Python for IoT Data Analytics

Data acquisition with Pandas

EIT4SEMM PHD Program, July 2023

Prof. Marco Di Felice

Department of Computer Science and Engineering, University of Bologna

marco.difelice3@unibo.it

Problem

`tetuan.csv`

Environmental data collected in Tetuan city

DateTime	Temperature	Humidity	Wind Speed
1/1/2017 0:00	6.559	73.8	83
1/1/2017 0:10	6.414	74.5	83
1/1/2017 0:20	6.313	74.5	0.08
1/1/2017 0:30	6.121	75	83
1/1/2017 0:40	5.921	75.7	81
1/1/2017 0:50	5.853	76.9	81
1/1/2017 1:00	5.641	77.7	0.08
1/1/2017 1:10	5.496	78.2	85

TASK 1 (Generic data): LECTURE 1

Load the dataset into memory into a Python data structure

TASK 2 (Generic data): LECTURE 1

Compute the avg temperature

TASK 3 (Timeseries) LECTURE 2

Compute the avg temperature every minute

TASK 4 (Timeseries) LECTURE 2

Resample the temp values with a different frequency

TASK 5 (Timeseries) LECTURE 2

Plot the time-series and export the plot

TASK 6 (Timeseries) LECTURE 3-4

Forecast the next n temp values

Python data structures

BUILTIN DATA STRUCTURES

- **Lists**
  - Mutable, heterogenous list of values

```
l = [1,2,3,5,6, "Bologna"]
```

- **Dictionaries**
  - List of keys-values

```
d = {"lat":43.12, "long":11.23}
```

- **Tuples**
  - Immutable, heterogenous list of values

```
t = (1,2,3)
```

- **Sets**
  - Immutable, unique values

```
s= {1,2,3}
```

Pandas Library: Overview

- **Python software library for data processing and analysis**
  - Developed by Wes McKinney in 2008
  - Open source project since 2010
- **Used in tandem with:**
  - Numerical computing tools (e.g. NumPy and SciPy)
  - Analytical models (e.g. Statsmodel and Scikit-learn)
  - Data visualization libraries (e.g. matplotlib)
- **Designed for working with:**
  - Tabular data (ordered collection of columns)
  - Heterogeneous data (columns can be of different types)
- **Import pandas module within the current project:**

```
import pandas as pd
```

Pandas Library: Overview

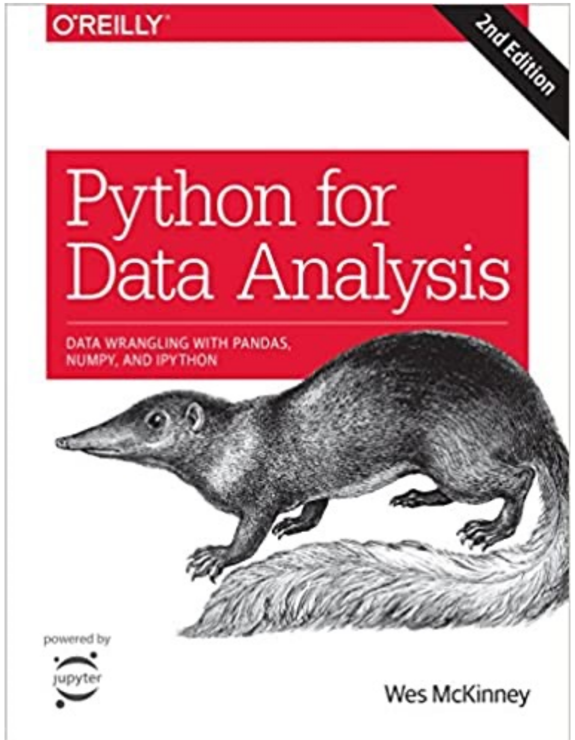
pandas

Tabular Data (2D)
Heterogeneous columns

NumPy

Multidimensional Data
Homogeneous columns

Pandas Library: Overview



Python for Data Analysis: Data Wrangling with Pandas, Numpy, and IPython

Author: Wes Mckinney

Official Pandas User Guide
https://pandas.pydata.org/docs/user_guide/index.html

Pandas Library: Overview

- **Built-in data-structures**
  - **Series**
  - DataFrame
- **Built-in methods**
  - Data Loading
  - Data Selection
  - Descriptive statistics
  - Data transforming
  - Data cleaning
  - Data wrangling: join, combine and concat
  - Data aggregation

Pandas Library: Overview

- **Series**
  - One-dimensional array-like object containing a sequence of values and an associated array of data labels, called its **index**.
  - When not specified, the index consists of integer values in range [0:N-1], where N is the lenght of the sequence of values.

Population (K) of provincial districts of Emilia

0	388
1	102
2	194
3	171
4	184

Series (with *implicit* index)

```
obj=pd.Series([388, 102,
194, 171, 184])
```

Bologna	388
Piacenza	102
Parma	194
Reggio Emilia	171
Modena	184

Series (with *explicit* index)

```
obj=pd.Series([388, 102,
   194, 171, 184],
 index=['Bologna','Piacen
  za', 'Parma', 'Reggio
   Emilia', 'Modena'])
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - **DataFrame**
- **Built-in methods**
  - Data Loading
  - Data Selection
  - Descriptive statistics
  - Data transforming
  - Data cleaning
  - Data wrangling: join, combine and concat
  - Data aggregation

Pandas Library: Overview

DataFrame

- Two-dimensional array-like object containing an ordered collection of columns (e.g. Series); each column can belong to a different type (numeric, string, boolean, etc).
- It has two indexes: a *column* index and a *row* index.
- It can be thought as a dict of dicts or a dict of Series, all sharing the same index

Series

	Population
Bologna	388
Piacenza	102
Parma	194
Reggio Emilia	171
Modena	184

\+

Series

	University
Bologna	UNIBO
Piacenza	POLIMI
Parma	UNIPR
Reggio Emilia	UNIMORE
Modena	UNIMORE

=

DataFrame

	Population	University
Bologna	388	UNIBO
Piacenza	102	POLIMI
Parma	194	UNIPR
Reggio Emilia	171	UNIMORE
Modena	184	UNIMORE

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

DataFrame

- Several possible data inputs to a DataFrame constructor: e.g. dict of Series, dict of arrays, lists, tuples, List of dicts, List of Series, List of Tuples, another DataFrame, NumPy array, …

DATAFRAME as a DICT of SERIES

```
>>> ser1=pd.Series([388, 102, 194, 171, 184], index=['Bologna','Piacenza', 'Parma', 'Reggio Emilia', 'Modena'])

>>> ser2=pd.Series(['UNIBO','POLIMI','UNIPR','UNIMORE','UNIMORE'], index=['Bologna','Piacenza', 'Parma', 'Reggio Emilia’, ‘Modena’])

>>> frame=pd.DataFrame({'population':ser1, 'university':ser2})
```

```
>>> print(frame)
              population university
Bologna              388      UNIBO
Piacenza             102     POLIMI
Parma                194      UNIPR
Reggio Emilia        171    UNIMORE
Modena               184    UNIMORE
```

Pandas Library: Overview

- **DataFrame**
  - Several possible data inputs to a DataFrame constructor: e.g. dict of Series, dict of arrays, lists, tuples, List of dicts, List of Series, List of Tuples, another DataFrame, NumPy array, …

DATAFRAME as a DICT of LIST

```
>>> data={'population':[388,102,194,171,184],
'university':['UNIBO','POLIMI','UNIPR','UNIMORE', 'UNIMORE']}

>>> frame=pd.DataFrame(data,index=['Bologna','Piacenza','Parma','Reggio
Emilia', 'Modena'])
```

```
>>> print(frame)
              population university
Bologna              388      UNIBO
Piacenza             102     POLIMI
Parma                194      UNIPR
Reggio Emilia        171    UNIMORE
Modena               184    UNIMORE
```

Pandas Library: Overview

DataFrame

- New columns can be added/modified by assignment; however, the value's length must match the length of the DataFrame, otherwise **missing values** are automatically inserted.

```
>>> prefix=pd.Series(['051','0521'],index=['Bologna','Parma'])
>>> frame['prefix']=prefix
```

```
>>> print(frame)
              population university prefix
Bologna              388      UNIBO    051
Piacenza             102     POLIMI    NaN
Parma                194      UNIPR   0521
Reggio Emilia        171    UNIMORE    NaN
Modena               184    UNIMORE    NaN
```

Pandas Library: Overview

DataFrame

- Each DataFrame has two **axes** (axis 0 → rows, axis 1 → columns)
- Each column can be associated to a *different data type* (heterogeneous table)

```
>>> df=pd.DataFrame({'office':['A','B','C','D'], 'size':[30,20,25,10]})
>>> df.columns
Index(['office', 'size'], dtype='object')
>>> df.index
RangeIndex(start=0, stop=4, step=1)
```

```
>>> df.dtypes
office    object
size      int64
dtype: object
```

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - **Data Loading**
  - Data Selection
  - Descriptive statistics
  - Data transforming
  - Data cleaning
  - Data wrangling: join, combine and concat
  - Data aggregation

Pandas Library: Overview

Data Loading

- Pandas features a number of functions for reading tabular data as a DataFrame object
- In most cases, Pandas performs *type inference* on the columns, since the data types are often not part of the data format
- In addition, it supports *data chunking* for very large files, i.e. the possibility to iterate through smaller chunks of a file instead of loading it in one go
- Some supported data sources:

  `read_csv`: text file with arbitrary delimiter (comma as default)

  `read_excel`: read tabular data from an Excel XLS or XLSX file

  `read_html`: read all tables found in a given HTML document

  `read_json`: read data from a JSON string representation

  `read_sql`: read the results of a SQL query as a Pandas DataFrame

  `read_hdf`: read Hierarchical Data Format (HDF) files

Pandas Library: Overview

- **Data Loading (CSV file)**
  - The simplest case is when loading a comma-separated (CSV) text file with column names appearing as first row of the file.

```
state,capital city,extension,code
Italy,Rome,301340,0039
Spain,Madrid,505990,0034
France,Paris,640679,0033
```

FILE.TXT

```
>>> df=pd.read_csv('file.txt')
>>> print(df)
    state  capital city  extension  code
0   Italy          Rome     301340    39
1   Spain        Madrid     505990    34
2  France         Paris     640679    33
```

```
>>> df=pd.read_csv('file.txt',
                   index_col='state')
>>> print(df)
        capital city  extension  code
state
Italy           Rome     301340    39
Spain         Madrid     505990    34
France         Paris     640679    33
```

Pandas Library: Overview

Data Loading (CSV file)

- More than 50 parameters to customize the data loading to the current data format!

Argument	Description
`path`	String indicating filesystem location, URL or file-like object
`delimiter`	Character or regular expression to split fields in each row
`header`	Row number to use as column number (default 0)
`index_column`	Column numbers or names to use as raw index in the DataFrame
`na_values`	Missing value placeholder, it will be replaced with NA
`nrows, skiprows`	Number of rows to read/skip from the beginning of file
`parse_dates`	Parse data to `datetime` (arguments: True, False or column numbers)

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

❑ Data Loading (CSV file)

- Missing data are marked by a sentinel value expressed by the na_values parameter (default: NA, NULL or empty space); they are replaced with NaN values in the loaded DataFrame

```
state,capital city,extension,code
Italy,Rome,301340,0039
Spain,Madrid,505990,0034
France,Paris,640679,0033
UK, London,,
USA, New York,,001
```

FILE.TXT

```
>>> df=pd.read_csv('file.txt', index_col='state')
>>> print(df)
       capital city  extension  code
state
Italy          Rome   301340.0  39.0
Spain        Madrid   505990.0  34.0
France        Paris   640679.0  33.0
UK           London        NaN   NaN
USA        New York        NaN   1.0
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

Writing Data to Text Format

- Exporting DataFrame to CSV is straightforward: simply invoke the `to_csv` method!
- Sentinel values for the missing values can be denoted with the `na_rep` argument
- Delimiters can be denoted with the `sep` argument (default: comma)

```
>>> df=pd.DataFrame({'a':[1,2,3,4,5],'b':[6,7,8,9,10]})
>>> df.to_csv('file.txt', sep='|')
```

```
|a|b
0|1|6
1|2|7
2|3|8
3|4|9
4|5|10
```

FILE.TXT

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - Data Loading
  - **Data Selection**
  - Descriptive statistics
  - Data transforming
  - Data cleaning
  - Data wrangling: join, combine and concat
  - Data aggregation

Pandas Library: Overview

- **Series selection** can be performed in two ways:
  - DataFrame as an Object, access the property with notation `object.property`
  - DataFrame as a Dictionary, access its columns using the indexing [ ] notation; in this case, slicing operations can be performed like in traditional Python arrays.

```
>>> print(frame.population)
Bologna          388
Piacenza         102
Parma            194
Reggio Emilia    171
Modena           184
Name: population, dtype: int64
```

```
>>> print(frame['population'])
Bologna          388
Piacenza         102
Parma            194
Reggio Emilia    171
Modena           184
Name: population, dtype: int64
```

Pandas Library: Overview

- **Pandas-specific operators for rows/columns selection**:
  - `dataframe.iloc[where_i,where_j]`
  - **index-based** selection, i.e. select rows and columns by integer positions
  - `where_i` is the row selection (: otherwise), `where_j` is the column selection (can be omitted).

SELECT FIRST ROW

```
>>> print (frame.iloc[0])
population      388
university    UNIBO
prefix          NaN
Name: Bologna, dtype: object
```

SELECT FIRST COLUMN

```
>>> print (frame.iloc[:,0])
Bologna          388
Piacenza         102
Parma            194
Reggio Emilia    171
Modena           184
Name: population, dtype: int64
```

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

- **Pandas-specific operators for rows/columns selection**:
  - `dataframe.loc[lab_i,lab_j]`
  - **label-based** selection, i.e. select rows and columns by index value
  - `lab_i` is the row index (: otherwise), `lab_j` is the column label (can be omitted)

SELECT FIRST ROW

```
>>> print (frame.loc['Bologna'])
population      388
university    UNIBO
prefix          NaN
Name: Bologna, dtype: object
```

SELECT FIRST COLUMN

```
>>> print (frame.loc[:,'population'])
Bologna           388
Piacenza          102
Parma             194
Reggio Emilia     171
Modena            184
Name: population, dtype: int64
```

Pandas Library: Overview

- **Conditional Filtering**
  - `dataframe.loc[condition]`
  - Condition is a boolean expression that is evaluated over each row, producing a Series of `true/false` values
  - The `loc[boolean array]` operator allows filtering out the rows associated to `false` values

```
>>> frame.loc[frame.population>150]
               population university prefix
Bologna               388      UNIBO    NaN
Parma                 194      UNIPR    NaN
Reggio Emilia         171    UNIMORE    NaN
Modena                184    UNIMORE    NaN
```

Pandas Library: Overview

- **Quick Row selection**
  - `dataframe.head(n)`: select the first n rows of the DataFrame
  - `dataframe.tail(n)`: select the last n rows of the DataFrame
  - `info()`: return the characteristics and shape of the DataFrame

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, Bologna to Modena
Data columns (total 2 columns):
---  ------      --------------  -----
 0   population  5 non-null      int64
 1   university  5 non-null      object
dtypes: int64(1), object(1)
memory usage: 120.0+ bytes
```

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - Data Loading
  - Data Selection
  - **Descriptive statistics**
  - Data transforming
  - Data cleaning
  - Data wrangling: join, combine and concat
  - Data aggregation

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

Computing Descriptive Statistics

- Pandas offers a wide set of mathematical methods for *reductions* or *summary statistics*
- Each method extracts a single value from a Series/column of a DataFrame
- Missing values can be skipped with the `skipna` option
- The axis to reduce over can be selected with the `axis` option (0= row, 1= column)

```
>>> df1=pd.DataFrame({'a':[1,2,3,4,5],'b':[6,7,8,9,10]})
>>> df1.mean()
```

```
a    3.0
b    8.0
```

```
>>> df1=pd.DataFrame({'a':[1,2,3,4,5],'b':[6,7,8,9,10]})
>>> df1.mean(axis=1)
```

```
0    3.5
1    4.5
2    5.5
3    6.5
4    7.5
```

Pandas Library: Overview

Computing Descriptive Statistics

- Pandas offers a wide set of mathematical methods for *reductions* or *summary statistics*

Method	Description
`count`	Return the non-NA values
`min, max`	Compute maximum and minimum values
`argmin,argmax, idmin, idmax`	Compute index locations (arg) or index labels (id) at which maximum/minimum values are obtained
`quantile`	Compute sample quantile between 0 and 1
`sum`	Sum of values
`mean, median`	Mean/median of values
`var, std`	Sample variance/standard deviation of values

Pandas Library: Overview

- **Computing Descriptive Statistics**
  - Other statistics like correlation and covariance are computed from pairs of arguments
  - The `corr/cov` method of Series computes respectively the correlation and covariance of the overlapping, non-NA aligned-by-index values in two Series
  - Similarly, DataFrame's `corr` and `cov` methods return a full correlation/covariance matrix.

```
>>> ser1=pd.Series([1,2,3,4,5])
>>> ser2=pd.Series([2,4,6,8,10])
>>> ser1.corr(ser2)
0.9999999999999999
>>> df1.cov(df1)
2.5
```

Pandas Library: Overview

Computing Descriptive Statistics

- Multiple summary statistics can be produced in one shot through the `describe` method

```
>>> df1=pd.DataFrame({'a':[1,2,3,4,5],'b':[6,7,8,9,10]})
>>> df1.describe()
```

```
              a          b
count  5.000000   5.000000
mean   3.000000   8.000000
std    1.581139   1.581139
min    1.000000   6.000000
25%    2.000000   7.000000
50%    3.000000   8.000000
75%    4.000000   9.000000
max    5.000000  10.000000
```

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - Data Loading
  - Data Selection
  - Descriptive statistics
  - **Data transforming**
  - Data cleaning
  - Data wrangling: join, combine and concat
  - Data aggregation

Pandas Library: Overview

Functional Mapping

- Transform each value of a Series/DataFrame according to a user-defined function
- `series.map(function)`: element-wise transformation for Series
- `dataframe.applymap(function)`: element-wise transformation for DataFrame
- `dataframe.map(function, axis=0|1)`: apply a function on one-dimensional arrays to each column or row

```
>>> df1=pd.DataFrame({'a':[1,2,3,4,5],'b':[6,7,8,9,10]})
>>> square=lambda x: x**2
>>> df1.applymap(square)
```

	a	b
0	1	36
1	4	49
2	9	64
3	16	81
4	25	100

```
>>> maxvalue=lambda x: x.max()
>>> df1.apply(maxvalue)
```

a	5
b	10

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - Data Loading
  - Data Selection
  - Descriptive statistics
  - Data transforming
  - **Data cleaning**
  - Data wrangling: join, combine and concat
  - Data aggregation

Pandas Library: Overview

Data Cleaning

- Pandas uses the sentinel value **NaN** (Not a Number) to represent missing values
- Pandas offers tow main options/methods to handle missing values:
- `pd.dropna`: drop rows containing a missing value
- `pd.fillna`: fill in missing data with some value or using an interpolation method (e.g. ffill)

```
>>> df=pd.DataFrame({'a':[1,2,3,np.nan,4],'b':[5,6,7,np.nan,np.nan]})
>>> cleaned=df.dropna()
>>> print(cleaned)
     a    b
0  1.0  5.0
1  2.0  6.0
2  3.0  7.0
```

Pandas Library: Overview

Data Cleaning

- Filling in missing data
- Pandas offers tow main options/methods to handle missing values:
- `pd.dropna`: drop rows containing a missing value
- `pd.fillna`: fill in missing data with some value or using an interpolation method (e.g. ffill)

```
>>> df=pd.DataFrame({'a':[1,2,3,nan,4],'b':[5,6,7,nan,nan]})
>>> cleaned=df.fillna({'a':0, 'b':5})
>>> print(cleaned)
```

```
     a    b
0  1.0  5.0
1  2.0  6.0
2  3.0  7.0
3  0.0  5.0
4  4.0  5.0
```

Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - Data Loading
  - Data Selection
  - Descriptive statistics
  - Data transforming
  - Data cleaning
  - **Data wrangling: join, combine and concat**
  - Data aggregation

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

Data Wrangling

- In many applications, data to analyze may be spread across a number of different files
- Pandas offers three main methods to *combine* different DataFrame/Series objects:

1. `pandas_concat`: smush elements of two different objects together along an axis
2. `pandas_merge`: work similar to **SQL join** operator, combine different datasets according to common values on specific columns
3. `pandas_join`: same as `pandas_merge`, it combines different datasets according to common values on the column indexes

Pandas Library: Overview

Data Concatenation

- `pandas_concat([df1,df2])` glues together values and indexes of DataFrames df1, df2
- It is typically used when *the objects to concatenate have the same fields/columns names*
- In case of DataFrames with different columns, missing values are added in the result

```
>>> df1=pd.DataFrame({'cities':['Bologna','Piacenza','Parma'],'population':[388,102,194]})
>>> df2=pd.DataFrame({'cities':['Reggio Emilia','Modena'],'population':[171,184]})
>>> pd.concat([df1,df2])
```

```
          cities  population
0        Bologna         388
1       Piacenza         102
2          Parma         194
0  Reggio Emilia         171
1         Modena         184
```

```
>>> df1=pd.DataFrame({'cities':['Bologna','Piacenza','Parma'],'population':[388,102,194]})
>>> df2=pd.DataFrame({'a':[3,4,5],'b':[6,7,8]})
>>> pd.concat([df1,df2])
```

```
     a    b    cities  population
0  NaN  NaN   Bologna       388.0
1  NaN  NaN  Piacenza       102.0
2  NaN  NaN     Parma       194.0
0  3.0  6.0       NaN         NaN
1  4.0  7.0       NaN         NaN
2  5.0  8.0       NaN         NaN
```

ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

- **Data Merging**
  - pandas_merge(df1,df2) combines datasets by linking rows according to equal values on join columns, specified by the on argument
  - If the join columns are not specified, merge uses the overlapping column names as keys
  - Like in SQL join operators, inner, left, right or outer join types can be executed

```
>>> df1=pd.DataFrame({'cities':['Bologna','Piacenza','Parma'],'population':[388,102,194]})
>>> df2=pd.DataFrame({'cities':['Bologna','Rimini','Parma'],'altitude':[54,6,57]})
>>> pd.merge(df1,df2,on=‘cities’)
    cities  population  altitude
0  Bologna         388        54
1    Parma         194        57
```

Pandas Library: Overview

Data Joining

- `pandas_join(df1,df2)` combines datasets by linking rows according to equal values on the index columns; it supports left join by default
- Other join version (`inner`, `right` or `outer`) can be specified by the how argument

```
>>> df1=pd.DataFrame({'population':[388,102,194],
'prefix':['051','0523','0521']},index=['Bologna','Piacenza','Parma'])

>>> df2=pd.DataFrame({'altitude':[54,6,57]},index=['Bologna','Rimini','Parma'])

>>> df1.join(df2)

          population prefix  altitude

Bologna          388    051      54.0

Piacenza         102   0523       NaN

Parma            194   0521      57.0
```

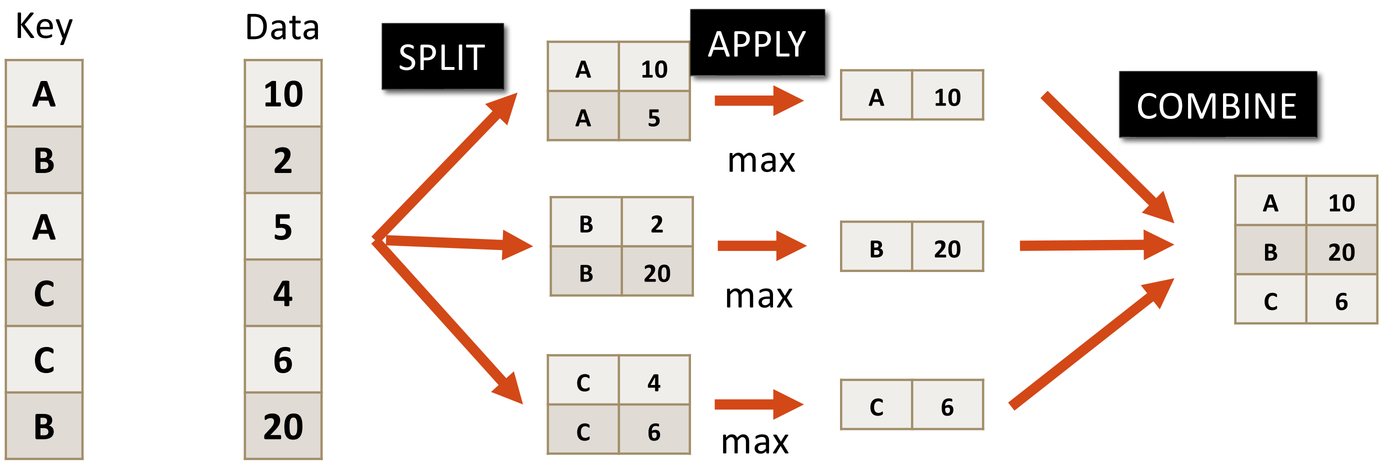
Pandas Library: Overview

- **Built-in data-structures**
  - Series
  - DataFrame
- **Built-in methods**
  - Data Loading
  - Data Selection
  - Descriptive statistics
  - Data transforming
  - Data cleaning
  - Data wrangling: join, combine and concat
  - **Data aggregation**

Pandas Library: Overview

Split-Apply-Combine

- Data grouping and aggregation are frequent operations when working with datasets
- Pandas supports the *split-apply-combine* paradigm:
  - **Split**: Split the Pandas object into groups based on one or more keys
  - **Apply**: A function is applied to each group, returning zero, one or more rows
  - **Combine**: The results of the functions are combined together into a new Pandas object



Pandas Library: Overview

Split-Apply-Combine

- DataFrame.groupby(key): Each grouping key can take many forms; in the simplest case, it is a single or a list of column names of the DataFrame
- The result is a GroupBy object which can be iterated upon, generating a sequence of 2-tuples

```
>>> df=pd.DataFrame({'cities':['Bologna', 'Pisa', 'Roma', 'Siena',
'Parma'],'region':['Emilia-Romagna','Toscana','Lazio','Toscana','Emilia-
Romagna']})

>>> grouped=df.groupby('region')

>>> for group in grouped:

...     print(group)
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Pandas Library: Overview

- **Split-Apply-Combine**
  - DataFrame.groupby(key): Each grouping key can take many forms; in the simplest case, it is a single or a list of column names of the DataFrame
  - Indexing a GroupBy object with a column name or array of column names has the effect of *column subsetting* for aggregation

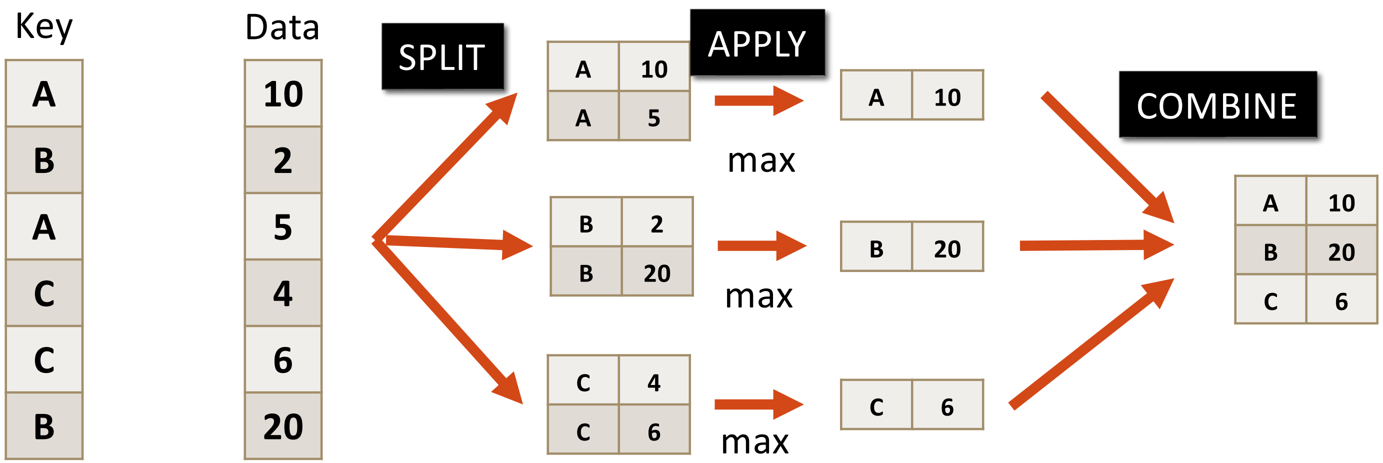
```
>>> df=pd.DataFrame({'cities':['Bologna', 'Pisa', 'Roma', 'Siena',
'Parma'],'region':['Emilia-Romagna','Toscana','Lazio','Toscana','Emilia-
Romagna'],'population':[390,91,2835,54,198]})

>>> grouped=df.groupby('region')['population’]

>>> for name in grouped:

        print(name)
```

```
('Emilia-Romagna', 0    390     4    198

Name: population, dtype: int64)

('Lazio', 2    2835

Name: population, dtype: int64)

('Toscana', 1    9    3   54

Name: population, dtype: int64)
```

Pandas Library: Overview

Split-Apply-Combine

- Data grouping and aggregation are frequent operations when working with datasets
- Pandas supports the *split-apply-combine* paradigm:
  - **Split**: Split the Pandas object into groups based on one or more keys
  - **Apply**: A function is applied to each group, returning zero, one or more rows
  - **Combine**: The results of the functions are combined together into a new Pandas object



Pandas Library: Overview

- **Split-Apply-Combine**
  - Pandas offers many built-in operators for scalar aggregation, i.e. extract a single scalar value from each group
  - `count, sum, mean, medium, std, var, min, max, prod, first, last`

```
>>> df=pd.DataFrame({'cities':['Bologna', 'Pisa', 'Roma', 'Siena',
'Parma'],'region':['Emilia-Romagna','Toscana','Lazio','Toscana','Emilia-
Romagna'],'population':[390,91,2835,54,198]})

>>>
grouped=df.groupby('region')['population'].sum()

>>> print(grouped)
```

```
region
Emilia-Romagna     588
Lazio             2835
Toscana            145
Name: population, dtype: int64
```

Pandas Library: Overview

Split-Apply-Combine

- The method apply provides general-purpose GroupBy objects processing
- It takes as argument a user-defined function, which is invoked on each group; the results of each invokation are then concatenated together.

```
>>> def filter(df):
...     return 0
>> df=pd.DataFrame({'cities':['Bologna', 'Pisa', 'Roma', 'Siena', 'Parma'],'region':['Emilia-Romagna','Toscana','Lazio','Toscana','Emilia-Romagna'],'population':[390,91,2835,54,198]})
>>> df.groupby('region').apply(filter)
```

```
region
Emilia-Romagna    0
Lazio             0
Toscana           0
dtype: int64
```