

# Preparazione dei Dati in Python con Pandas (parte 2)

## 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



## Riepilogo: Serie

- Una *serie* pandas è una sequenza di **valori** in cui ognuno ha un'**etichetta** associata, ad es.:
  1. il **numero di abitanti** (in migliaia) di ciascuna **città**
  2. il **numero di ordini** di ogni **utente** di un ecommerce
  3. il **valore di chiusura giornaliero** di un titolo azionario
- Possiamo selezionare valori dalle serie tramite
  - selezione per etichette (singola, multiple o intervallo)
  - selezione booleana con altra serie con stesse etichette
- Le serie offrono molte operazioni, ad es.
  - conteggio, rimozione, sostituzione valori mancanti
  - riduzione (somma, media, ...)
  - ordinamento (per etichette o per valori)
  - discretizzazione da valori continui ad intervalli

temperatura[pioggia]

Serie 'temperatura':  
 2025-06-01 25  
 2025-06-02 28  
 2025-06-03 22  
 2025-06-04 30  
 2025-06-05 26

Serie 'pioggia' (maschera booleana):  
 2025-06-01 False  
 2025-06-02 True  
 2025-06-03 False  
 2025-06-04 True  
 2025-06-05 False

Temperature nei giorni in cui ha piovuto:  
 2025-06-02 28  
 2025-06-04 30

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

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
...	...

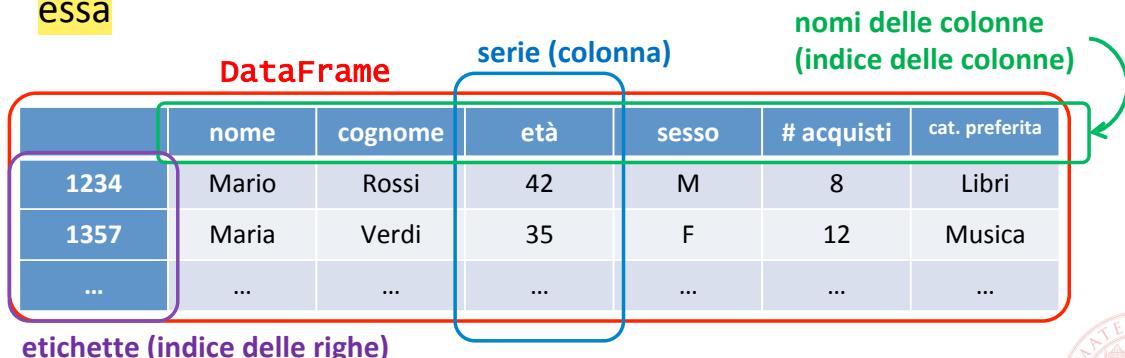
3



Preparazione dei dati con Pandas

## Riepilogo: 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



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

4



## Merge (*join*) di DataFrame

- La funzione `merge` crea un DataFrame dalla combinazione di altri due con la semantica dei *join* nei database relazionali
  - il join naturale tra due tabelle contiene le coppie di righe per le quali i valori di una coppia di colonne di join delle tabelle sono uguali
- La funzione accetta come parametri i due DataFrame (*left* e *right*), seguiti dai parametri che indicano le colonne di join
- Il join più semplice è tra una coppia di colonne
  - se le colonne hanno lo stesso nome, lo si indica col parametro `on`
  - altrimenti si usano i parametri `left_on` e `right_on`
  - di default si considerano tutte le colonne omonime tra i due frame
- Viene generato un nuovo DataFrame con valori degli indici delle righe diversi dai valori iniziali (i.e. da 0 a N-1)



## Un esempio di Join (inner)

- Siano dati i seguenti DataFrame...

	dept_id	dept_name
0	31	Sales
1	33	Engineering
2	34	Clerical
3	35	Marketing

**departments:**  
elenco reparti con dept\_ID e nome

**employees:**  
elenco impiegati con nome e ID del reparto

	emp_name	dept_id
0	Rafferty	31
1	Jones	33
2	Heisenberg	33
3	Robinson	34
4	Smith	34
5	Williams	NaN

- Per associare gli impiegati ai nomi dei rispettivi reparti, si esegue il join (inner) sulle colonne `dept_id` dei frame
- ```
>>> pd.merge(employees, departments,
            on="dept_id")
```
- l'ultima riga di `employees` manca, in quanto è assente il relativo valore di `dept_id`

**SQL:** `SELECT *`  
`FROM employees e JOIN departments d`  
`ON e.dept_id = d.dept_id`

|   | emp_name   | dept_id | dept_name   |
|---|------------|---------|-------------|
| 0 | Rafferty   | 31      | Sales       |
| 1 | Jones      | 33      | Engineering |
| 2 | Heisenberg | 33      | Engineering |
| 3 | Robinson   | 34      | Clerical    |
| 4 | Smith      | 34      | Clerical    |



## Join: Esempio con Colonne con Nome Diverso

- Siano dati i seguenti DataFrame...

departments:  
elenco reparti con  
**ID\_dept** e nome

|   | ID_dept | dept_name   |
|---|---------|-------------|
| 0 | 31      | Sales       |
| 1 | 33      | Engineering |
| 2 | 34      | Clerical    |
| 3 | 35      | Marketing   |

employees:  
elenco impiegati  
con nome e ID del  
reparto

|   | emp_name   | dept_id |
|---|------------|---------|
| 0 | Rafferty   | 31      |
| 1 | Jones      | 33      |
| 2 | Heisenberg | 33      |
| 3 | Robinson   | 34      |
| 4 | Smith      | 34      |
| 5 | Williams   | NaN     |

dato  
mancante

- Per associare gli impiegati ai nomi dei rispettivi reparti, si esegue il join sulle colonne **ID\_dept** e **dept\_id** dei frame

```
>>> pd.merge(employees, departments,
    left_on="ID_dept", right_on="dept_id")
```

- l'ultima riga di employees manca, in quanto è assente il relativo valore di dept\_id

**SQL:** SELECT \*  
FROM employees e JOIN departments d  
ON e.dept\_id = d.ID\_dept

|   | emp_name   | dept_id | dept_name   |
|---|------------|---------|-------------|
| 0 | Rafferty   | 31      | Sales       |
| 1 | Jones      | 33      | Engineering |
| 2 | Heisenberg | 33      | Engineering |
| 3 | Robinson   | 34      | Clerical    |
| 4 | Smith      | 34      | Clerical    |

7



## Altri Tipi di Join: Left, Right, Outer

- Col parametro **how** si specifica il **tipo di join** da applicare
  - "inner" (default): considera l'intersezione tra i valori di join dei 2 frame
  - "left"/"right": usa tutti i valori del primo/secondo frame, inserendo valori NA al posto di quelli assenti nel secondo/primo frame;
    - ogni riga del primo/secondo frame appare almeno una volta
  - "outer": usa tutte le chiavi di entrambi i frame
  - si tratta degli stessi tipi di join disponibili in SQL

- Ad es., per eseguire il join di prima includendo la riga con valore mancante

```
>>> pd.merge(employees, departments,
    on="dept_id", how="left")
```

**SQL:** SELECT \*  
FROM employees e LEFT JOIN departments d  
ON e.dept\_id = d.dept\_id

|   | emp_name   | dept_id | dept_name   |
|---|------------|---------|-------------|
| 0 | Rafferty   | 31      | Sales       |
| 1 | Jones      | 33      | Engineering |
| 2 | Heisenberg | 33      | Engineering |
| 3 | Robinson   | 34      | Clerical    |
| 4 | Smith      | 34      | Clerical    |
| 5 | Williams   | NaN     | NaN         |



## Join sugli Indici dei DataFrame

- Il join può essere eseguito anche tra etichette degli indici invece che tra colonne “standard” del DataFrame
- Per usare le etichette come join nel primo e/o secondo frame, impostare `left_index` e/o `right_index` a `True`
- Si ipotizzi ad esempio che nel frame `departments` sia l’indice a contenere gli ID dei reparti
- Per associare gli impiegati ai nomi dei rispettivi reparti in questo caso scriviamo:

```
>>> pd.merge(employees,
             departments,
             [left_on="dept_id",
              right_index=True])
```

|    | dept_name   |
|----|-------------|
| 31 | Sales       |
| 33 | Engineering |
| 34 | Clerical    |
| 35 | Marketing   |

|   | emp_name   | dept_id | dept_name   |
|---|------------|---------|-------------|
| 0 | Rafferty   | 31      | Sales       |
| 1 | Jones      | 33      | Engineering |
| 2 | Heisenberg | 33      | Engineering |
| 3 | Robinson   | 34      | Clerical    |
| 4 | Smith      | 34      | Clerical    |



## Indici Gerarchici

- Finora abbiamo visto indici (sia in serie che in righe e colonne di DataFrame) composti da valori semplici (es. stringhe)
- pandas supporta anche indici *gerarchici* composti da più livelli
  - ciascuno degli  $N$  livelli ha un nome (opzionale) e un tipo di valori
  - ogni etichetta dell’indice è una tupla di  $N$  valori
  - le etichette sono spesso univoche anche se i valori di ogni indice non lo sono (come in una tabella RDBMS con chiave primaria composita)
- Gli indici gerarchici sono utili per trattare dati a più dimensioni



# Indici Gerarchici: Esempio

- La seguente tabella rappresenta un sommario delle vendite annuali in diversi negozi di una catena

| anno | negozi | categoria | clienti distinti | acquisti | fatturato |
|------|--------|-----------|------------------|----------|-----------|
| 2015 | Roma   | carne     | 68               | 82       | 343 €     |
| 2015 | Roma   | pesce     | 48               | 74       | 464 €     |
| ...  | ...    | ...       | ...              | ...      | ...       |
| 2019 | Milano | bibite    | 75               | 91       | 242 €     |
| 2019 | Milano | birre     | 56               | 83       | 253 €     |

- I **livelli dell'indice** sono le diverse dimensioni su cui possiamo analizzare i dati
  - possiamo raggruppare dati per anno, negozio, categoria o combinazione di essi
- Le **colonne** contengono i dati effettivi che possono essere elaborati
  - selezione dati singolo negozio, medie per anno, grafici a torta per categorie, ...



## Impostare un Indice Gerarchico

- In ciascuna riga del DataFrame d'esempio
  - il valore "pop" costituisce **un dato** (popolazione)
  - gli altri indicano **a cosa si riferisce** (stato e anno)
- Sarebbe opportuno quindi che le colonne "year" e "state" siano usate come indici
- Usando il metodo `set_index`, si sostituisce l'indice corrente con una o più colonne
  - con un nome singolo si ha un indice semplice
  - con una lista si crea un indice gerarchico
  - come negli altri casi è creata una copia del frame, salvo specificare `inplace=True`
  - `reset_index` resetta l'indice al default
    - gli indici esistenti diventano colonne del dataframe, per evitarlo `df.reset_index(drop=True)`

|       | 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 |

set\_index

| year | state  | pop |
|------|--------|-----|
| 2000 | Ohio   | 1.5 |
| 2001 | Ohio   | 1.7 |
| 2002 | Ohio   | 3.6 |
| 2001 | Nevada | 2.4 |
| 2002 | Nevada | 2.9 |
| 2003 | Nevada | 3.2 |



## Esempio di DataFrame con Indice Gerarchico

```
>>> dfmi = df.set_index(["state", "year"])
>>> dfmi
```

nomi dei livelli dell'indice delle righe (sono copiati i nomi delle colonne)

nei livelli più a sinistra, uno stesso valore che sarebbe ripetuto su più righe è indicato solo nella prima

- Estraendo la colonna “pop” si ottiene una serie con lo stesso indice gerarchico

```
>>> dfmi["pop"]
state   year
ohio   2000    1.5
        2001    1.7 [...]
```

|        | state | year | pop |
|--------|-------|------|-----|
| Ohio   | 2000  | 1.5  |     |
|        |       | 2001 | 1.7 |
|        |       | 2002 | 3.6 |
| Nevada | 2001  | 2.4  |     |
|        |       | 2002 | 2.9 |
|        | 2003  | 3.2  |     |

nomi delle colonne (in questo caso ne è rimasta una sola)



## Accesso per Etichette a Indici Gerarchici

- Usando un indice gerarchico, ciascuna riga è identificata da una tupla piuttosto che da un valore singolo

```
>>> dfmi.loc[("ohio", 2001), "pop"] # un valore
```

1.7

```
>>> dfmi.loc[("ohio", 2001)] # intera riga
```

pop 1.7

Name: (ohio, 2001), dtype: float64

|        | state | year | pop |
|--------|-------|------|-----|
| Ohio   | 2000  | 1.5  |     |
|        |       | 2001 | 1.7 |
|        |       | 2002 | 3.6 |
| Nevada | 2001  | 2.4  |     |
|        |       | 2002 | 2.9 |
|        | 2003  | 3.2  |     |

- È anche possibile indicare un'etichetta parziale (solo i primi x valori) per selezionare un gruppo di righe

```
>>> dfmi.loc["ohio"] # tutte le righe ("ohio", ...)
```

pop

year

2000 1.5

2001 1.7

2002 3.6

il primo livello (“state”), avendone selezionato una singola etichetta, viene rimosso dal DataFrame risultante, lasciando solamente il secondo (“year”)



## Spostare Livelli tra Indici di Righe e Colonne (Pivoting)

- Avendo un DataFrame con indici di righe e/o colonne a più livelli, può essere utile “girare” un livello dall’uno all’altro
- Il metodo `stack` toglie un livello indicato dall’indice delle colonne e lo aggiunge come ultimo all’indice delle righe
  - “stack” perché sezioni di tabella da affiancate diventano *impilate*
- `unstack` compie l’operazione opposta (da righe a colonne)
  - questa operazione si può eseguire su una serie con indice multilivello per ottenere un DataFrame
- `pivot` invece toglie colonne dal frame per trasformarle in livelli degli indici di riga (`index`) o colonna (`columns`)
- I DataFrame ottenuti con questi metodi riportano valori NA in corrispondenza di righe o colonne mancanti



## Pivoting: Esempi (1) - unstack

- Dal dataframe d’esempio con indice gerarchico `dfmi`...
- ... prendo la colonna “pop” e sposto il livello “state” dalle righe alle colonne

```
>>> dfmi[“pop”].unstack(“state”)
```

| state  | year | pop |
|--------|------|-----|
| Ohio   | 2000 | 1.5 |
|        | 2001 | 1.7 |
|        | 2002 | 3.6 |
| Nevada | 2001 | 2.4 |
|        | 2002 | 2.9 |
|        | 2003 | 3.2 |

| state | Nevada | Ohio |
|-------|--------|------|
| 2000  | NaN    | 1.5  |
| 2001  | 2.4    | 1.7  |
| 2002  | 2.9    | 3.6  |
| 2003  | 3.2    | NaN  |

unstack  
i dati passano da  
essere  
“impilati” (“stack”)  
uno sopra l’altro ad  
essere affiancati

poiché le righe  
(“Nevada”,  
2000) e  
(“Ohio”, 2003)  
non esistono in  
“pop”,  
otteniamo  
valori NA



## Pivoting: Esempi (1) - stack

- Dal dataframe della slide precedente con indice gerarchico `dfmi`...

- ... (i) applichiamo questa volta stack  
`dfmi=pd.DataFrame(dfmi.stack())`

impila l'ultimo livello  
di colonne  
(state, nevada, ohio)

| year | state  | pop |
|------|--------|-----|
| 2000 | Ohio   | 1.5 |
| 2001 | Nevada | 2.4 |
|      | Ohio   | 1.7 |
| 2002 | Nevada | 2.9 |
|      | Ohio   | 3.6 |
| 2003 | Nevada | 3.2 |

(ii) riordiniamo i livelli gerarchici

`dfmi.reorder_levels(["state", "year"])`  
 (iii) riassegnamo il nome pop alla colonna e  
 riordiniamo le righe

`dfmi.rename(  
 columns={ dfmi.columns[0]: "pop"},  
 inplace = True)  
dfmi.sort_index(ascending=False)`  
 e riotteniamo il dataframe iniziale

| state | Nevada | Ohio |
|-------|--------|------|
| year  |        |      |
| 2000  | NaN    | 1.5  |
| 2001  | 2.4    | 1.7  |
| 2002  | 2.9    | 3.6  |
| 2003  | 3.2    | NaN  |

| state  | year | pop |
|--------|------|-----|
| Ohio   | 2002 | 3.6 |
|        | 2001 | 1.7 |
|        | 2000 | 1.5 |
| Nevada | 2003 | 3.2 |
|        | 2002 | 2.9 |
|        | 2001 | 2.4 |



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

17

Stack: trasforma l'ultimo livello delle colonne in un livello dell'indice di righe

## Pivoting: Esempi (2)

- Dalla tabella iniziale `df` (senza indici) uso pivot per ottenere una tabella con una riga per stato e una colonna per anno

`>>> df.pivot(index="state", columns="year")`

|       | 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 |

l'indice delle righe  
esistente viene scartato

pivot

| state  | year | pop  |      |      |      |
|--------|------|------|------|------|------|
|        |      | 2000 | 2001 | 2002 | 2003 |
| Nevada | NaN  | 2.4  | 2.9  | 3.2  |      |
| Ohio   | 1.5  | 1.7  | 3.6  | NaN  |      |

di nuovo, otteniamo valori mancanti  
per le combinazioni stato+anno  
assenti nel DataFrame originale



## Divisione di un DataFrame in Gruppi

- In molti casi è utile suddividere i dati di un DataFrame in gruppi per eseguire calcoli separati su ciascuno
  - calcolare statistiche aggregate (es. media, devstd , ..) divise per gruppi
  - trasformare dati in modo differenziato per ciascun gruppo
- Il metodo `groupby` crea una partizione in gruppi delle righe di un frame, con logica simile alla clausola GROUP BY di SQL
  - sono definite una o più chiavi, ciascuna corrispondente ad una colonna o ad un'espressione calcolata su una o più colonne
  - tutte le righe con stessi valori sulle rispettive colonne di `groupby` sono inserite in uno stesso gruppo
- L'oggetto restituito contiene i dati sui gruppi creati e fornisce diversi metodi per lavorare con essi

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

19



```
df = pd.DataFrame({
    'reparto': ['A', 'B', 'A', 'C', 'B'],
    'valore': [10, 20, 15, 5, 25],
}, index=['r0','r1','r2','r3','r4'])
```

```
grp.groups:
    Gruppo 'A': etichette = Index(['r0', 'r2'], dtype='object')
    Gruppo 'B': etichette = Index(['r1', 'r4'], dtype='object')
    Gruppo 'C': etichette = Index(['r3'], dtype='object')
```

```
grp.indices:
    Gruppo 'A': posizioni = [0, 2]
    Gruppo 'B': posizioni = [1, 4]
    Gruppo 'C': posizioni = [3]
```

## Estrarre i Gruppi da un DataFrame

- Raggruppiamo ad esempio il frame `df` in base al valore della colonna "state"
  - `bystate = df.groupby("state")`
- Otteniamo una partizione di `df` in due gruppi, chiamati "Ohio" e "Nevada"
- L'attributo `ngroups` indica il numero di gruppi ottenuti
 

```
>>> bystate.ngroups
2
```
- Gli attributi `groups` e `indices` sono dizionari che associano ai nomi dei gruppi le etichette e le posizioni delle righe relative
 

```
>>> list(bystate.groups)
['Nevada', 'Ohio']
```

|       | 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 |



## Accedere ai Dati dei Gruppi

- Iterando (es. con `for`) sull'oggetto dato da `groupby` si ottiene per ogni gruppo creato una tupla con
  - il nome del gruppo (i.e. medesimo valore di raggruppamento)
  - DataFrame con le righe contenute nel gruppo

```
for group, data in bystate:  
    [...]
```

- Col metodo `get_group` si può ottenere un DataFrame con le righe di un gruppo dato il suo nome
  - in questo caso il frame non riporta le colonne di raggruppamento

```
>>> bystate.get_group("Ohio")  
      year  pop  
one   2000  1.5  
two   2001  1.7  
three 2002  3.6
```

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

21



## Funzioni di Aggregazione per Gruppi

- Gli oggetti “groupby” hanno analoghi metodi di riduzione (i.e. aggregazione) di serie e DataFrame (`min`, `max`, `sum`, `mean`)
- Utilizzandoli, si ottiene un DataFrame con l'aggregazione richiesta calcolata separatamente su ciascun gruppo
  - ad ogni riga corrisponde un gruppo, ad ogni colonna corrisponde una colonna del frame su cui è calcolata la funzione di aggregazione

la colonna di raggruppamento è usata come nome del livello (unico in questo caso) dell'indice

```
>>> df.mean()          # medie globali  
year    2001.50  
pop     2.55  
>>> bystate.mean()   # medie per gruppo  
                year  pop  
state  
Nevada  2002  2.833333  
Ohio    2001  2.266667
```

SQL:  
SELECT MEAN(year), MEAN(pop)  
FROM df

SQL:  
SELECT state, MEAN(year),  
 MEAN(pop)  
FROM df  
GROUP BY state

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

22



## Raggruppamento di Risultati Derivati

Allineamento per indice: se passi una Series, Pandas allinea automaticamente i suoi valori a quelli del DataFrame in base all'indice  
Array-like senza indice: se fornisci una lista o un array, l'elemento in posizione i viene abbinato alla riga in posizione i del DataFrame

- Oltre a nomi di colonne, `groupby` accetta array o serie con indici che combaciano: i gruppi si basano sui valori in esso

```
>>> df.groupby(df.year >= 2002)
```

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

- Si può dare un nome alla serie con `rename`

– utile per distinguere le colonne nel risultato, vedi dopo

```
>>> group_key = (df.year >= 2002).rename("since_02")
>>> df.groupby(group_key).mean()
```

restituisce per ogni riga  
di df True se year >= 2002, False altrimenti

Serie Booleana

| since_02 | year        | pop      |
|----------|-------------|----------|
| False    | 2000.666667 | 1.866667 |
| True     | 2002.333333 | 3.233333 |

serve a dare un nome  
alla Series. Questo  
nome comparirà come  
etichetta dell'indice  
risultante dopo il  
groupby (anziché  
comparire come "None").



## Raggruppamento con Colonne Multiple

- Usando molteplici colonne di raggruppamento, si generano tanti gruppi quante le combinazioni distinte dei loro valori
- Ogni gruppo è identificato da una tupla con i medesimi valori

```
>>> since_02 = (df.year >= 2002).rename("since_02")
```

Pandas considera tutti i "grouper" contemporaneamente e costruisce i gruppi in base alla tupla di valori per ciascuna riga.

| ("Nevada", False) |       |        | ("Nevada", True) |       |        | ("Ohio", False) |       |      | ("Ohio", True) |       |     |       |      |      |     |
|-------------------|-------|--------|------------------|-------|--------|-----------------|-------|------|----------------|-------|-----|-------|------|------|-----|
| year              | state | pop    | year             | state | pop    | year            | state | pop  | year           | state | pop |       |      |      |     |
| four              | 2001  | Nevada | 2.4              | five  | 2002   | Nevada          | 2.9   | one  | 2000           | Ohio  | 1.5 | three | 2002 | Ohio | 3.6 |
|                   |       |        | six              | 2003  | Nevada | 3.2             | two   | 2001 | Ohio           | 1.7   |     |       |      |      |     |

- Applicando funzioni di aggregazione, si ottiene un frame il cui indice delle righe ha un livello per ogni valore

```
>>> mg.mean()
```

| state  | since_02 | year   | pop  |
|--------|----------|--------|------|
|        |          | False  | True |
| Nevada | False    | 2001.0 | 2.40 |
|        | True     | 2002.5 | 3.05 |
| Ohio   | False    | 2000.5 | 1.60 |
|        | True     | 2002.0 | 3.60 |



## Esecuzione di Statement SQL su DataFrame

- Con `Pandasql` si eseguono statement SQL su DataFrame

- e.g.

```
>>> import pandasql as ps
>>> query = \
    """SELECT state, avg(pop), min(pop), max(pop)
       FROM df
      WHERE year >= 2002
      GROUP BY state"""
>>> print(ps.sqldf(query))
state  avg(pop)  min(pop)  max(pop)
0  Nevada    3.05     2.9     3.2
1   Ohio     3.60     3.6     3.6
```

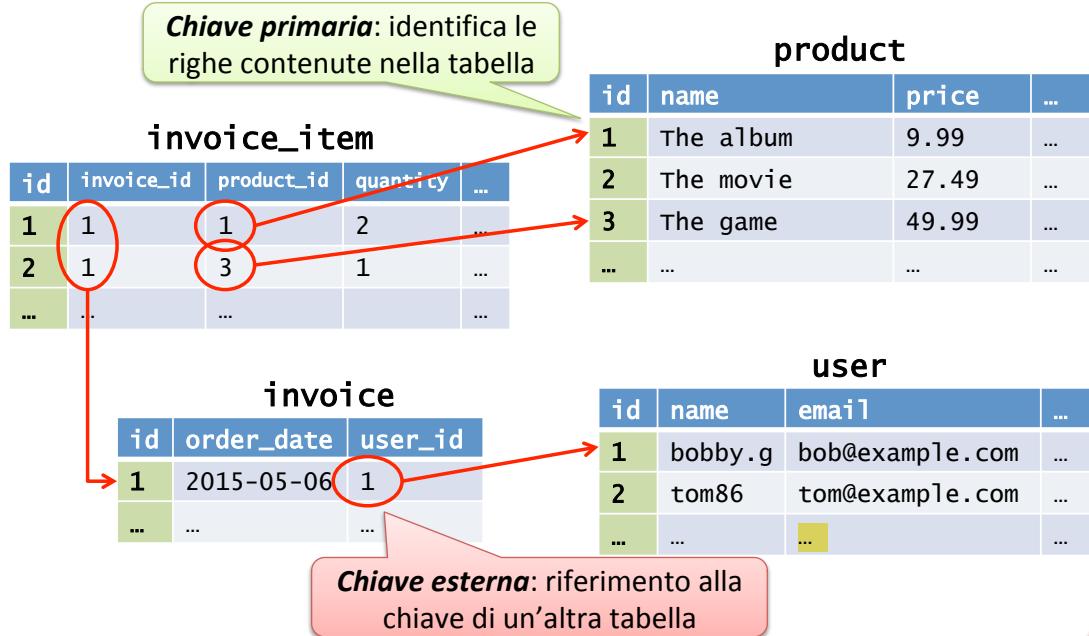


## Database Relazionali

- I **database relazionali** sono tutt'ora la soluzione più usata per la gestione di grandi volumi di dati
  - interrogazione, manipolazione dei dati e definizione della loro struttura tramite linguaggio standard SQL (*Structured Query Language*)
  - supporto di transazioni atomiche concorrenti (*ACID*), importante in applicazioni multi-utente, incluse le applicazioni Web
- Ogni database è costituito da una collezione di tabelle
  - ogni colonna è di un tipo specifico, non sono accettati dati di altri tipi
- In ogni tabella una o più colonne costituiscono la **chiave primaria**, il cui valore identifica univocamente ciascuna riga
- Tabelle diverse sono messe in relazioni tramite **chiavi esterne**, ovvero riferimenti a chiavi primarie di altre tabelle



# Esempio di Database Relazionale



## RDBMS (Relational DataBase Management System)

- Un **RDBMS** è un software che gestisce uno o più database relazionali, risolvendo diversi problemi complessi
  - ottimizzazione dei tempi di risposta delle operazioni, in particolare tramite l'uso di **indici** per individuare rapidamente specifiche righe
  - gestione di operazioni **concorrenti** sui dati con transazioni **ACID** (atomiche, consistenti, isolate, persistenti)
  - controllo degli accessi (gestione utenti con diversi diritti d'accesso)
- Un RDBMS è costituito in genere da **processi dedicati (server)**, a cui altri processi (*client*) si interfacciano con driver appositi
- Come RDBMS di riferimento consideriamo **PostgreSQL**
  - software open source affidabile con 20+ anni di sviluppo
  - alta compatibilità con lo standard SQL
  - altamente estendibile con nuovi tipi di dati, indici, funzioni, ...



## SQLite

- Al contrario di PostgreSQL e altri RDBMS, **SQLite** è un RDBMS **incorporato**: **non** usa un'architettura client-server
- Per usare SQLite, un'applicazione ne deve includere la libreria ed utilizzare la relativa API direttamente nel codice
- Le operazioni sul database sono eseguite nello **stesso processo dell'applicazione**
- Il database risiede in un **singolo file** (eventualmente in RAM)
  - per fare una copia/backup del database è sufficiente copiare il file
- SQLite è usato per dati in quantità non elevate e gestiti da **una singola applicazione mono-utente**, ad es.:
  - dati delle app Android (ad es. tutti i messaggi di testo di WhatsApp!)
  - configurazione di Firefox e Thunderbird



## Python DB API e Modulo `sqlite3`

- Pur con implementazioni diverse, i vari RDBMS si basano su concetti comuni ed usano lo stesso linguaggio (SQL)
- Il documento **PEP 249** propone un'**interfaccia standard** seguita da molte librerie Python che si interfacciano a diversi RDBMS
  - <https://www.python.org/dev/peps/pep-0249/>  
*(Python DB API Specification 2.0, da qui in breve **DB API**)*
  - simile all'API JDBC (*Java DataBase Connectivity*) in Java
  - Analogamente La stringa di connessione al db, dipende dal RDBMS prescelto
- Sull'standard si basano anche vari framework che lavorano con RDBMS, ad esempio per l'*Object Relational Mapping*
- `sqlite3` è un modulo della libreria standard Python per gestire database SQLite conforme alla DB API



## DB API / sqlite3: Connessione al Database

- La DB API prevede che il modulo che la implementa dichiari una funzione `connect` per connettersi al RDBMS supportato
- A `connect` vanno passati i parametri della connessione
  - in generale: indirizzo del database, nome utente, password, ...
- Nel caso di `sqlite3`, alla funzione `connect` passiamo il nome di un file SQLite da aprire, che viene creato se non esiste
  - E.g. `connect(dsn='myhost:MYDB', user='myuser', password='234$')`
  - per un db temporaneo in RAM si passa `dsn=':memory:'`
- Viene restituito un oggetto che rappresenta la connessione, che va chiuso col metodo `close` al termine dell'utilizzo
  - si può utilizzare il costrutto `with` per chiuderla automaticamente

```
>>> with sqlite.connect("stuff.db") as conn:  
...     ... # do something with conn
```

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

31



## DB API / sqlite3: Operazioni sul Database

- Un *cursor* permette di eseguire operazioni sul database
  - Per creare un cursore, chiamare `cursor()` sulla connessione
- ```
>>> cur = conn.cursor()
```
- Per eseguire un comando SQL, si invoca il metodo `execute`
- ```
>>> cur.execute("DELETE FROM product WHERE id=42")
```
- Un comando può contenere dei *parametri* passati da codice
    - usando dei segnaposto “`?`” e passando i parametri in una tupla
    - cur.execute("UPDATE product SET price=? WHERE id=?", (new\_price, id))
    - usando segnaposto “`:nome`” e passando i parametri in un dizionario
    - cur.execute("UPDATE product SET name=:n WHERE id=:id", {"id": id, "n": new\_id})
    - librerie diverse da sqlite3 possono usare formati di parametri diversi

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

32



## DB API / sqlite3: Interrogazioni

- Se si usa il cursore per eseguire un'interrogazione...
 

```
>>> cur.execute("SELECT name, price FROM product")
```
- ... si può iterare su di esso (es. con for) per reperire le righe una alla volta
  - ogni riga è restituita di default in forma di una tupla: con l'*unpacking* è possibile assegnare i valori delle singole colonne a diverse variabili

```
>>> for name, price in cur:
...     print("- {}: $ {}".format(name, price))
```
- In alternativa ad iterare col cursore, si può
  - ottenere una singola riga col metodo `fetchone` (può essere chiamato più volte per reperire righe successive) `cur.fetchone()`
  - ottenere tutte le righe in blocco in una lista col metodo `cur.fetchall()`
  - ottenere blocchi di k righe (k intero > 0) con `cur.fetchmany(k)`

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

33



## DB API / sqlite3: Transazioni

- Di default i comandi di scrittura dei dati non hanno effetto immediato sul database
  - al contrario ad es. di JDBC, dove di default è attivo l'*auto-commit*
- Per le transazioni, l'oggetto connessione fornisce i metodi
  - `commit` per confermare e chiudere la transazione corrente
  - `rollback` per annullare le modifiche
  - NB: la chiusura della connessione al database non comporta un commit automatico, va effettuato deliberatamente!
- Nel caso di sqlite3, le transazioni sono implementate ponendo un lock sull'intero file del database
  - questo è accettabile se il DB è utilizzato da un singolo processo
  - in altri casi (es. PostgreSQL) la gestione è a carico del RDBMS sottostante, che supporta transazioni concorrenti

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

34



## Importare un DataFrame da un Database Relazionale

- La funzione `read_sql` di pandas crea un DataFrame dal risultato di una **query SQL** su un database relazionale
  - i parametri necessari sono la `query` e la connessione al database

```
>>> with sqlite3.connect("stuff.db") as conn:  
...     data = pd.read_sql("SELECT * FROM items", conn)
```

- Come in `read_csv`, si può specificare con `index_col` una colonna da usare come indice, di solito la chiave primaria
- La query può contenere **parametri** come in `execute` nella DB API, in tal caso i loro valori vanno passati con l'opzione `params`

```
pd.read_sql("SELECT * FROM items WHERE price < ?",
            params = (10, ))
```

la virgola è necessaria per indicare che si tratta di una tupla di un elemento



## SQL e pandas



- Con `read_sql` si può eseguire facilmente **una parte di elaborazione dei dati in SQL** prima di passarli a pandas
- Quali operazioni conviene eseguire in SQL e quali in pandas ?**
- pandas lavora in RAM, per cui conviene effettuare in SQL **operazioni che riducano a monte il volume di dati importati**
  - effettuare join tra tabelle, selezionare colonne specifiche, creare nuove feature per fare previsioni, classificazioni con attributi calcolati
  - un RDBMS esegue queste operazioni con grande efficienza
- Una volta importati i dati d'interesse, si usano pandas e altre librerie Python per eseguire **elaborazioni più complesse**
  - applicare trasformazioni arbitrarie ai valori, estrarre parole dai testi, ...
  - alcune operazioni sono possibili anche in SQL, tramite estensioni specifiche dei vari RDBMS



# matplotlib

- **matplotlib** è una libreria Python di uso comune per la creazione di grafici dai dati
  - istogrammi, andamento di una serie, grafici a dispersione, grafici a torta, box plot, ...
- Legge dati da vettori in forma di oggetti “array-like”, quali liste Python, array NumPy e serie pandas
- Permette di personalizzare molti aspetti dei grafici
  - stile di linee e punti, titoli, valori sugli assi, linee guida, legenda, ...
- Serie e DataFrame pandas forniscono metodi per generare rapidamente grafici matplotlib dai dati contenuti



## Uso di matplotlib

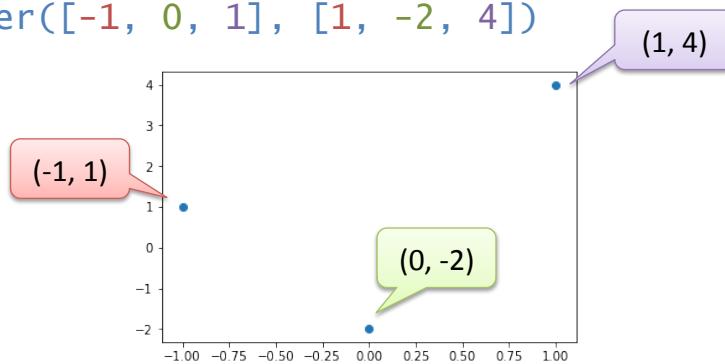
- matplotlib è spesso usato tramite la API `pyplot`, con cui si creano grafici aggiungendo progressivamente elementi
  - imita le istruzioni per creare grafici usate in Matlab
  - esiste anche un’API object-oriented, consigliata per grafici complessi
- La API `pyplot` è importata convenzionalmente come “`plt`”  
`import matplotlib.pyplot as plt`
- I grafici si possono mostrare in una finestra o esportare su file
- In Jupyter è inoltre possibile **integrare grafici nel notebook**, inserendo in una cella tutti i comandi per costruire una figura
  - per abilitare l’integrazione di matplotlib in Jupyter eseguiamo una cella a inizio documento col comando `%matplotlib inline`
  - Altri comandi “magici”: `%who` elenca variabili/funzioni, `%system comando` (e.g. `pwd`) esegue comandi shell, `%load_ext autoreload %autoreload 2` ricarica in automatico file di codice modificati referenziati, `%time` tempo di run di una cella Jupyter, `%run myfile.py` esegue file python e carica le relative funzioni/metodi nel file jupyter



# Grafico a Dispersione

- Un grafico a dispersione (*scatter plot*) mostra un insieme di dati su un piano 2D in base ai valori di due attributi X e Y
  - si può usare per valutare visivamente la correlazione tra due variabili
- Si genera con la funzione `scatter`, passando due vettori di pari lunghezza con i valori X e Y

```
plt.scatter([-1, 0, 1], [1, -2, 4])
```

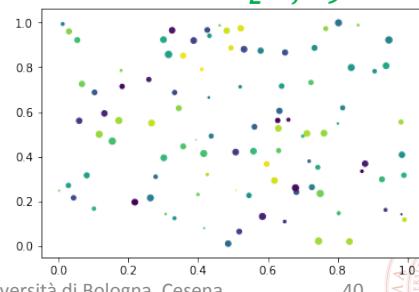


## Grafico a Dispersione: Dimensione e Colore dei Punti

- Con gli attributi `s` e `c` di `scatter` è possibile determinare dimensione (size) e colore dei punti
  - il colore è dato per nome (es. "red") o in RGB (es. "#FF0000")
- Sia dimensione che colore possono essere differenziate per i vari punti passando vettori di valori
  - per il colore si può passare un vettore di numeri, che sono convertiti in colori in base ad una colormap specificabile

```
# genera 4 sequenze di 100 float casuali in [0,1)
```

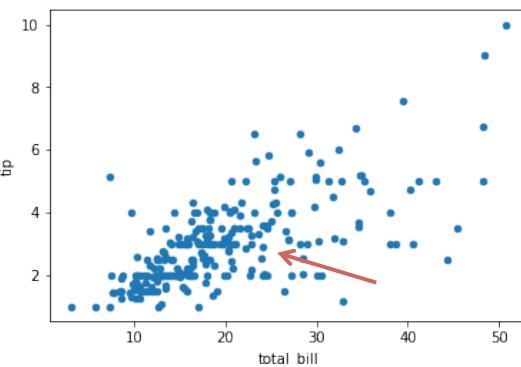
```
x, y, size, col = \
    np.random.random((4, 100))
# scala dimensioni da 0-1 a 0-30
size *= 30
plt.scatter(x, y, s=size, c=col)
```



# Grafico a Dispersione da un DataFrame

- Il metodo `plot.scatter` dei DataFrame crea un grafico con un punto per riga con valori X e Y da due colonne indicate  
`tips.head()` →  
`tips.plot.scatter("total_bill", "tip")`

i nomi delle colonne sono automaticamente indicati lungo gli assi



|   | total_bill | tip  | sex    | smoker | day | time   | size |
|---|------------|------|--------|--------|-----|--------|------|
| 0 | 16.99      | 1.01 | Female | No     | Sun | Dinner | 2    |
| 1 | 10.34      | 1.66 | Male   | No     | Sun | Dinner | 3    |
| 2 | 21.01      | 3.50 | Male   | No     | Sun | Dinner | 3    |
| 3 | 23.68      | 3.31 | Male   | No     | Sun | Dinner | 2    |
| 4 | 24.59      | 3.61 | Female | No     | Sun | Dinner | 4    |

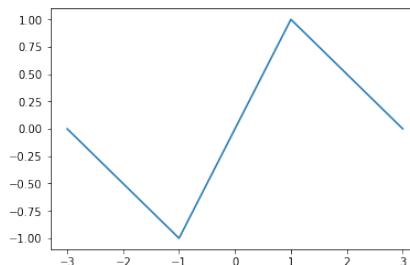
questo dataset d'esempio riporta gli importi totali ("total\_bill") di alcuni conti di un ristorante e la mancia lasciata ("tip")



# Grafico a Linea

- Un grafico a linea mostra l'andamento di una variabile Y rispetto ad una X in modo simile al grafico di una funzione
  - molto usato per serie temporali (es. l'andamento di un titolo di Borsa)
- Si genera con la funzione `plot`, sempre passando vettori di pari lunghezza delle coordinate X e Y
  - passando solo un vettore con le Y, le X default sono gli interi da 0 a N-1

`plt.plot([-3, -1, 1, 3], [0, -1, 1, 0])`

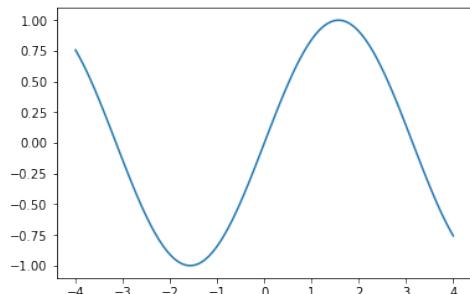


# Grafico di una Funzione

- Data una funzione ed un vettore di  $N$  valori di input, si ha il vettore degli  $N$  output corrispondenti
  - esempio: le funzioni universali di NumPy
- Si può creare un grafico della funzione definendo un vettore di X e applicando ad esso la funzione per ottenere le Y
  - è comune usare `linspace` per generare il vettore di X

```
x = np.linspace(-4, 4, 100)
y = np.sin(x)
plt.plot(x, y)
```

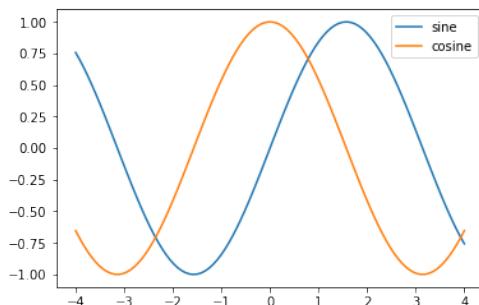
maggiore è il numero di valori usati come campioni, maggiore è la qualità del grafico



# Grafico con Linee Multiple e Legenda

- Chiamando più volte `plot` è possibile tracciare diverse serie di valori nello stesso grafico
  - sono automaticamente scelti colori differenti
- Con `Legend` è possibile aggiungere una legenda, indicando le etichette nell'ordine in cui sono aggiunte le serie
  - la posizione della legenda è scelta in automatico, ma si può impostare con un parametro `loc`

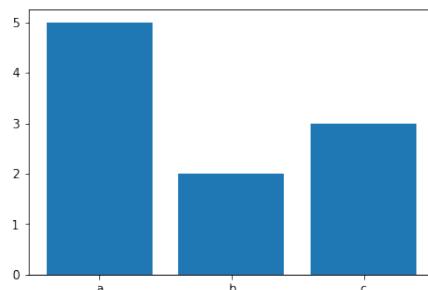
```
x = np.linspace(-4, 4, 100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
posizione = "upper left"
plt.legend(loc = posizione, ["sine", "cosine"])
```



# Grafico a Barre

- Un grafico a barre mostra una sequenza di valori tramite barre di altezza (o larghezza) proporzionale
  - usato per mostrare distribuzioni di valori, si veda slide dopo
- Si può creare con `bar`, passando una sequenza di etichette ed una di valori corrispondenti

```
plt.bar(["a", "b", "c"], [5, 2, 3])
```



# Iistogramma di una Serie di Valori

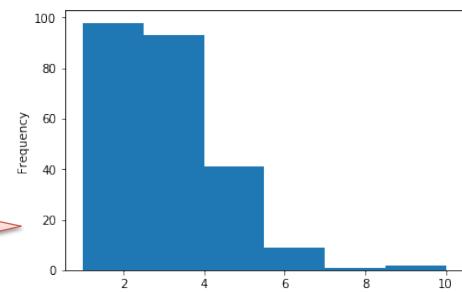
- Un istogramma visualizza tramite un grafico a barre la distribuzione di un insieme di valori in una scala continua
- Il metodo `plot.hist` genera un istogramma dai valori di una serie
  - i valori sono suddivisi in un numero specificato (default: 10) di intervalli di pari ampiezza, `hist(6)` -> 6 istogrammi

nella colonna “tip” ad es. il valore massimo è 10, ma valori superiori a 4 sono poco comuni

`tips.head()`

|   | total_bill | tip  | sex    | smoker | day | time   | size |
|---|------------|------|--------|--------|-----|--------|------|
| 0 | 16.99      | 1.01 | Female | No     | Sun | Dinner | 2    |
| 1 | 10.34      | 1.66 | Male   | No     | Sun | Dinner | 3    |
| 2 | 21.01      | 3.50 | Male   | No     | Sun | Dinner | 3    |
| 3 | 23.68      | 3.31 | Male   | No     | Sun | Dinner | 2    |
| 4 | 24.59      | 3.61 | Female | No     | Sun | Dinner | 4    |

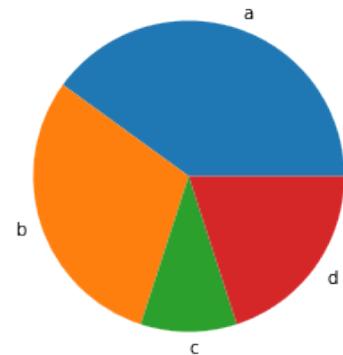
`tips["tip"].plot.hist(6)`



# Grafico a Torta

- Un grafico a torta è impiegato per mostrare la distribuzione di valori di variabili categoriche (nominali oppure ordinali)
- Si può ottenere con la funzione `pie`, passando una sequenza di  $N$  valori per generare  $N$  spicchi di dimensioni proporzionate
  - con `labels` si indica un vettore di  $N$  etichette associate agli spicchi

```
plt.pie([40, 30, 10, 20], labels=["a", "b", "c", "d"])
```



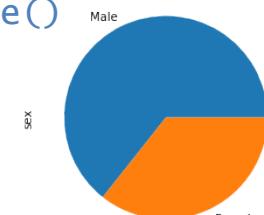
# Grafico a Torta di Frequenze di Valori

- Il metodo `plot.pie` crea un grafico a torta con dati e etichette di una serie
- Si può applicare ad una serie ottenuta da `value_counts` per visualizzare la distribuzione di valori in una serie o colonna
  - se la serie è di valori continui, la si può discretizzare con `(q)cut`
  - `pd.qcut` crea un num. prefissato di intervalli equamente frequenti
  - `pd.cut` crea un numero prefissato di intervalli equamente spaziati

`tips.head()`

|   | total_bill | tip  | sex    | smoker | day | time | size   |   |
|---|------------|------|--------|--------|-----|------|--------|---|
| 0 | 16.99      | 1.01 | Female |        | No  | Sun  | Dinner | 2 |
| 1 | 10.34      | 1.66 | Male   |        | No  | Sun  | Dinner | 3 |
| 2 | 21.01      | 3.50 | Male   |        | No  | Sun  | Dinner | 3 |
| 3 | 23.68      | 3.31 | Male   |        | No  | Sun  | Dinner | 2 |
| 4 | 24.59      | 3.61 | Female |        | No  | Sun  | Dinner | 4 |

`tips["sex"] \ .value_counts() \ .plot.pie()`



## Box Plot

- E.g.: i punti blu sono una serie di pagamenti in un ristorante che dividiamo in 4 fasce di pari frequenza
- Qual è il valore delle 4 Fasce ? Ed il pagamento min e max ? Outliers ?
- Box plot mostra tutto ciò
  - mediana, e.g. 17.795 è il pagamento che cade a metà tra tutti i pagamenti
  - primo e terzo quartile: q1 e q3, primo e ultimo 25% dei pagamenti
    - e.g.  $\frac{1}{4}$  dei pagamenti è inferiore a q1: 13.325,  $\frac{3}{4}$  è superiore a q3: 24.175
    - il 50% è compreso tra q1 e q2, InterQuantile Range (IQR) = q3 - q1 = 10.85,
    - i **baffi** sono il min e max nell'intervallo [q1 - 1.5 x IQR, q3 + 1.5 x IQR] e.g. 3.07 e 40.17, gli **outliers o valori estremi** sono i punti sopra e sotto i baffi



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

49

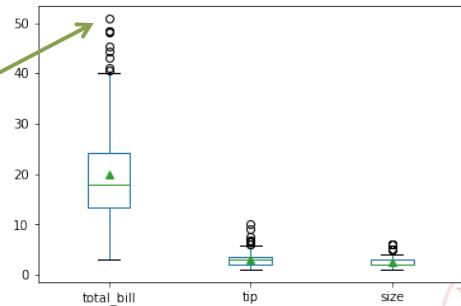


## Box Plot di Serie e DataFrame

- Il metodo `plot.box` di una serie ne visualizza un box plot
- Chiamato sui DataFrame, visualizza un grafico unico con i box plot di **tutte le colonne numeriche**
- Di default in tali box plot valori molto piccoli o grandi rispetto agli altri (*outlier*) sono visualizzati separatamente come cerchi

`tips.head()`

|   | total_bill | tip  | sex    | smoker | day | time | size   |   |
|---|------------|------|--------|--------|-----|------|--------|---|
| 0 | 16.99      | 1.01 | Female |        | No  | Sun  | Dinner | 2 |
| 1 | 10.34      | 1.66 | Male   |        | No  | Sun  | Dinner | 3 |
| 2 | 21.01      | 3.50 | Male   |        | No  | Sun  | Dinner | 3 |
| 3 | 23.68      | 3.31 | Male   |        | No  | Sun  | Dinner | 2 |
| 4 | 24.59      | 3.61 | Female |        | No  | Sun  | Dinner | 4 |

`tips.plot.box(showmeans=True)`

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

51



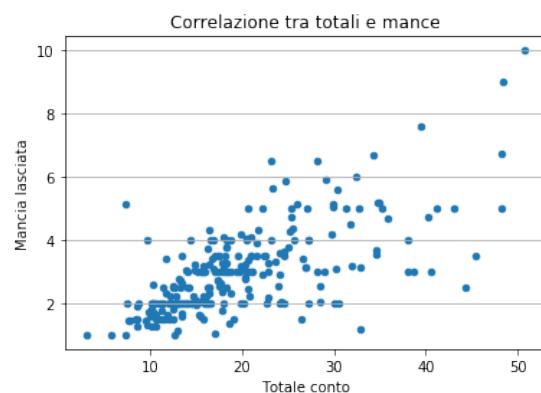
# Personalizzazione dei Grafici

- pyplot fornisce funzioni per personalizzare un grafico dopo averlo creato, aggiungendo o modificando dettagli
  - lavorando in Jupyter, tutte le funzioni relative allo stesso grafico si invocano in un'unica cella
- `title` imposta un titolo generale, mostrato in alto
- `xlabel` e `ylabel` impostano le etichette degli assi X e Y
- `grid` aggiunge linee guida all'interno del grafico per agevolarne la lettura
  - si possono aggiungere per entrambi gli assi (default) o ad uno solo (specificando `axis="x"` o `axis="y"`)



## Esempio di Grafico Personalizzato

```
tips.plot.scatter("total_bill", "tip")
plt.title("Correlazione tra totali e mance")
plt.xlabel("Totale conto")
plt.ylabel("Mancia lasciata")
plt.grid(axis="y")
```



## Easter Egg: Grafici in stile XKCD

```
with plt.xkcd():
    tips.plot.scatter("total_bill", "tip")
    plt.title("Correlazione tra totali e mance")
    plt.xlabel("Totale conto")
    plt.ylabel("Mancia lasciata")
```

