rappresentata semplicemente con stringhe

```
>>> weather_str = pd.Series(["Sunny", "Sunny",
                             "Cloudy", "Rainy", "Rainy", "Cloudy"])
>>> weather_str
0    Sunny
1    Sunny
[...]
5    Cloudy
dtype: object
```

- Per efficienza pandas offre però un tipo di dato “category”, che memorizza una serie categorica in modo ottimizzato
  - i valori possibili vanno ovviamente determinati a priori



## Creare una Serie Categorica

- Si crea una serie <sup>categorica</sup> ~~categorica~~ indicando **category** come **dtype**

```
>>> weather = pd.Series(["Sunny", "Sunny", "Cloudy",
                         "Rainy", "Rainy", "Cloudy"], dtype="category")
>>> weather
0    Sunny
[...]
5    Cloudy
dtype: category
Categories (3, object): [Cloudy, Rainy, Sunny]
```

- In alternativa si può convertire una serie esistente
  - il metodo **astype**, come in NumPy, crea un nuovo oggetto con gli stessi dati convertiti in un altro tipo

```
>>> weather = weather_str.astype("category")
```





## Dati Mancanti

- Nella pratica, un set di dati ha spesso dei **valori mancanti**
  - ad es. perché non esistono o non sono stati forniti
- Una serie può avere valori mancanti, detti **NA (Not Available)**
- Nel caso di serie numeriche, un dato mancante è rappresentato internamente dal valore **nan (Not a Number)**
- Come in altri linguaggi, il valore **nan** non risulta mai uguale, maggiore o minore di altri numeri (**incluso se stesso!**)

```
>>> np.nan == np.nan
False
```

- Qualsiasi espressione numerica con **nan** ha risultato **nan**

```
>>> 2 * np.nan - 1
nan
```



## Creare una Serie da un Dizionario (dict)

- Si può creare una **Series** copiando i dati di un **dict**
- Ad ogni coppia **chiave-valore** è fatta corrispondere una coppia **etichetta-valore**
- Specificando un **index**, si selezionano chiavi specifiche
  - se una **chiave in index** non è nel **dict**, il suo valore sarà **NA**
- Senza specificare un **index** si considera tutto il **dict** in ordine indefinito

```
>>> sdata = {
    'Ohio': 35000,
    'Texas': 71000,
    'Oregon': 16000,
    'California': 5000
}

>>> states = ['Utah', 'Ohio', 'Oregon', 'Texas']

>>> pd.Series(sdata, index=states)
Utah      NaN
Ohio    35000.0
Oregon   16000.0
Texas    71000.0
dtype: float64
```



## Accesso agli Elementi di una Serie

- Gli elementi di una serie si possono leggere e scrivere come se si trattasse di un `dict` in cui le chiavi sono le etichette
    - per leggere un valore, se ne specifica l'etichetta come indice
    - è possibile assegnare un nuovo valore ad un'etichetta esistente
    - si possono inserire nuove etichette che sono aggiunte alla fine
    - si può eliminare un valore con `del`
- ```
>>> ser["a"]      # leggi valore etichettato con a
-5
>>> ser["d"] = 6   # imposta valore con etichetta d
>>> del ser["d"]    # elimina valore con etichetta d
```
- Si può in generale usare una serie come se fosse un `dict`
    - ad es. si può verificare la presenza di un'etichetta con l'operatore `in` o si possono iterare le coppie etichetta-valore col metodo `items`



## Accesso a Elementi Multipli

- Si può estrarre parte di una serie in modo simile ai `ndarray`
  - Indicando come indice una **lista o array di etichette**, viene estratta una serie con i soli valori indicati
- ```
>>> ser[["c", "a"]]
c      3
a     -5
```
- |   |    |
|---|----|
| d | 4  |
| b | 7  |
| a | -5 |
| c | 3  |
- Indicando un **intervallo con due etichette**, si selezionano tutti gli elementi nel mezzo, **inclusi entrambi gli estremi**
- ```
>>> ser["d":"a"]
d      4
b      7
a     -5
```



## Operazioni Comuni sulle Serie

- Le serie supportano in generale molte operazioni simili a quelle dei `ndarray`
- Si può applicare un'operazione con uno scalare a tutti i valori
- Si può applicare una funzione universale NumPy
- Con un confronto si ottiene una **serie booleana**, utilizzabile come filtro

```
>>> ser * 2
d      8
b     14
[...]
>>> np.exp(ser)
d     54.598150
b    1096.633158
[...]
>>> ser > 3
d     True
b     True
a    False
c    False
>>> ser[ser > 3]
d      4
b      7
```

|   |    |
|---|----|
| d | 4  |
| b | 7  |
| a | -5 |
| c | 3  |

Seleziona valori di `ser` le cui etichette sono quelle dei valori di `ser` maggiori di 3



## Operazioni Binarie sulle Serie

- Le serie supportano anche operazioni binarie tra esse
  - con operatori `+`, `-`, `*`, ...
  - con funzioni universali
- L'operazione è applicata **per elementi con pari etichetta**
  - non viene considerato l'ordine dei valori
- Per ogni etichetta presente **solo in un operando** si avrà un valore **NA** nel risultato

```
>>> x = pd.Series({
    "b": 3,
    "a": 2,
    "c": 5})
>>> y = pd.Series({
    "a": 2,
    "b": 4,
    "d": 8})
>>> x + y
a     4.0
b     7.0
c     NaN
d     NaN
dtype: float64
```



## Metodi di Riduzione delle Serie

- Le serie offrono metodi per calcolare statistiche aggregate sui valori, simili a quelli degli `ndarray` di NumPy
  - `sum` (somma), `mean` (media), `min` (minimo), `max` (massimo), ...
- Di default, eventuali valori mancanti vengono ignorati

```
>>> pd.Series([2, np.nan, 6, 4]).mean()
4.0
```

- Specificando `skipna=False` invece i NA invalidano il calcolo

```
>>> pd.Series([2, np.nan, 6, 4]).mean(skipna=False)
nan
```

- Rispetto a NumPy sono aggiunti i metodi `idxmin` e `idxmax`, che restituiscono l'etichetta del valore minimo o massimo

```
>>> pd.Series({"a": 6, "b": 10, "c": 7}).idxmax()
'b'
```



## Valori Distinti in una Serie

- Il metodo `unique` restituisce un vettore con tutti i valori distinti in una serie
- `nunique` ne restituisce direttamente la quantità
- `value_counts` restituisce una nuova serie che associa ad ogni valore distinto (usato come etichetta) il suo numero di occorrenze, ordinandoli dal più al meno frequente (con `ascending` si cambia l'ordine)

```
>>> x = pd.Series([6, 2, 1, 2, 5, 1, 1])
>>> x.unique()
array([6, 2, 1, 5])
>>> x.nunique()
4
>>> x.value_counts()
1      3
2      2
6      1
5      1
dtype: int64
>>> x.value_counts(ascending=True)
```



## Verifica e Rimozione dei Valori Mancanti

- I metodi `isna` e `notna` verificano quali elementi (non) sono mancanti e restituiscono una serie `bool`
- `count` restituisce il numero di valori non NA nella serie
- `dropna` rimuove i valori mancanti dalla serie
  - di default, viene creata una copia della serie
  - indicando `inplace=True` viene invece modificata la serie stessa

```
>>> s = pd.Series([1,2, np.nan, 4], index=list("abcd"))
```

```
>>> s.isna()
```

```
a    False
b    False
c     True
d    False
```

|   |    |
|---|----|
| a | 1  |
| b | 2  |
| c | NA |
| d | 4  |

```
>>> s.count()
3
```

```
>>> s.dropna()
```

```
a    1.0
b    2.0
d    4.0
```

```
>>> s.dropna(inplace=True)
[...]
```



## Sostituzione dei Valori Mancanti

- Il metodo `fillna` permette di rimpiazzare i valori NA
  - anche qui viene creata una copia a meno che non si specifichi `inplace=True`
- Indicando un valore, tutti gli NA sono sostituiti con esso
  - è comune usare la media
- Usando invece il parametro `method` pari a `ffill` o `bfill` ogni NA è sostituito col valore non NA prima o dopo (se esiste!)
  - utile per serie temporali

(A) `s.fillna(s.mean())`

(B) `s.fillna(method="ffill")`

(C) `s.fillna(method="bfill")`

|   | S  | (A) | (B) | (C) |
|---|----|-----|-----|-----|
| a | 1  | 1   | 1   | 1   |
| b | NA | 2   | 1   | 2   |
| c | NA | 2   | 1   | 2   |
| d | 2  | 2   | 2   | 2   |
| e | 3  | 3   | 3   | 3   |
| f | NA | 2   | 3   | NA  |

Non essendoci alcun valore valido successivo, rimane NA



La discretizzazione serve per trasformare una variabile numerica continua in una variabile categorica, suddividendo i suoi valori in intervalli (detti bin o bucket). Questo può essere utile per:

- Raggruppare i dati in categorie (es. età -> giovani, adulti, anziani).
- Visualizzazioni (es. istogrammi).
- Algoritmi di apprendimento automatico che lavorano meglio con feature categoriche.

## Discretizzazione (*binning*)

- Una serie numerica può contenere **molte valori distinti**, ma in alcuni casi **serve avere un insieme discreto e finito di valori**
  - ad es. per raggruppare i dati o per alcuni algoritmi di apprendimento
- Per rendere discreti valori numerici è comune suddividere l'intervallo di valori possibili in parti più piccole (*bin* o *bucket*)
- pandas offre funzioni per creare una serie categorica da una numerica, che usa gli **intervalli di valori come categorie**
- La funzione **cut** suddivide i valori dati
  - in un numero dato di intervalli di larghezza fissa
  - oppure in intervalli con limiti specificati manualmente
- La funzione **qcut** (*quantile cut*) suddivide invece i dati in modo che ogni bucket abbia (circa) la stessa quantità di valori

Suddivide l'intervallo dei dati in bin con limiti fissi (uguale larghezza o definiti da te). Utile se vuoi controllare i limiti esatti degli intervalli.

# Creo 4 intervalli di larghezza uguali  
pd.cut(data, bins=4)

pd.qcut(data, q=4)

Divide i dati in quantili, quindi ogni bin contiene (quasi) lo stesso numero di elementi.  
Utile per dati non distribuiti uniformemente.

Applicazioni Data Intensive - G. Moro, R. Pasolini - DISI, Università di Bologna, Cesena

27



## Discretizzazione: Esempi

```
>>> x
0    1
1    1
2    1
3    2
4    2
5    3
6    4
7    5
dtype: int64
```

una serie  
numerica con  
valori  
sbilanciati  
(valori piccoli  
più frequenti)

```
>>> pd.cut(x, [0, 1, 3, 6])
0    (0, 1]
1    (0, 1]
2    (0, 1]
3    (1, 3]
4    (1, 3]
5    (1, 3]
6    (3, 6]
7    (3, 6]
dtype: category
```

3 intervalli di  
larghezza  
variabile con  
limiti indicati  
dall'utente

```
>>> pd.cut(x, 3)
0    (0.996, 2.333]
1    (0.996, 2.333]
2    (0.996, 2.333]
3    (0.996, 2.333]
4    (0.996, 2.333]
5    (2.333, 3.667]
6    (3.667, 5.0]
7    (3.667, 5.0]
dtype: category
```

3 intervalli di  
larghezza  
fissa: molti  
elementi  
vanno nel  
primo  
intervallo

```
>>> pd.qcut(x, 3)
0    (0.999, 1.333]
1    (0.999, 1.333]
2    (0.999, 1.333]
3    (1.333, 2.667]
4    (1.333, 2.667]
5    (2.667, 5.0]
6    (2.667, 5.0]
7    (2.667, 5.0]
dtype: category
```

3 intervalli  
con numero  
di elementi  
bilanciato: il  
primo  
intervallo è  
più stretto



## DataFrame

- Un **DataFrame** rappresenta un set di dati in forma relazionale
- Può essere visto come una sequenza di colonne rappresentate da **serie** di diverso tipo **con etichette condivise**
  - le etichette sono di solito identificatori univoci delle righe
- Ogni serie (colonna) ha un **nome**, utilizzabile per accedere ad essa

|      | nome  | cognome | età | sexo | # acquisti | cat. preferita |
|------|-------|---------|-----|------|------------|----------------|
| 1234 | Mario | Rossi   | 42  | M    | 8          | Libri          |
| 1357 | Maria | Verdi   | 35  | F    | 12         | Musica         |
| ...  | ...   | ...     | ... | ...  | ...        | ...            |



## Creare un DataFrame

- Si può creare un **DataFrame** da un **dict** con la lista o serie di valori di ciascuna colonna mappati al suo nome
  - con **columns** si indica l'**ordine delle colonne** (default: alfabetico)
  - si può specificare l'**indice delle righe** come parametro **index**

```
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,
    columns=['year', 'state', 'pop'],
    index=['one', 'two', 'three',
           'four', 'five', 'six'])
```

|       | year | state  | pop |
|-------|------|--------|-----|
| one   | 2000 | Ohio   | 1.5 |
| two   | 2001 | Ohio   | 1.7 |
| three | 2002 | Ohio   | 3.6 |
| four  | 2001 | Nevada | 2.4 |
| five  | 2002 | Nevada | 2.9 |
| six   | 2003 | Nevada | 3.2 |



## Creare un DataFrame da una Matrice

- Un **DataFrame** con colonne tutte dello stesso tipo può essere creato da un **ndarray** con 2 dimensioni
  - il parametro **columns** è usato per indicare i nomi delle colonne

```
>>> values = np.arange(9).reshape((3, 3))
```

```
>>> values
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> df = pd.DataFrame(values,
                       columns=["A", "B", "C"])
```

|   | A | B | C |
|---|---|---|---|
| 0 | 0 | 1 | 2 |
| 1 | 3 | 4 | 5 |
| 2 | 6 | 7 | 8 |

- In questo esempio, non specificando **index**, le righe sono etichettate automaticamente con gli interi da 0 a N-1



## Attributi di Base di un DataFrame

- Un **DataFrame** fornisce molti attributi simili a quelli di un **ndarray** a due dimensioni (una matrice)
- shape** fornisce una tupla col numero di righe e di colonne
- Le liste dei nomi di righe e colonne costituiscono gli *indici* del **DataFrame**, accessibili con gli attributi **index** e **columns**
  - si ottengono oggetti **Index**, utilizzabili in genere come liste o vettori

```
>>> df.shape
(6, 3)
```

```
>>> df.index
Index(['one', 'two', 'three', 'four', ...] ...)
```

- L'attributo **T** fornisce (come per gli **ndarray**) una vista *trasposta* del **DataFrame**, con righe e colonne scambiate





## Leggere Dati da CSV

- pandas fornisce varie funzioni per importare dati da sorgenti esterne di diverso tipo e caricarli in un `DataFrame`
- Tra queste `read_csv` consente di creare un `DataFrame` caricando i dati da un file CSV
  - `read_csv` carica l'intero file in memoria; per caricare e processare una riga alla volta si usa invece il modulo `CSV` di Python
- Va passato un oggetto file da cui leggere, oppure direttamente il nome o l'URL di un file da aprire
 

```
>>> data = pd.read_csv("mydata.csv")
```
- I dati letti in forma testuale sono convertiti nei tipi appropriati (numeri interi, reali, ...), dedotti automaticamente da pandas
  - si può però controllare con i parametri, vedi prossima slide



## Opzioni di Lettura dei CSV

- Il metodo `read_csv` ha diversi parametri opzionali, ad es.:
  - `sep`: separatore di colonna da usare (default ",")
  - `names`: nomi delle colonne (di default letti dalla prima riga)
  - `index_col`: numero/nome della colonna da usare come indice
    - se non specificata, viene generato un indice numerico 0...N-1
  - `dtype`: tipo di dati delle colonne o dizionario che associa il tipo corretto al nome di ciascuna colonna
  - `nrows`: massimo numero di righe da leggere
- Usando questi parametri opportunamente è possibile importare i dati in modo da ottimizzare l'uso della memoria
  - con `dtype` possiamo definire tipi di dati efficienti da usare
  - con `nrows` si possono importare poche righe per verificare preventivamente i tipi di dati da usare, per poi caricare l'intero file



## Estrarre una Colonna di un DataFrame

- Una colonna di `DataFrame` può essere estratta in forma di serie usando **il suo nome come indice**
- Se il nome è un identificatore valido non usato da pandas, si può accedere alla colonna **come se fosse un attributo**
- I nomi delle righe sono usati come etichette della serie

Le serie possono avere un nome (attributo `name`): in questo caso è usato il nome della colonna

```
>>> df
      year  state  pop
one    2000   Ohio  1.5
two    2001   Ohio  1.7
[...]
six    2003  Nevada  3.2
>>> df["year"]
one    2000
two    2001
[...]
six    2003
Name: year, dtype: int64
>>> df.year # equivalente
```



## Assegnare Valori ad una Colonna

- Assegnando una sequenza di valori ad una colonna, si modificano i valori presenti in quella colonna del `DataFrame`
  - se una colonna col nome dato non esiste, viene creata
- Il valore assegnato ad una colonna (anche quando si crea il `DataFrame`) può essere
  - una qualsiasi **sequenza** (tupla, lista, vettore NumPy, ...), i cui valori sono considerati nel loro ordine
  - una **serie** le cui etichette combacino con quelle del `DataFrame`
    - per etichette assenti si avranno valori NA nella nuova colonna
  - un **valore scalare**, che sarà replicato in tutte le righe
- Ad es. `df["foo"] = 0` imposta a 0 tutti i valori nella colonna "foo", creandola come ultima se non esiste
- Per eliminare una colonna, usare `del df[colonna]`



## Colonne con Valori Derivati

- Usando le operazioni e funzioni valide per le serie, è facile aggiungere colonne i cui valori sono derivati dalle altre
- Possibili esempi includono
  - somme, differenze, ... tra colonne diverse

```
acct["profit"] = acct["revenue"] - acct["expense"]
```

```
items["total_price"] = \
    items["quantity"] * items["unit_price"]
```

- variabili booleane da condizioni su colonne

```
people["is_adult"] = people["age"] >= 18
```

- suddivisione di valori in intervalli

```
people["age_range"] = \
    pd.cut(people["age"], [0, 18, 30, 45, 60, 100])
```



## Accedere a Valori Singoli di un DataFrame

- Per accedere a valori singoli o porzioni dei dati, i DataFrame forniscono diversi oggetti “selettori” (*accessors*)
- Con `at`, si accede ad un **valore singolo**, dando come indici le **etichette (nomi)** della riga e della colonna

|   |       | 0    | 1      | 2   |
|---|-------|------|--------|-----|
|   |       | year | state  | pop |
| 0 | one   | 2000 | Ohio   | 1.5 |
| 1 | two   | 2001 | Ohio   | 1.7 |
| 2 | three | 2002 | Ohio   | 3.6 |
| 3 | four  | 2001 | Nevada | 2.4 |
| 4 | five  | 2002 | Nevada | 2.9 |
| 5 | six   | 2003 | Nevada | 3.2 |

```
>>> df.at["one", "year"]    # riga "one", col. "year"
2000
```

```
>>> df.at["one", "year"] = 2001    # cambia il valore
```

- Con `iat` si utilizzano invece gli indici per **posizione**

```
>>> df.iat[1, 0]    # seconda riga, prima colonna
2001
```



## Estrarre Parti di DataFrame

- `at` e `iat` permettono di estrarre solo valori singoli
- Con i selettori `loc` e `iloc` è possibile selezionare per etichette o per posizioni **righe e colonne multiple**
- Si selezionano righe e colonne come negli `ndarray`
  - si può indicare **un intervallo o una lista** di etichette o posizioni (negli intervalli di etichette, entrambi gli estremi sono inclusi)
  - indicando un **array o serie di valori booleani**, si selezionano le etichette o posizioni con valori `True` (ad es. che soddisfino una condizione)
  - l'indice delle colonne si può omettere per selezionarle tutte
- Selezionando **una singola riga o colonna**, si ottiene **una serie** le cui etichette sono i nomi delle colonne o delle righe
  - proprio come in NumPy selezionando una singola riga o colonna di matrice si ottiene un vettore

Applicazioni Data Intensive - G. Moro, R. Pasolini - DISI, Università di Bologna, Cesena

39



## Estrarre Parti di DataFrame: Esempi

- Una riga data l'etichetta o la posizione

```
>>> df.loc["one"]
>>> df.iloc[0]
year      2000
state     ohio
pop        1.5
Name: one, dtype: object
```

- Alcune colonne

```
>>> df.loc[:, ["state", "pop"]]
>>> df.iloc[:, [1, 2]]
```

**SQL:**  
SELECT state, pop FROM df

- Intervallo di righe

```
>>> df.loc["one":"three"]
>>> df.iloc[:3]
```

|       | year | state | pop |
|-------|------|-------|-----|
| one   | 2000 | Ohio  | 1.5 |
| two   | 2001 | Ohio  | 1.7 |
| three | 2002 | Ohio  | 3.6 |



|       | state  | pop |
|-------|--------|-----|
| one   | Ohio   | 1.5 |
| two   | Ohio   | 1.7 |
| three | Ohio   | 3.6 |
| four  | Nevada | 2.4 |
| five  | Nevada | 2.9 |
| six   | Nevada | 3.2 |



## Selezionare Intervalli di Righe da un DataFrame

- Si può selezionare un intervallo di righe per posizione (non per etichetta) anche usandolo come indice del DataFrame
  - ovvero senza bisogno di utilizzare `iloc`
- Esistono inoltre i metodi `head` e `tail` per selezionare le prime o ultime *N* righe (default: 5)
  - sono utilizzabili anche sulle serie
  - usati spesso in modalità interattiva per visualizzare alcune righe d'esempio di un DataFrame caricato esternamente

```
>>> df[10:20]    # seleziona righe dall'11a alla 20a
```

```
>>> df.head()    # prime 5 righe
>>> df.tail(10)   # ultime 10 righe
```

SQL:  
SELECT \*  
FROM df  
LIMIT 5



## Selezionare Righe tramite Condizioni

- Possiamo selezionare le righe anche tramite array o serie booleane, che possono essere ricavate da condizioni sui dati

```
>>> df[df.year >= 2002]
```

|       | year | state  | pop |
|-------|------|--------|-----|
| three | 2002 | Ohio   | 3.6 |
| five  | 2002 | Nevada | 2.9 |
| six   | 2003 | Nevada | 3.2 |

SQL:  
SELECT \*  
FROM df  
WHERE year >= 2002

|       | year | state  | pop |
|-------|------|--------|-----|
| one   | 2000 | Ohio   | 1.5 |
| two   | 2001 | Ohio   | 1.7 |
| three | 2002 | Ohio   | 3.6 |
| four  | 2001 | Nevada | 2.4 |
| five  | 2002 | Nevada | 2.9 |
| six   | 2003 | Nevada | 3.2 |

- Su array/serie booleane è possibile usare gli operatori `&` (AND), `|` (OR) e `~` (NOT)

```
>>> df[(df.state == "ohio") & (df["pop"] < 2)]
```

|     | year | state | pop |
|-----|------|-------|-----|
| one | 2000 | Ohio  | 1.5 |
| two | 2001 | Ohio  | 1.7 |

SQL:  
SELECT \*  
FROM df  
WHERE state = 'ohio'  
AND pop < 2

Non si può usare `df.pop` in quanto `pop` è già un metodo di pandas



## Metodi di Riduzione sui DataFrame

- I DataFrame offrono gli stessi metodi delle serie per calcolare statistiche aggregate (`sum`, `mean`, `min`, `max`, ...)
- Di default restituiscono una serie col risultato dell'operazione per ogni colonna dove sia applicabile
  - si può anche qui specificare `skipna=False` per non ignorare gli NA

```
>>> df.mean()
year    2001.50
pop      2.55
```

SQL:

```
SELECT MEAN(year), MEAN(pop)
FROM df
```

- Specificando `axis=1`, l'operazione si applica per righe

```
>>> df.sum(axis=1)
one    2001.5
two    2002.7
[...]
```

In questo caso specifico l'aggregazione per righe non ha senso, ma può essere utile quando tutte le colonne hanno valori confrontabili tra loro (es. popolazione in diversi anni)



## Sommario di un DataFrame

- Col metodo `describe` si ottengono le principali statistiche aggregate delle colonne di un DataFrame (o di una serie)
  - utile in modalità interattiva per un'analisi ad alto livello delle colonne presenti in un dataset caricato

```
>>> df.describe()
              year    pop
count    6.000000  6.000000
mean     2001.50000  2.550000
std       1.048809  0.836062
min       2000.00000  1.500000
25%       2001.00000  1.875000
50%       2001.50000  2.650000
75%       2002.00000  3.125000
max       2003.00000  3.600000
```

← num. valori non NA

← media

← deviazione standard

percentili

es. il 25% dei valori di pop è inferiore a 1,875 ed il restante 75% è superiore a questa soglia



## Riordinare i Dati in un DataFrame

- Il metodo `sort_index` ordina le righe di un DataFrame in ordine crescente di etichetta
- `sort_values` ordina le righe invece secondo una o più colonne indicate (per nome o per posizione)
- Entrambi i metodi forniscono alcuni parametri:
  - di default è restituito un nuovo DataFrame, con `inplace=True` si modifica quello esistente
  - con `ascending=False` si usa l'ordine decrescente (se si ordina su  $N$  colonne, si può indicare una lista di  $N$  booleani per differenziarle)
  - con `axis=1` si ordinano le colonne invece delle righe
  - `na_position` indica se collocare eventuali valori NA per ultimi ("last", default) o per primi ("first")



## Riordinare i Dati: Esempi

**SQL:**  
 SELECT \*  
 FROM df  
 ORDER BY  
 pop DESC

DataFrame originale

df

|       | year | state  | pop |
|-------|------|--------|-----|
| one   | 2000 | Ohio   | 1.5 |
| two   | 2001 | Ohio   | 1.7 |
| three | 2002 | Ohio   | 3.6 |
| four  | 2001 | Nevada | 2.4 |
| five  | 2002 | Nevada | 2.9 |
| six   | 2003 | Nevada | 3.2 |

Ordinamento per etichette (in ordine alfabetico)

`df.sort_index()`

|       | year | state  | pop |
|-------|------|--------|-----|
| five  | 2002 | Nevada | 2.9 |
| four  | 2001 | Nevada | 2.4 |
| one   | 2000 | Ohio   | 1.5 |
| six   | 2003 | Nevada | 3.2 |
| three | 2002 | Ohio   | 3.6 |
| two   | 2001 | Ohio   | 1.7 |

Ordinamento per popolazione decrescente

`df.sort_values("pop", ascending=False)`

|       | year | state  | pop |
|-------|------|--------|-----|
| three | 2002 | Ohio   | 3.6 |
| six   | 2003 | Nevada | 3.2 |
| five  | 2002 | Nevada | 2.9 |
| four  | 2001 | Nevada | 2.4 |
| two   | 2001 | Ohio   | 1.7 |
| one   | 2000 | Ohio   | 1.5 |



## Download da Web di File Dati e Utilizzo in Pandas: Esempio

```
In [209]: import os
if not os.path.exists("covid19-regioni.json"):
    from urllib.request import urlopen
    urlretrieve("https://raw.githubusercontent.com/pcm-dpc/COVID-19/master/dati-json/dpc-covid19-ita-regioni.json",
               "covid19-regioni.json")

In [210]: covid19_regioni = pd.read_json("covid19-regioni.json")

In [212]: covid19_regioni.columns

Out[212]: Index(['data', 'stato', 'codice_regione', 'denominazione_regione', 'lat',
                'long', 'ricoverati_con_sintomi', 'terapia_intensiva',
                'totale_ospedalizzati', 'isolamento_domiciliare',
                'totale_attualmente_positivi', 'nuovi_attualmente_positivi',
                'dimessi_guariti', 'deceduti', 'totale_casi', 'tamponi'],
                dtype='object')

In [213]: covid19_regioni.iloc[:5,:]

Out[213]:
```

|   | data                | stato | codice_regione | denominazione_regione | lat       | long      | ricoverati_con_sintomi | terapia_intensiva | total |
|---|---------------------|-------|----------------|-----------------------|-----------|-----------|------------------------|-------------------|-------|
| 0 | 2020-02-24 18:00:00 | ITA   | 13             | Abruzzo               | 42.351222 | 13.398438 | 0                      | 0                 |       |
| 1 | 2020-02-24 18:00:00 | ITA   | 17             | Basilicata            | 40.639471 | 15.805148 | 0                      | 0                 |       |
| 2 | 2020-02-24 18:00:00 | ITA   | 4              | P.A. Bolzano          | 46.499335 | 11.356624 | 0                      | 0                 |       |
| 3 | 2020-02-24 18:00:00 | ITA   | 18             | Calabria              | 38.905976 | 16.594402 | 0                      | 0                 |       |
| 4 | 2020-02-24 18:00:00 | ITA   | 15             | Campania              | 40.839566 | 14.250850 | 0                      | 0                 |       |

```
In [ ]: # ordina i dati per regione e poi per data
covid19_regioni.sort_values(['codice_regione', 'data'], inplace=True)

Applicazioni Data Intensive - G. Moro, R. Pasolini - DISI, Università di Bologna, Cesena
```

47



## Esercizi Facoltativi con Pandas

- riprodurre il file jupyter come mostrato nella slide precedente
  - il file con i dati è aggiornato ogni giorno alle 18 circa a questo indirizzo <https://raw.githubusercontent.com/pcm-dpc/COVID-19/master/dati-json/dpc-covid19-ita-regioni.json>
- interpretare i dati con **describe** e selezionare
  - solo i dati di Emilia Romagna, Lombardia e Veneto
  - solo i dati gli ultimi 5 giorni
  - calcolare per ogni giorno la percentuale del valore accumulato dei deceduti sui positivi
  - confrontare media e deviazione standard dei nuovi attualmente positivi di Lombardia, Veneto ed Emilia Romagna dell'ultima settimana con i rispettivi nuovi positivi degli ultimi tre giorni
  - come sopra ma per guariti
  - il giorno con il maggior numero di nuovi positivi

