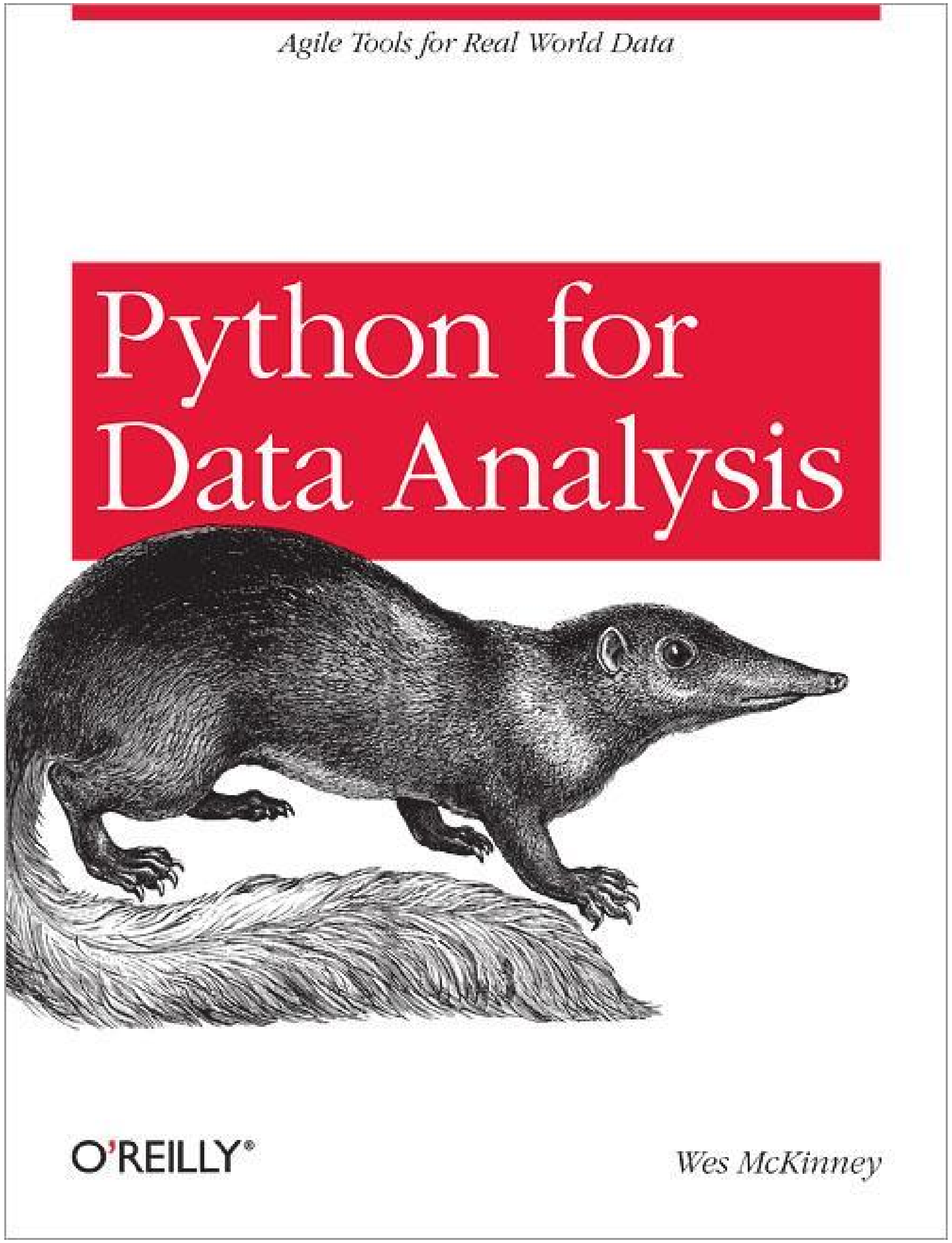
Python for Data Science

Pandas

ENSAE-ENSAI
Formation continue
(Cepe)

Outline

1 Introduction



1 Introduction

- Pandas is a **toolkit** to manipulate data
- It based on **numpy** and **scipy** for computations (faster)
- **DataFrame** is the key element (like in R)
- It became quickly the data manipulation and analysis **standard** over data ecosystem in Python

1 Introduction

- You need pandas when you work with tabular or structured data
- To:
    - *import* and *clean* data
    - *explore* data
    - *process* and *prepare* data
    - and *analyse*

1 Introduction: key features

- Fast and easy API to I/O in different formats
- Working with missing records
- Merging / joining (`concat`, `join`)
- Grouping
- Reshaping
- Powerful time series manipulation

2 Data structures: import

```
import pandas as pd
import numpy as np
```

- The pandas import is always simple as this
- We import np to have access to all numpy methods

2 Data structures: Series

```
s = pd.Series([0.2, 0.4, 0.8, 1.6])

s.index
# returns: Int64Index([0, 1, 2, 3], dtype="int64")

s.values
# returns: array([ 0.2, 0.4, 0.8, 1.6 ])

s[0]  # to access data

s2 = pd.Series([0.2, 0.4, 0.8, 1.6], index=['a', 'c', 'b', 'd'])
```

- A Series is a **one-dimensional array-like** object containing an array of data
- We can access to values and indexes
- And so the data is indexed inside and can be accessible with: `s[0]`
- You can also create your own index => Series can be created from dictionaries

2 Data structures: Series

```
agedata = {"francois": 51, "angela": 51, "barack": 55}

s3 = pd.Series(agedata)  # create from dict

s3[s3 > 52]  # filtering

s3 * 2  # scalar multiplication
s3.mean()
s3.std() # standard deviation
s3.max()
s3.abs() # transform all values to absolute

np.exp(s3)  # exponential

"angela" in s3  # boolean to find if a key is in

s3[["angela", "barack"]]  # get several values
```

- Series can be created from dictionaries (keys will be sorted)
- On Series we can do **filtering**, **scalar multiplication** or math functions applying
- Determine if an index is in the series with **in**

2 Data structures: Series

```
agedata = {"francois": 51, "angela": 51, "barack": 55}
presidents = ["barack", "francois", "angela", "georges"]

s4 = pd.Series(agedata, index=presidents)

# same
pd.isnull(s4)
s4.isnull()

pd.notnull(s4)

s3 + s4
```

```
In [20]:
s4

Out[20]:
barack       55
francois     51
angela       51
georges     NaN
dtype: float64
```

```
In [25]:
s3 + s4

Out[25]:
angela      102
barack      110
francois    102
georges     NaN
dtype: float64
```

- NaN (Not a Number) is the null element in a Series object
- isnull and notnull returns if elements are nulls
- We can add 2 Series

2 Data structures: Series

```
s4.name = "presidents_ages"
s4.index.name = "name"

s4.index = ["Lula", "Cameron", "Renzi", "Putin"]
```

- An index can have a name
- Like a Series
- We can change the index afterwards

2 Data structures: DataFrame

```
data = {
    "city": ["Paris", "London", "Berlin"],
    "density": [3550, 5100, 3750],
    "area": [2723, 1623, 984],
    "population": [9645000, 8278000, 3675000],
}

df = pd.DataFrame(data)
```

In [55]:

```
df
```

Out[55]:

	area	city	density	population
0	2723	Paris	3550	9645000
1	1623	London	5100	8278000
2	984	Berlin	3750	3675000

- A dataframe is a tabular data structure, containing an ordered collection of columns,each of which can be a different value type (numeric, boolean, string, etc.)
- DataFrame has both rows and columns index

2 Data structures: DataFrame

```
columns = ["city", "area", "population", "density"]
df = pd.DataFrame(data, columns=columns)

df["area"]
df.area
# returns a Series object of the areas in the df

df.dtypes  # to get the types of columns

df.info()
df.describe()  # give stats on the df
df.values # numpy.ndarray
df.index

df = df.set_index("city")
```

- Columns can be specified
- We get a Series if we index a DataFrame

2 Data structures: DataFrame

```
df["population"] / df["area"]

df["real_density"] = df["population"] / df["area"]

df.ix["Paris"]  # to get the row with index "Paris"
df.ix[["London", "Berlin"]]

df[df["real_density"] < 5000]  # to filter by density

df.sort_index(by="real_density", ascending=True)  # to sort
```

- Operations between columns are possible
- We can add new columns easily
- If we set index with `ix[]` rows are accessible

2 Data structures: DataFrame

Selecting the data

```
df["area"]  # get the column
df.ix["Paris"]  # get the row

# multiple columns
df[["area", "population"]]

# loc examples
df.loc["Paris", "area"]  # will return the exact value
df.loc[df["density"] < 5000, ["population", "area"]]

# iloc example
df.iloc[1, 2]
```

- Be careful when getting column or row
- For advanced indexing we have:
    - loc: selecting by label
    - iloc: selecting by position

2 Data structures: DataFrame

Assigning the data

```
df.iloc[1, 2] = 10
df.iloc[1, :] = 10

df[df["density"] == 10] = 6000
```

- After selecting (with all different ways) the data we can assign them

2 Data structures: DataFrame

Creating and dropping columns

```
# creating and dropping columns

# create from python list or pandas series
df['new_column'] = [1,2,3]

# create as a transformation of other columns
df['density_diff'] = df['density'] - df["real_density"]

# drop columns and rows
df.drop(['density_diff'], axis=1, inplace=True)
df.drop(['London'], axis=0, inplace=True)
```

3 Interactions I/O

Read the data: text file

read_csv	load delimited data from a file, URL, of file-like object (comma as default delimiter)
read_table	load delimited data from a file, URL, of file-like object ('\t' as default delimiter)
read_fwf	Read data in fixed-width column
read_clipboard	Read data from the clipboard

```
df = pd.read_csv("population.csv")

# with parameters
df1 = pd.read_csv(filename,
    sep=",",
    header=None, # Row number to use as the column names
    names=[], # List of column names to use
    index_col=[], # Column to use as the row labels of the DataFrame
    na_values=[], # Additional strings to recognize as NA/NaN
)
```

- **read_csv** is the most useful function to read the data (because the data is often used as csv format)
- we can specified a lot of parameters to read_*

3 Interactions I/O

Read the data: database

```
import sqlite3

connexion = sqlite3.connect(':memory:')

df = pd.io.sql.read_frame("SELECT * FROM table", connexion)
```

- We can do the same with a MySQL python connector

3 Interactions I/O

Write the data

```
df.to_csv("output/population_out.csv", index=False)
```

- **to_csv** is like read_csv and has the same parameters

4 Data Manipulation

Sorting and manipulating index

```
df.sort_values(by="Land area", ascending=False)

data1 = df[["City / Urban area", "Country", "Population"]].set_index([
        "City / Urban area", "Country"])

# reset the index to be incremental
data1.reset_index(inplace=True)
```

4 Data Manipulation

Join

```
# joining df on index

left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                     index=['K0', 'K1', 'K2'])
right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                      'D': ['D0', 'D2', 'D3']},
                      index=['K0', 'K2', 'K3'])
left.join(right)
```

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

4 Data Manipulation

Join

```
# joining df on a column

left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3'],
                     'key': ['K0', 'K1', 'K0', 'K1']})


right = pd.DataFrame({'C': ['C0', 'C1'],
                      'D': ['D0', 'D1']},
                      index=['K0', 'K1'])


result = left.join(right, on='key', how='inner')
# default join is left
```

left

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K0
3	A3	B3	K1

right

	C	D
K0	C0	D0
K1	C1	D1

Result

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K0	C0	D0
3	A3	B3	K1	C1	D1

4 Data Manipulation

Merge

```
# joining df with multiple columns

left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})


right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                      'key2': ['K0', 'K0', 'K0', 'K0'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})


result = pd.merge(left, right, on=['key1', 'key2'])
```

left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

Result

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2

4 Data Manipulation

Concat

left

	A	B	key1	key2
0	A0	B0	K0	K0
1	A1	B1	K0	K1
2	A2	B2	K1	K0
3	A3	B3	K2	K1

right

	C	D	key1	key2
0	C0	D0	K0	K0
1	C1	D1	K1	K0
2	C2	D2	K1	K0
3	C3	D3	K2	K0

```
# concatenating df

df2 = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                    'key2': ['K0', 'K1', 'K0', 'K1'],
                    'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3']})


df1 = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                    'key2': ['K0', 'K0', 'K0', 'K0'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']})

new_df1 = pd.concat([df1, df2], axis=0)

new_df2 = pd.concat([df1, df2], axis=1)
```

new_df1

	A	B	C	D	key1	key2
0	NaN	NaN	C0	D0	K0	K0
1	NaN	NaN	C1	D1	K1	K0
2	NaN	NaN	C2	D2	K1	K0
3	NaN	NaN	C3	D3	K2	K0
0	A0	B0	NaN	NaN	K0	K0
1	A1	B1	NaN	NaN	K0	K1
2	A2	B2	NaN	NaN	K1	K0
3	A3	B3	NaN	NaN	K2	K1

new_df2

	C	D	key1	key2	A	B	key1	key2
0	C0	D0	K0	K0	A0	B0	K0	K0
1	C1	D1	K1	K0	A1	B1	K0	K1
2	C2	D2	K1	K0	A2	B2	K1	K0
3	C3	D3	K2	K0	A3	B3	K2	K1

4 Data Manipulation

Aggregations

```
# aggregate df
population = pd.read_csv('population.csv')
population.groupby('Country').mean()
```

	Population	Land area	Density
Country			
Argentina	11200000.0	2266.0	4950.000000
Australia	2724000.0	1790.0	1516.666667
Austria	1550000.0	453.0	3400.000000
Azerbaijan	2100000.0	544.0	3850.000000
Belgium	1570000.0	712.0	2200.000000

	City / Urban area	Population	Land area	Density
Country				
Argentina	Buenos Aires	11200000	2266	4950
Australia	Sydney	3502000	2080	2100
Austria	Vienna	1550000	453	3400
Azerbaijan	Baku/Sumqayit	2100000	544	3850
Belgium	Brussels	1570000	712	2200

```
population.groupby('Country').max()
```

	City / Urban area	Population	Land area	Density
Country				
Argentina	Buenos Aires	11200000	2266	4950
Australia	Sydney	3502000	1687	2100
Austria	Vienna	1550000	453	3400
Azerbaijan	Baku/Sumqayit	2100000	544	3850
Belgium	Brussels	1570000	712	2200

```
population.groupby('Country').first()
```

	City / Urban area	Population	Land area	Density
Country				
Argentina	1	1	1	1
Australia	3	3	3	3
Austria	1	1	1	1
Azerbaijan	1	1	1	1
Belgium	1	1	1	1

```
population.groupby('Country').count()
```

4 Data Manipulation

A simple plot

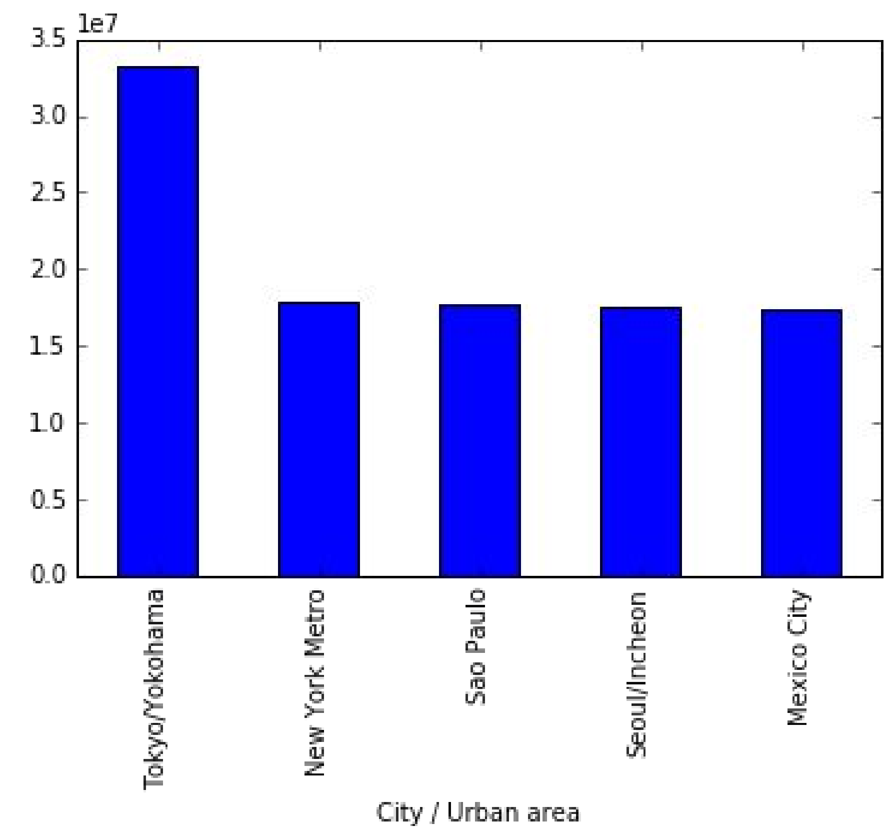
```
%matplotlib inline

# a quick viz ...

pop_series = population.set_index('City / Urban Area')['Population'].copy()
# we must use copy() otherwise the columns is a view (reference)

pop_series.sort(inplace=True, ascending=False)

pop_series.head().plot(kind='bar')
```



4 Data Manipulation

Apply

```
# simple operations are executed directly on columns:

# population in millions
population['Population'] / 1000000

# concatenation of country and city
'Country: ' + population['country'] + ', City:' + population['City / Urban area']

# but some operations need to be executed on elements:

# convert column values to lowercase
population['country'].lower() # doesn't work
population['country'].apply(lambda x: x.lower())

# include condition in the transformation: if population is under 3 million replace value with '<3M'
'<3M' if population['Population'] < 3000000 else pass # doesn't work
population['Population'].apply(lambda x: '<3M' if x < 3000000 else x)
```

4 Data Manipulation

Other useful operations

```
population = pd.read_csv('population.csv')


# return unique values in column
population.Country.unique()

# return counts of unique values in column
population.Country.value_counts()

population.rename(columns={'Country': 'country'}, inplace=True)

# return shape in format: (num rows, num cols)
population.shape

# transpose dataframe
population.T
```

5 Exercise 1

World population:

1. Calculate density_estimation from Population (hint: population / area)
2. Calculate **error** (substract density_estimation from real density)
3. Calculate **proportional_error** - absolute error divided by real density
4. Sort dataframe by proporional_error and find in which city the **estimation is the worst** (highest proportional error)

5 Exercise 2

Movie Ratings:

Download the file http://files.grouplens.org/datasets/movielens/ml-1m.zip and save it in the *data/* folder

1. Open **data/ml-1m.zip** and extract the data files users.dat, ratings.dat, movies.dat

2. **Load** files in Pandas (hint: you need to set the correct delimiter!).
   Use the column names:
   - users.dat: user id, gender, age, occupation code, zip
   - ratings.dat: user id, movie id, rating, timestamp
   - movies.dat: movie id, title, genre

3. **Join** the files to get a single table with all the data

4. The 5 movies with the most number of ratings

5. Create a list called **active_titles** that is made up of movies each having **at least 250 ratings**

6. For the subset of movies in the active_titles list compute the following:
   a. The 3 movies with the **highest average** rating for females. Do the same for males.
   b. The **10 movies men liked much more than women** and the 10 movies women liked more than men (use the difference in average ratings and sort ascending and descending).
   c. The 5 movies that had the **highest standard deviation** in rating.

Thanks a lot!

kkarp@equancy.com