Politecnico di Torino

1859

Data Science and Machine Learning for Engineering Applications

Pandas

DataBase and Data Mining Group

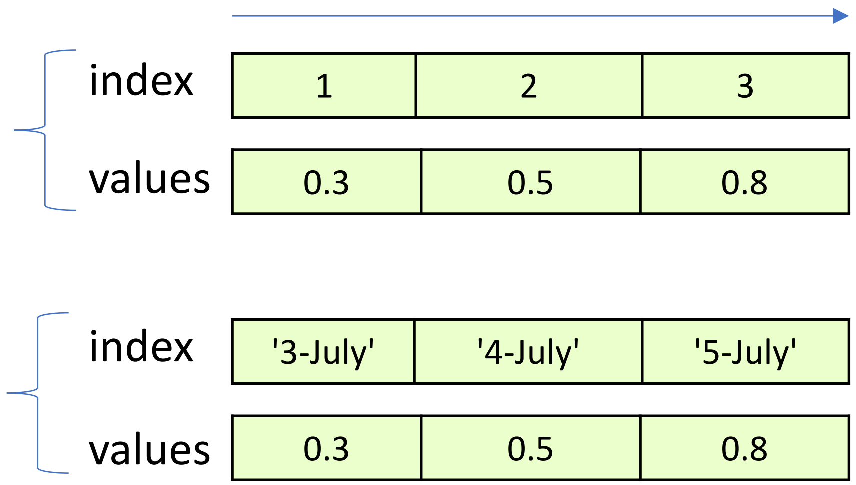
Salvatore Greco
Andrea Pasini
Flavio Giobergia
Elena Baralis
Tania Cerquitelli

Introduction to Pandas

- Pandas
  - Provides useful data structures (Series and DataFrames) and data analysis tools
  - Based on **Numpy** arrays
  - Tools:
    - Managing **tables** and **series**
      - data selection
      - grouping, pivoting
    - Managing **missing data**
    - **Statistics** on data

Pandas Series

- **Series**: 1-Dimensional sequence of homogeneous elements
- Elements are associated to an explicit **index**
  - index elements can be either strings or integers
- Examples:



Pandas Series

- **Creation** from list
  - When not specified, index is set automatically with a progressive number

In [1]:
```
import pandas as pd
s1 = pd.Series([2.0, 3.1, 4.5])
print(s1)
```

Out[1]:
```
0    2.0
1    3.1
2    4.5
```

Pandas Series

- **Creation** from list, specifying index

```
In [1]:  pd.Series([2.0, 3.1, 4.5], index=['mon', 'tue', 'wed'])
```

```
Out[1]:  'mon'    2.0
         'tue'    3.1
         'wed'    4.5
```

Pandas Series

- **Creation** from dictionary
  - keys define the index

In [1]:
```
pd.Series({'c':2.0, 'b':3.1, 'a':4.5})
```

Out[1]:
```
'c'    2.0
'b'    3.1
'a'    4.5
```

Pandas Series

- Obtaining **values** and **index** from a Series

In [1]:
```
s1 = pd.Series([2.0, 3.1, 4.5], index=['mon', 'tue', 'wed'])
print(s1.values)   # Numpy array
print(s1.index)
```

Out[1]:
```
[2.0, 3.1, 4.5]
Index(['mon', 'tue', 'wed'], dtype='object')
```

- **Index** is a custom Python object defined in Pandas

Pandas Series

- Accessing Series elements
- **Access by Index**
  - **Explicit:** the one specified while creating a Series
    - Use the Series.**loc** attribute
  - **Implicit:** number associated to the element order (similarly to Numpy arrays)
    - Use the Series.**iloc** attribute

Pandas Series

- Accessing Series elements

In [1]:
```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc['a'])          # With explicit index
print(s1.iloc[0])           # With implicit index
s1.loc['b'] = 10            # Allows editing values
print(f"Series:\n{s1}")
```

Out[1]:
```
2.0
2.0
Series:
'a'    2.0
'b'    10
'c'    4.5
```

Pandas Series

- Accessing Series elements: **slicing**

In [1]:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc['b':'c']) # explicit index (stop element included)
print(s1.iloc[1:3])    # implicit index (stop element excluded)
```

Out[1]:

```
b 3.1
c 4.5


b 3.1
c 4.5
```

Pandas Series

- Accessing Series elements: **masking**

In [1]:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1[(s1>2) & (s1<10)])
```

Out[1]:

```
b 3.1
c 4.5
```

Pandas Series

- Accessing Series elements: **fancy indexing**

In [1]:

```
s1 = pd.Series([2.0, 3.1, 4.5], index=['a', 'b', 'c'])
print(s1.loc[['a', 'c']])
print(s1.iloc[[0, 2]])
```

Out[1]:

```
a 2.0
c 4.5


a 2.0
c 4.5
```

Pandas DataFrame

- **DataFrame**: 2-Dimensional array
  - Can be thought as a table where **columns are Series** objects that share the **same index**
    - Each column has a **name**

- Example:

Index	'Price'	'Quantity'	'Liters'
'Water'	1.0	5	1.5
'Beer'	1.4	10	0.3
'Wine'	5.0	8	1

Pandas DataFrame

- **Creation** from Series
  - Use a **dictionary** to set column names

In [1]:
```
price = pd.Series([1.0, 1.4, 5], index=['a', 'b', 'c'])
quantity = pd.Series([5, 10, 8], index=['a', 'b', 'c'])
liters = pd.Series([1.5, 0.3, 1], index=['a', 'b', 'c'])
df = pd.DataFrame({'Price':price, 'Quantity':quantity,
                   'Liters':liters})
print(df)
```

Out[1]:
```
   Price   Quantity   Liters
a    1.0          5      1.5
b    1.4         10      0.3
c    5.0          8      1.0
```

Pandas DataFrame

- **Creation** from dictionary of key-list pairs
  - **Each value (list)** is associated to a **column**
    - Column name given by the key
  - **Index** is automatically set to a progressive number
    - Unless explicitly passed as parameter (index=…)
- Example:

In [1]:

```
dct = { "c1": [0, 1, 2], "c2": [0, 2, 4] }
df = pd.DataFrame(dct)
print(df)
```

Out[1]:

```
    c1     c2
0    0      0
1    1      2
2    2      4
```

Pandas DataFrame

- **Creation** from list of dictionaries
  - **Each dictionary** is associated to a **row**
  - **Index** is automatically set to a progressive number
    - Unless explicitly passed as parameter (index=…)
- Example:

```
In [1]:   dic_list = [{'c1':i, 'c2':2*i} for i in range(3)]
          df = pd.DataFrame(dic_list)
          print(df)
```

```
Out[1]:      c1     c2
          0   0      0
          1   1      2
          2   2      4
```

Pandas DataFrame

- **Creation** from 2D Numpy array
- Example:

In [1]:

```
arr = np.arange(6).reshape((3,2))
df = pd.DataFrame(arr, columns=['c1', 'c2'],
                       index=['a', 'b', 'c'])
print(df)
```

Out[1]:

```
    c1    c2
a    0     1
b    2     3
c    4     5
```

DataFrames and I/O

- Load DataFrame from **csv** file
  - Allows specifying the column **delimiter (sep)**
  - Automatically read **header** from first line of the file (after **skipping** the specified number of rows)
  - Column data types are inferred

```
df = pd.read_csv('./mycsv.csv', sep=',', skiprows=1)
```

mycsv.csv

```
MyTitle
c1,c2,c3
0,1,2
3,4,5
6,7,8
```

$\longrightarrow$

	c1	c2	c3
0	0	1	2
1	3	4	5
2	6	7	8

DataFrames and I/O

- Load DataFrame from **csv** file
  - If it contains **null** values, you can specify how to recognize them
  - Empty columns are converted to "NaN" (Not a Number)
    - Using np.nan (NumPy's representation of NaN)
  - The string 'NaN' is automatically recognized

```
df = pd.read_csv('./mycsv.csv', sep=',',
                 na_values=['no info', 'x'])
```

mycsv.csv

```
c1,c2,c3
0,no info,
3,4,5
6,x,NaN
```

	c1	c2	c3
0	0	NaN	NaN
1	3	4.0	5.0
2	6	NaN	NaN

type(np.nan) → float,
hence c2 and c3 are floats

DataFrames and I/O

- **Save** DataFrame to csv

```
df.to_csv('./savedcsv.csv', sep=',')
```

	c1	c2	c3
0	0	NaN	2
1	3	4	5
2	6	NaN	NaN

savedcsv.csv

```
 c1,c2,c3
0,0,,2
1,3,4,5
2,6,,
```

- Use **index=False** to avoid writing the index

```
df.to_csv('./savedcsv.csv', sep=',', index=False)
```

DataFrames and I/O

- Many other data types are supported
  - Excel, HTML, HDF5, SAS, JSON
- Check the pandas documentation
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

Pandas DataFrame

- Obtaining **column names** and **index** from a DataFrame

Index	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1

In [2]:

```
print(df.columns)  # Index object with column names
print(df.index)    # Index object
```

Out[2]:

```
Index(['Price', 'Quantity', 'Liters'], dtype='object')
Index(['a', 'b', 'c'], dtype='object')
```

Pandas DataFrame

- **Accessing** DataFrame data
  - Get a 2D Numpy array

Index	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1

In [2]:

```
print(df.values)   # Numpy array with data
```

Out[2]:

```
array([[1.0, 5.0, 1.5],
       [1.4, 10.0, 0.3],
       [5.0, 8.0, 1.0]])
```

Pandas DataFrame

- **Accessing** DataFrames
  - Access a DataFrame column
  - Access rows and columns with indexing
    - **df.loc**
      - **Explicit** index
      - Slicing, masking, fancy indexing
    - **df.iloc**
      - **Implicit** index
- Whether a **copy** or **view** will be returned it depends on the context
  - Usually it is difficult to make assumptions
  - https://pandas-docs.github.io/pandas-docs-travis/user_guide/indexing.html

Pandas DataFrame

- **Accessing** DataFrame columns
  - Returns a **Series** with column data

Index	Price	Quantity	Liters
a	1.0	5	1.5
b	1.4	10	0.3
c	5.0	8	1

In [1]:

```
df['Quantity']
```

Out[1]:

```
a      5
b     10
c      8
```

Pandas DataFrame

- **Accessing** single DataFrame **row** by index
  - **loc** (explicit), **iloc** (implicit)
  - Return a **Series** with an element for each column

In [1]:
```
print(df.loc['a'])          # Get the first row (explicit)
print(df.iloc[0])           # Get the first row
```

Out[1]:
```
Price    1.0
Quantity 5.0
Liters   1.5


Price    1.0
Quantity 5.0
Liters   1.5
```

Pandas DataFrame

- **Accessing** DataFrames with **slicing**
  - Allows selecting rows and columns

```
In [1]:  print(df.loc['b':'c', 'Quantity':'Liters'])
```

```
Out[1]:
        Quantity   Liters
b             10      0.3
c              8        1
```

Pandas DataFrame

- **Accessing** DataFrames with **masking**
  - Select rows based on a condition

Index	Price	Quantity	Liters
a	1.0	**5**	**1.5**
b	1.4	10	0.3
c	5.0	8	1

In [1]:
```
mask = (df['Quantity']<10) & (df['Liters']>1)
df.loc[mask, 'Quantity':]   # Use masking and slicing
```

Out[1]:
```
    Quantity   Liters
a          5      1.5
```

Pandas DataFrame

- **Accessing** DataFrames with **fancy indexing**
  - To select **columns**…

Index	Price	Quantity	Liters
a	**1.0**	5	**1.5**
b	1.4	10	0.3
c	5.0	8	1

In [1]:
```
mask = (df['Quantity']<10) & (df['Liters']>1)
df.loc[mask, ['Price','Liters']]   # Use masking and fancy
```

Out[1]:
```
    Price       Liters
a     1.0          1.5
```

Pandas DataFrame

- **Accessing** DataFrames with **fancy indexing**
  - To select **rows** and **columns**…

Index	Price	Quantity	Liters
a	**1.0**	5	**1.5**
b	1.4	10	0.3
c	**5.0**	8	**1**

In [1]:
```
df.loc[['a', 'c'], ['Price','Liters']]
```

Out[1]:
```
     Price       Liters
a       1.0         1.5
c       5.0         1.0
```

Pandas DataFrame

- **Assign value** to selected items

In [1]:
```
df.loc[['a', 'c'], ['Price','Liters']] = 0
```

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

Pandas DataFrame

- **Add new column** to DataFrame
  - DataFrame is modified **inplace**

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

$\rightarrow$

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

In [1]:
```
df['Available'] = pd.Series([True, False, True],
                            index=['a', 'b', 'c'])
```

  - If the DataFrame already has a column with the specified name, then this is **replaced**

Pandas DataFrame

- **Add new column** to DataFrame
  - It is also possible to assign directly a **list**

Index	Price	Quantity	Liters
a	**0.0**	5	**0.0**
b	1.4	10	0.3
c	**0.0**	8	**0.0**

$\rightarrow$

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

In [1]:
```
df['Available'] = [True, False, True]
```

Pandas DataFrame

- **Drop column(s)**
  - Returns a **copy** of the updated DataFrame
    - Unless `inplace=True`, in which case the original DataFrame is modified
      - This applies to many pandas methods -- always check the documentation!

Index	Price	Quantity	Liters	Available
a	1.0	~~5~~	~~1.5~~	True
b	1.4	~~10~~	~~0.3~~	False
c	5.0	~~8~~	~~1~~	True

```
In [1]: df = df.drop(columns=['Quantity', 'Liters'])
```

Pandas DataFrame

- **Rename column(s)**
  - Use a **dictionary** which maps old names with new names
  - Returns a **copy** of the updated DataFrame

Index	Price	Quantity	Liters	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

$\rightarrow$

Index	Price	**nItems**	**[L]**	Available
a	1.0	5	1.5	True
b	1.4	10	0.3	False
c	5.0	8	1	True

In [1]:
```
df = df.rename(columns={'Quantity': 'nItems',
                        'Liters': '[L]'})
```

Computation with Pandas

- Unary operations on Series and DataFrames
  - exponentiation, logarithms, …
- Operations between Series and DataFrames
  - Operations are performed **element-wise**, being aware of their **indices/columns**
- Aggregations (min, max, std, …)

Computation with Pandas

- Unary operations on Series and DataFrames
  - They work with any **Numpy** ufunc
  - The operation is applied to each element of the Series/DataFrame
- Examples:
  - res = my_series/4 + 1
  - res = np.abs(my_series)
  - res = np.exp(my_dataframe)
  - res = np.sin(my_series/4)
  - …

Computation with Pandas

- Operations between Series (+,-,*,/)
  - Applied element-wise after **aligning indices**
    - Index elements which do not match are set to **NaN** (Not a Number)
  - Example:
    - res = my_series1 + my_series2

After index alignment index in the result is **sorted**

Index	
b	3
a	1
c	**10**

my_series1

Index	
a	1
b	3
d	**30**

my_series2

Index	
a	2
b	6
c	**NaN**
d	**NaN**

res

Computation with Pandas

- Operations between DataFrames
  - Applied element-wise after **aligning indices** and **columns**
  - Example (align **index**):
    - res = my_dataframe1 + my_dataframe2

Index in the result is **sorted**

Index	Total	Quantity
b	3	4
a	1	2
c	**10**	**20**

my_dataframe1

Index	Total	Quantity
a	1	2
b	3	4
d	**30**	**40**

my_dataframe2

Index	Total	Quantity
a	2	4
b	6	8
c	**NaN**	**NaN**
d	**NaN**	**NaN**

res

Computation with Pandas

- Operations between DataFrames
  - Example (align **columns**)
    - res = my_dataframe1 + my_dataframe2

Columns in the result are **sorted**

Index	Total	Quantity
a	1	**2**
b	3	**4**
c	5	**6**

my_dataframe1

Index	Total	Price
a	1	**2**
b	3	**4**
c	5	**6**

my_dataframe2

Index	Price	Quantity	Total
a	**NaN**	**NaN**	2
b	**NaN**	**NaN**	6
c	**NaN**	**NaN**	10

res

Computation with Pandas

- Operations between DataFrames and Series
  - The operation is applied between the Series and each **row** of the DataFrame
    - Follows **broadcasting** rules
  - Example:
    - res = my_dataframe1 + my_series1

Index	Total	Quantity
a	1	2
b	3	4
c	5	6

my_dataframe1

Index	
Total	1
Quantity	2

my_series1

Index	Total	Quantity
a	2	4
b	4	6
c	6	8

res

Computation with Pandas

- Pandas Series and DataFrames allow performing aggregations
  - mean, std, min, max, sum
- Examples

In [1]:
```
my_series.mean()   # Return the mean of Series elements
```

  - For DataFrames, aggregate functions are applied **column-wise** and return a Series

In [1]:
```
my_df.mean()       # Return a Series
```

Computation with Pandas

- Example of **aggregations** with DataFrames: z-score normalization

In [1]:
```
mean_series = df.mean()
std_series = df.std()
df_norm = (df-mean_series)/std_series
```

Index	Total	Quantity
a	1	2
b	3	4
c	5	6

my_dataframe1

Index	
Total	3.0
Quantity	4.0

mean_series

Index	
Total	2.0
Quantity	2.0

std_series

Missing values

- Represented with **sentinel** value
  - **None**: Python null value
  - **np.nan**: Numpy Not A Number
- None is a Python **object**:
  - np.array([4, None, 5]) has dtype=Object
- np.NaN is a **Floating point** number
  - np.array([4, np.nan, 5]) has dtype=Float
- Using **nan** achieves better **performances** when performing numerical computations

Missing values

- Pandas supports both **None** and **NaN**, and automatically converts between them when appropriate
- Example:

```
In [1]:  pd.Series([4, None, 5, np.nan])
```

```
Out[1]:  0    4.0
         1    NaN
         2    5.0
         3    NaN
         dtype=float64
```

Missing values

- Operating on missing values (for Series and DataFrames)
  - isnull()
    - Return a boolean mask indicating null values
  - notnull()
    - Return a boolean mask indicating not null values
  - dropna()
    - Return filtered data containing null values
  - fillna()
    - Return new data with filled or input missing values

Missing values

- Operating on missing values: **isnull, notnull**
  - Return a new Series/DataFrame with the same shape as the input

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.isnull()
```

Out[1]:
```
0     False
1     True
2     False
3     True
dtype=bool
```

Missing values

- Operating on missing values: **dropna**
  - For Series it removes null elements

In [1]:
```
s1 = pd.Series([4, None, 5, np.nan])
s1.dropna()
```

Out[1]:
```
0    4.0
2    5.0
dtype=float64
```

Missing values

- Operating on missing values: **dropna**
  - For DataFrames it removes **rows** that contain at least a missing value (default behaviour)
    - Passing `how=all` removes rows if they contain all NaN's

Index	Total	Quantity
a	1	2
b	3	NaN
c	5	6

$\longrightarrow$

Index	Total	Quantity
a	1	2
c	5	6

  - Alternatively, it is possible to remove columns

```
dropped_df = df.dropna(axis='columns')
```

Missing values

- Operating on missing values: **fillna**
  - Fill null fields with a specified value (for both Series and DataFrames)

In [1]:

```
s1 = pd.Series([4, None, 5, np.nan])
s1.fillna(0)
```

Out[1]:

```
0    4.0
1    0.0
2    5.0
3    0.0
dtype=float64
```

Missing values

- Operating on missing values: **fillna**
  - The parameter **method** allows specifying different filling techniques
    - **ffill**: propagate last valid observation forward
    - **bfill**: use next valid observation to fill gap

In [1]:

```
s1 = pd.Series([4, None, 5, np.nan])
s1.fillna(method='ffill')
```

Out[1]:

```
0    4.0
1    4.0
2    5.0
3    5.0
```

Notebook Examples

- 3-Pandas Examples.ipynb
  - 1. Accessing DataFrames and Series

Grouping data

- Pandas provides the equivalent of the SQL group by statement
- It allows the following operations:
  - **Iterating** on groups
  - **Aggregating** the values of each group (mean, min, max, …)
  - **Filtering** groups according to a condition

Grouping data

- **Applying** group by
  - Specify the column(s) where you want to group (**key**)
  - Obtain a DataFrameGroupBy object

```
df = pd.DataFrame({'k' : ['a','b','a','b'],
                   'c1': [2,10,3,15], 'c2' : [4,20,5,30]})
grouped_df = df.groupby('k')   # 2 groups: 'a' and 'b'
```

Index	**k**	c1	c2
0	a	2	4
1	b	10	20
2	a	3	5
3	b	15	30

$\longrightarrow$

Index	**k**	c1	c2
0	a	2	4
2	a	3	5
1	**b**	**10**	**20**
3	**b**	**15**	**30**

Grouping data

- **Iterating** on groups
  - Each group is a subset of the original DataFrame

In [1]:

```
for key, group_df in grouped_df:
    print(key)
    print(group_df)
```

Out[1]:

```
a
    k1  c1   c2
0   a    2    4
2   a    3    5
b
    k1  c1   c2
1   b   10   20
3   b   15   30
```

Index	**k1**	c1	c2
0	a	2	4
2	a	3	5

Index	**k1**	c1	c2
1	b	10	20
3	b	15	30

Grouping data

- **Aggregating** by group (min, max, mean, std)
  - The output is a DataFrame with the result of the aggregation for each group

In [1]:
```
grouped_df.mean()  # Mean, separately for each group
```

Out[1]:
```
k     c1    c2
a    2.5   4.5
b   12.5  25.0
```

Index	k1	c1	c2
0	a	2	4
2	a	3	5

Index	k1	c1	c2
1	b	10	20
3	b	15	30

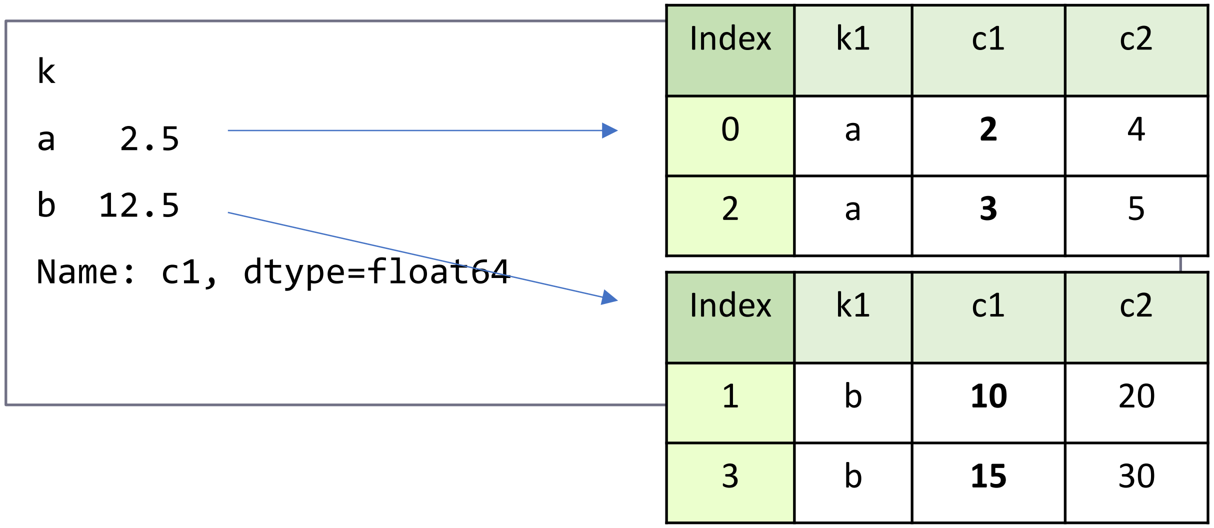
The index of the result is the key of each group

Grouping data

- **Aggregating** a single column by group
  - The output is a Series with the result of the aggregation for each group

In [1]:
```
grouped_df['c1'].mean()
```

Out[1]:
```
k
a    2.5
b   12.5
Name: c1, dtype=float64
```

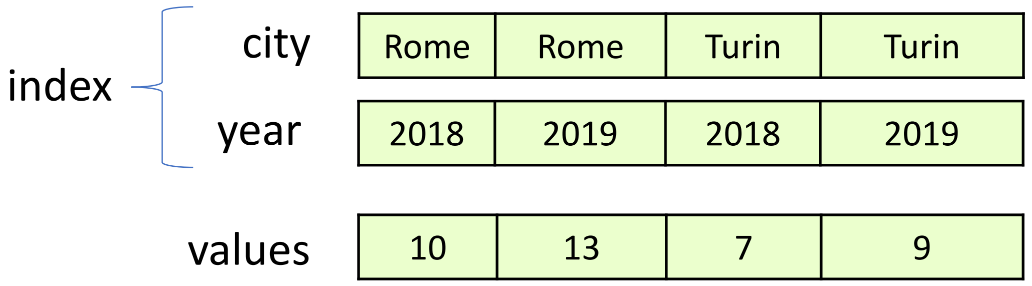


Index	k1	c1	c2
0	a	**2**	4
2	a	**3**	5

Index	k1	c1	c2
1	b	**10**	20
3	b	**15**	30

Multi-Index

- **Multi-Index** allows specifying an index hierarchy for
  - Series
  - DataFrames
- Example: index a Series by city and year



index	city	Rome	Rome	Turin	Turin
	year	2018	2019	2018	2019
	values	10	13	7	9

Multi-Index

- **Multi-indexed DataFrame**
  - Specify a multi-index for **rows**
  - **Columns** can be multi-indexed as well

		Humidity		**Temperature**	
		max	min	max	min
Turin	2018	33	48	6	33
	2019	35	45	5	35
Rome	2018	40	59	2	33
	2019	41	57	3	34

Multi-Index

- **Reset Index:** transform index to DataFrame columns and create new (single level) index

In [1]:

```
df.index.names = ['city', 'year']
df_reset = df.reset_index()
print(df_reset)
```

Out[1]:

```
      city   year  c1      c2
                   a   b   a   b
0     Rome   2018  0   1   2   3
1     Rome   2019  4   5   6   7
2    Turin   2018  8   9  10  11
3    Turin   2019 12  13  14  15
```

New index

Uncovered Topics

DataFrames and I/O

- Load DataFrame from **json** file

```
df = pd.read_json('./myjson.json')
```

myjson.json

```
{"c1":{"0":0, "1":3, "2":6},
"c2":{"0":null, "1":4, "2":null},
"c3":{"0":2, "1":5, "2":null}}
```

$\rightarrow$

	c1	c2	c3
0	0	NaN	2
1	3	4	5
2	6	NaN	NaN

- Use **pd.to_json(path)** to save a DataFrame in json format

Combining Pandas objects

- Pandas provides 2 methods for combining Series and DataFrames
  - concat()
    - Concatenate a sequence of Series/DataFrames
  - append()
    - Append a Series/DataFrame to the specified object

Combining Pandas objects

- Concatenating 2 Series
  - Index is preserved, even if **duplicated**
    - There is nothing that prevents duplicate indices in pandas!

In [1]:

```
s1 = pd.Series(['a', 'b'], index=[1,2])
s2 = pd.Series(['c', 'd'], index=[1,2])
pd.concat((s1, s2))
```

Out[1]:

```
1     a
2     b
1     c
2     d
dtype=object
```

Combining Pandas objects

- Concatenating 2 Series
  - To avoid duplicates use **ignore_index**

In [1]:

```
s1 = pd.Series(['a', 'b'], index=[1,2])
s2 = pd.Series(['c', 'd'], index=[1,2])
pd.concat((s1, s2), ignore_index=True)
```

Out[1]:

```
0    a
1    b
2    c
3    d
dtype=object
```

Combining Pandas objects

- Concatenating 2 DataFrames
  - Concatenate **vertically** by default

```
In [1]:  pd.concat((df1, df2))
```

Index	Total	Quantity
a	1	2
b	3	4

Index	Total	Quantity
c	5	6
d	7	8

$\rightarrow$

Index	Total	Quantity
a	1	2
b	3	4
c	5	6
d	7	8

Combining Pandas objects

- Concatenating 2 DataFrames
  - Missing columns are filled with NaN

In [1]:
```python
pd.concat((df1, df2))
```

Index	Total	Quantity
a	1	2
b	3	4

Index	Total	Quantity	Liters
c	5	6	1
d	7	8	2

$\rightarrow$

Index	Total	Quantity	Liters
a	1	2	**NaN**
b	3	4	**NaN**
c	5	6	1.0
d	7	8	2.0

Combining Pandas objects

- The **append**() method is a shortcut for concatenating DataFrames
  - Returns the result of the concatenation

```
In [1]:  df_concat = df1.append(df2)
```

is equivalent to:

```
In [1]:  df_concat = pd.concat((df1, df2))
```

Combining Pandas objects

- Joining DataFrames with relational algebra: **merge**()
  - Merge on:
    - The column(s) with same name in the two DFs, by default
    - Specific columns, by specifying `on=columns`
      - `left_on` and `right_on` may also be used
    - The indices, if `left_index/right_index` are True
      - This preserves the indices (discarded otherwise)
  - Depending on the DataFrames, a **one-to-one**, **many-to-one** or **many-to-many** join can be performed
    - `validate='1:1'|'1:m'|'m:1'|'m:m'` to enforce the specific merge

```
In [1]:  joined_df = pd.merge(df1, df2)
```

Combining Pandas objects

- Examples (1)

`pd.merge(df1, df2)` ➔ merge on columns in common, [“k1”]

Index	**k1**	c2
i1	0	a
i2	1	b

Index	**k1**	c3
i1	1	b1
i2	0	a1

→

Index	**k1**	c2	c3
0	0	a	a1
1	1	b	b1

`pd.merge(df1, df2, right_index=True, left_index=True)` ➔ merge on index

Index	k1	c2
i1	0	a
i2	1	b
i3	0	c
i4	1	d

Index	k1	c3
i1	1	b1
i2	0	a1

→

Index	k1_x	c2	k1_y	c3
i1	0	a	1	b1
i2	1	b	0	a1

Combining Pandas objects

PoliTo DBMG

- Examples (2)

`pd.merge(df1, df2)` ➔ performs a one-to-one merge

Index	**k1**	c2
i1	0	a
i2	1	b

Index	**k1**	c3
i1	1	b1
i2	0	a1

→

Index	**k1**	c2	c3
0	0	a	a1
1	1	b	b1

`pd.merge(df1, df2)` ➔ performs a many-to-one merge

Index	**k1**	c2
i1	0	a
i2	1	b
i3	0	c
i4	1	d

Index	**k1**	c3
i1	1	b1
i2	0	a1

→

Index	**k1**	c2	c3
0	0	a	a1
1	0	c	a1
2	1	b	b1
3	1	d	b1

Grouping data

- **Filtering** data by group
  - The filter is expressed with a lambda function working with each group DataFrame (x)

```
In [1]:
# Keep groups for which column c1 has a mean > 5
grouped_df.filter(lambda x: x['c1'].mean()>5)
```

```
Out[1]:
      k     c1    c2
1     b     10    20
3     b     15    30
```

~~Index~~	~~k1~~	~~c1~~	~~c2~~
~~0~~	~~a~~	~~**2**~~	~~4~~
~~2~~	~~a~~	~~**3**~~	~~5~~

```
mean = 2.5
x: filtered
out
```

Index	k1	c1	c2
1	b	**10**	20
3	b	**15**	30

```
mean = 12.5
x: kept in
the result
```

Pivoting

- Pivoting allows inspecting relationships within a dataset
- Suppose to have the following dataset:

```
df = pd.DataFrame({'type':['a','b','b','a','b','a','b','a'],
                   'class':[3,2,3,3,2,1,1,2],
                   'fail':[1,1,1,0,1,0,0,0]})
```

Index	type	class	fail
0	a	3	1
1	b	2	1
2	b	3	1
3	a	3	0
4	b	2	1
5	a	1	0
6	b	1	0
7	a	2	0

- that shows **failures** for sensors of a given type and class during some test

Pivoting

```
In [1]:  df.pivot_table('fail', index='type',
                          columns='class', aggfunc='sum')
```

- Shows the number of **failures** for all the combinations of **type** and **class**

```
Out[1]:  class  1  2  3
         type
         a      0  0  1
         b      0  2  1
```

2 sensors of type b and class 2 had some failure

Index	type	class	fail
0	a	3	1
1	**b**	**2**	**1**
2	b	3	1
3	a	3	0
4	**b**	**2**	**1**
5	a	1	0
6	b	1	0
7	a	2	0

Pivoting

```
In [1]: df.pivot_table('fail', index='type',
                       columns='class', aggfunc='mean')
```

- Shows the percentage of **failures** for all the combinations of **type** and **class**

```
Out[1]: class    1    2    3
        type
        a      0.0  0.0  0.5
        b      0.0  1.0  1.0
```

50% of sensors of type a and class 3 had some failure

Index	type	class	fail
0	**a**	**3**	**1**
1	b	2	1
2	b	3	1
3	**a**	**3**	**0**
4	b	2	1
5	a	1	0
6	b	1	0
7	a	2	0

Multi-Index

- Building a **multi-indexed Series**

In [1]:

```
ix = [['Rome', 'Rome', 'Turin', 'Turin'],
      ['2018', '2019', '2018', '2019']]
s1 = pd.Series([10,13,7,9], index=ix)
s1 = s1.sort_index()  # Multi-Index must be sorted
                      # if you want to use slicing
print(s1)
```

Out[1]:

```
Rome    2018    10
        2019    13
Turin   2018     7
        2019     9
```

Multi-Index

- **Naming** index levels

In [1]:

```
s1.index.names=['city', 'year']
print(s1)
```

Out[1]:

```
city   year
Rome   2018    10
       2019    13
Turin  2018     7
       2019     9
```

Multi-Index

- **Accessing index levels**
  - **Slicing** and **simple indexing** are allowed
  - Slicing on index levels follows Numpy rules

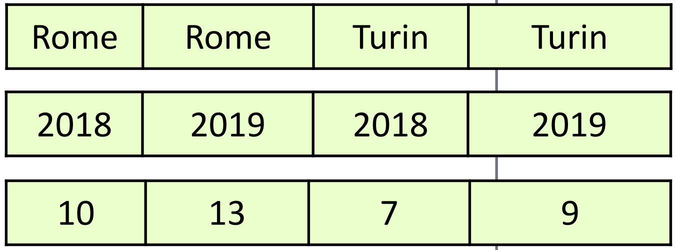
In [1]:

```
print(s1.loc['Rome'])      # Outer index level
print(s1.loc[:,'2018'])    # All cities, only 2018
```

Out[1]:

```
year
2018     10
2019     13

city
Rome      10
Turin      7
```



Rome	Rome	Turin	Turin
2018	2019	2018	2019
10	13	7	9

Multi-Index

- **Accessing index levels (Examples)**

In [1]:
```
print(s1.loc['Turin', '2018':'2019'])
print(s1[s1>10])     # Masking
```

Out[1]:
```
city    year
Turin   2018     7
        2019     9


city   year
Rome   2019     13
```

Rome	Rome	Turin	Turin
2018	2019	2018	2019
10	13	7	9

Multi-Index

- **Multi-indexed DataFrame: creation**

In [1]:

```
ix = [['Rome', 'Rome', 'Turin', 'Turin'],
      ['2018', '2019', '2018', '2019']]
cols = [['c1','c1','c2','c2'],['a','b','a','b']]
data = np.arange(16).reshape((4,4))
df = pd.DataFrame(data, index=ix, columns=cols)
print(df)
```

Out[1]:

```
             c1      c2
             a   b   a   b
Rome   2018  0   1   2   3
       2019  4   5   6   7
Turin  2018  8   9  10  11
       2019 12  13  14  15
```

Multi-Index

- **Multi-indexed DataFrame:** access with **outer** index level

In [1]:

```
print(df['c1'])               # Access by column
print(df.loc['Rome', 'c1']) # Access rows and cols
```

Out[1]:

```
             a   b
Rome   2018  0   1
       2019  4   5
Turin  2018  8   9
       2019 12  13

      a  b
2018  0  1
2019  4  5
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

Multi-Index

- **Multi-indexed DataFrame:** access with **outer** and **inner** index levels

In [1]:
```
df['c1', 'a']                   # Access by column
```

Out[1]:
```
Rome   2018    0
       2019    4
Turin  2018    8
       2019   12
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

Multi-Index

- **Multi-indexed DataFrame:** access with **outer** and **inner** index levels

In [1]:
```
ix = pd.IndexSlice
df.loc[ix['Rome', '2018'], ix['c1':'c2', 'a']]
```

Out[1]:
```
c1  a    0
c2  a    2
```

		c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

Multi-Index

- **Set Index:** transform columns to Multi-Index
  - Inverse function of reset_index()

```
In [1]: df_reset.set_index(['city', 'year'])
```

	city	year	c1		c2	
			a	b	a	b
0	Rome	2018	0	1	2	3
1	Rome	2019	4	5	6	7
2	Turin	2018	8	9	10	11
3	Turin	2019	12	13	14	15

$\rightarrow$

city	**year**	c1		c2	
		a	b	a	b
Rome	**2018**	0	1	2	3
	2019	4	5	6	7
Turin	**2018**	8	9	10	11
	2019	12	13	14	15

New index

Multi-Index

- **Unstack:** transform multi-indexed Series to a Dataframe

```
myseries.unstack()
```

city	year	
Rome	2018	0
	2019	4
Turin	2018	8
	2019	12

→

	2018	2019
Rome	0	4
Turin	8	12

Multi-Index

- **Stack:** inverse function of unstack()
  - From DataFrame to multi-indexed Series

```
mydataframe.stack()
```

	2018	2019
Rome	0	4
Turin	8	12

Rome	2018	0
	2019	4
Turin	2018	8
	2019	12

Multi-Index

- **Aggregates on multi-indices**
  - Allowed by passing the **level** parameter
  - Level specifies the **row granularity** at which the result is computed

```
my_dataframe.max(level='city')
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

→

city	c1		c2	
	a	b	a	b
Rome	4	5	6	7
Turin	12	13	14	15

Multi-Index

- **Aggregates on multi-indices**

```
my_dataframe.max(level='year')
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

$\rightarrow$

year	c1		c2	
	a	b	a	b
2018	8	9	10	11
2019	12	13	14	15

Multi-Index

- **Aggregates on multi-indices**
  - Can also aggregate columns
    - Specify axis=1

```
my_dataframe.max(axis=1, level=0)
```

city	year	c1		c2	
		a	b	a	b
Rome	2018	0	1	2	3
	2019	4	5	6	7
Turin	2018	8	9	10	11
	2019	12	13	14	15

$\rightarrow$

city	year	c1	c2
Rome	2018	1	3
Rome	2019	5	7
Turin	2018	9	11
Turin	2019	13	15