Data Wrangling and Data Analysis

Data Extraction 2

Hakim Qahtan

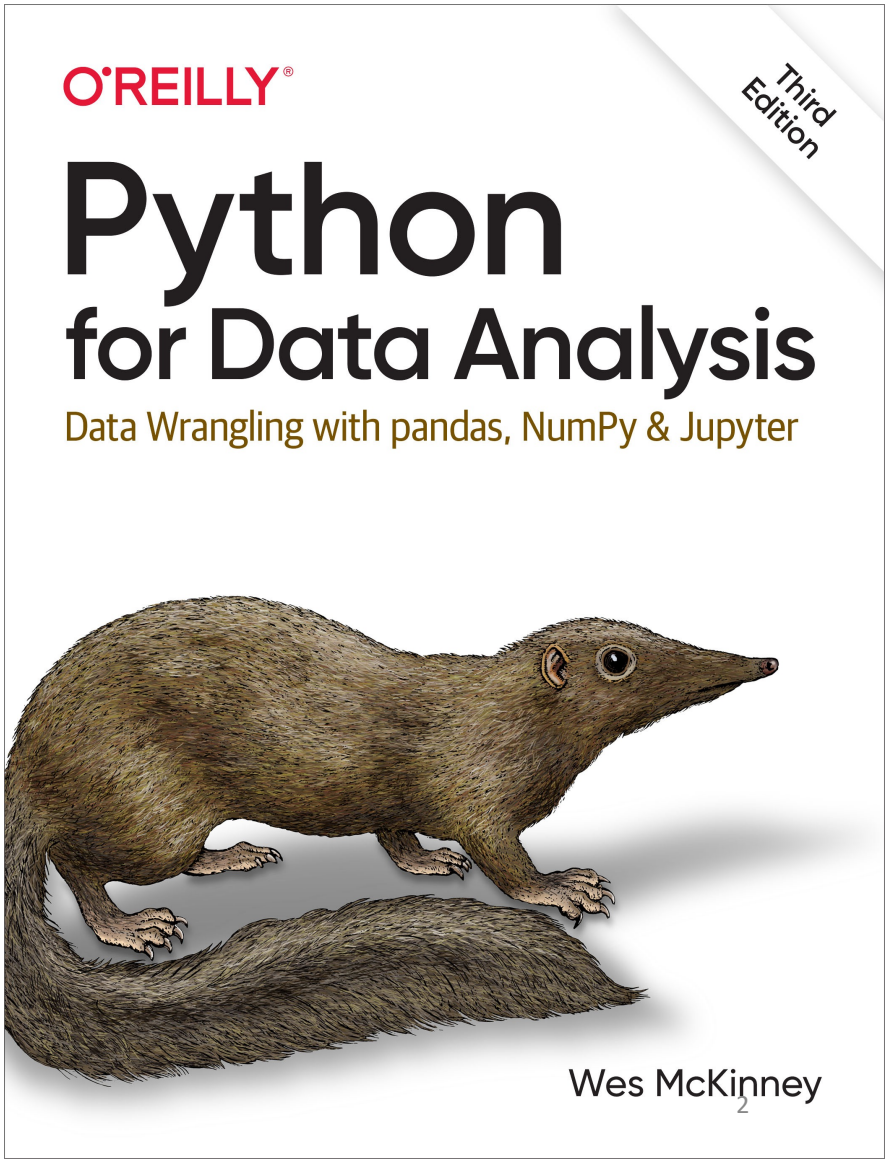
Department of Information and Computing Sciences
Utrecht University

Utrecht University

Reading Material for Today

Python for Data Analysis, 3E

CH 5, and 6.1.



Utrecht University

- **So Far**
  - Data models and focused on the relational model
  - Creating and querying databases
- **Today**
  - Querying Databases to obtain aggregated values
  - Data manipulation in Python
  - Dataframes
  - Connecting to databases
  - Data profiling

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department

```
SELECT AVG (salary)
FROM instructor
WHERE dept_name= 'Comp. Sci.';
```

- Find the total number of instructors who taught a course in the Spring 2010 semester

```
SELECT COUNT (DISTINCT ID)
FROM teaches
WHERE semester = 'Spring' AND year = 2010;
```

- Find the number of tuples in the *course* relation

```
SELECT COUNT (*)
FROM course;
```

Aggregate Functions (Cont.)

- Find the average salary of instructors in each department

```
SELECT dept_name, AVG (salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

ID	name	dept_name	salary
22322	Einstein	Physics	95000
33452	Gold	Physics	87000
21212	Wu	Finance	90000
10101	Brandt	Comp. Sci.	82000
43521	Katz	Comp. Sci.	75000
98531	Kim	Biology	78000
58763	Crick	Elec. Eng.	80000
52187	Mozart	History	65000
32343	El Said	History	86000

The query result

dept_name	avg_salary
Physics	91000
Finance	90000
Comp. Sci.	78500
Biology	78000
Elec. Eng.	80000
History	75500

Aggregate Functions (Cont.)

- Find the average salary of instructors in each department which has average salary greater than 80000 – use HAVING because WHERE cannot be used with aggregate functions

```
SELECT dept_name, AVG (salary) AS avg_salary
FROM instructor GROUP BY dept_name
HAVING avg_salary > 80000;
```

The query result

dept_name	avg_salary
Physics	91000
Finance	90000

ID	name	dept_name	salary
22322	Einstein	Physics	95000
33452	Gold	Physics	87000
21212	Wu	Finance	90000
10101	Brandt	Comp. Sci.	82000
43521	Katz	Comp. Sci.	75000
98531	Kim	Biology	78000
58763	Crick	Elec. Eng.	80000
52187	Mozart	History	65000
32343	El Said	History	86000

Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **SELECT-FROM-WHERE** expression that is nested within another query.
- The nesting can be done in the following SQL query

SELECT $A_1, A_2, \ldots, A_n$
FROM $r_1, r_2, \ldots, r_n$
WHERE *P*

as follows:

- A_i can be replaced be a subquery that generates a single value.
- r_i can be replaced by any valid subquery
- *P* can be replaced with an expression of the form:

  B <operation> (subquery)

Subqueries – Examples (Subquery in the WHERE clause)

- Find courses offered in Fall 2009 and in Spring 2010

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall' AND year = 2009 AND
  course_id IN (SELECT course_id
                FROM section
                WHERE semester = 'Spring' AND year= 2010);
```

Subqueries – Examples (Subquery in the FROM clause)

- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
SELECT dept_name, avg_salary
FROM (SELECT dept_name, AVG (salary) AS avg_salary
        FROM instructor
        GROUP BY dept_name)
WHERE avg_salary > 42000;
```

Subqueries – Examples (Subquery in the SELECT clause)

- List all departments along with the number of instructors in each department

```
SELECT dept_name,
        (SELECT COUNT(*)
           FROM instructor
          WHERE department.dept_name = instructor.dept_name)
        AS num_instructors
FROM department;
```

- Runtime error if subquery returns more than one result tuple
- Note that: subqueries are parenthesized SELECT-FROM-WHERE statements

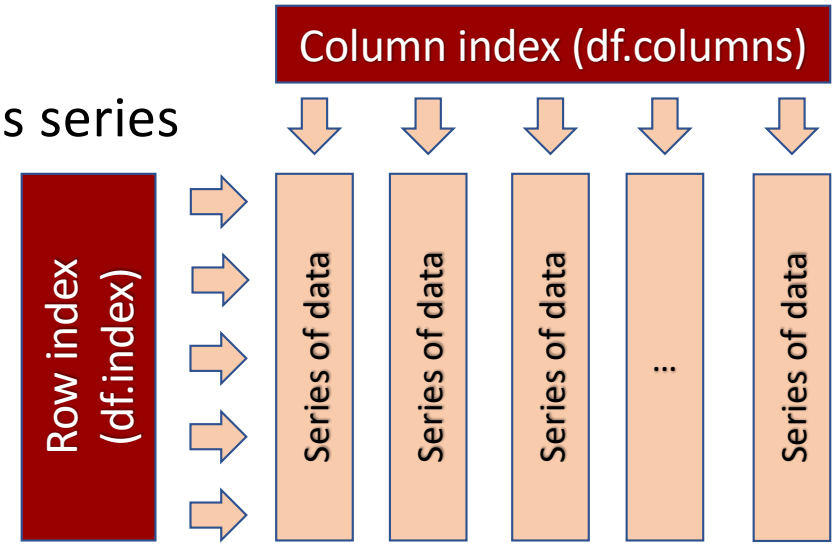
Data Extraction with Python

Aren't DBMSs Enough?

- DBMSs provides effective and efficient access for the data
- Performing data analysis is limited
- Programming languages (such as Python) provides libraries for wide range of data analysis techniques including different ML models

Pandas Dataframes

- The most popular way to handle data tables in Python is using Pandas dataframes
- DataFrame: a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.)
- Has columns and rows indexes
- Columns are made up of Pandas series



Creating DataFrame

In [1]:
```
import pandas as pd
data = {'State': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
 'Year': [2000, 2001, 2002, 2001, 2002, 2003],
 'Population': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
df = pd.DataFrame(data)
```

In [2]:
```
df
```

Out[2]:

	State	Year	Population
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

- Similarly: you can use the following code

```
import pandas as pd
data = [['Ohio', 2000, 1.5], ['Ohio', 2001, 1.7],
        ['Ohio', 2002, 3.6], ['Nevada', 2001, 2.4],
        ['Nevada', 2002, 2.9], ['Nevada', 2003, 3.2]]
cols = ['State', 'Year', 'Population']
df = pd.DataFrame(data, columns = cols)
```

Load DataFrame from CSV Files

- The simplest way is:

```
df = pd.read_csv('file.csv')    # often works
```

- More options can be added when loading a csv file into a dataframe

```
df = pd.read_csv('movies.csv', header=0,
    index_col=0, quotechar='"',sep=",",
    na_values = ['na', '-', '.', ‘’])
```

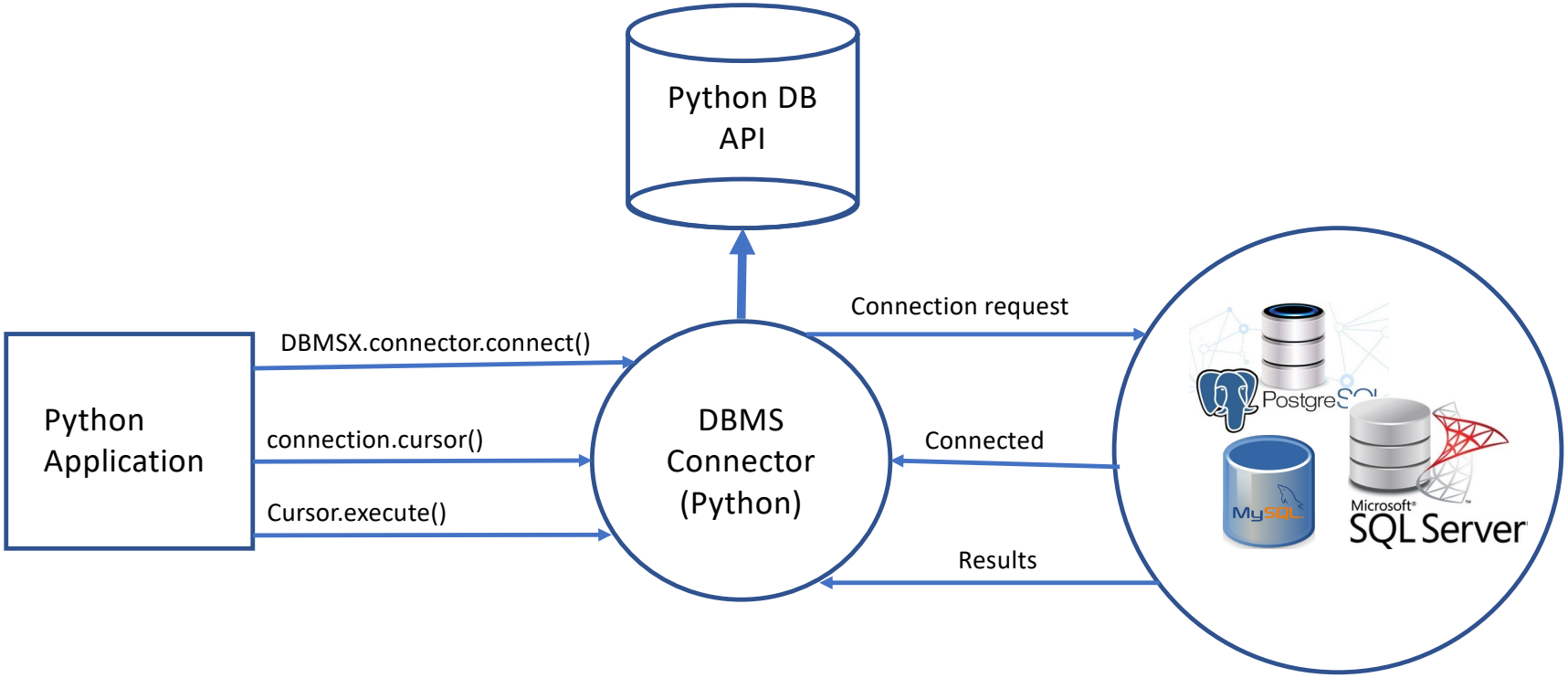
- More options can be found in Pandas documentation
- Remeber to import the Pandas library as pd

Load DataFrame from EXCEL Files

- Each Excel sheet in a Python dictionary

```
workbook = pd.ExcelFile('file.xlsx')
dictionary = {}
for sheet_name in workbook.sheet_names:
    df = workbook.parse(sheet_name)
    dictionary[sheet_name] = df
```

- The parse() method takes many arguments like read_csv().
- Refer to the Pandas documentation for more options


Python DB
API
Python
Application
DBMSX.connector.connect()
connection.cursor()
Cursor.execute()
DBMS
Connector
(Python)
Connection request
Connected
Results
PostgreSQL
MySQL
Microsoft®
SQL Server

Load DataFrame from MySQL Database

- Create Connector/Python to MySQL

```
from mysql.connector import connection
cnx = connection.MySQLConnection(user='user_name',
      password='password', host='127.0.0.1', database='university')
cursor = cnx.cursor()
```

- Create SQL query as a string and send the query to the DBMS

```
query = ("SELECT ID, name, tot_cred   FROM student WHERE tot_cred > 24")
cursor.execute(query)
```

- Process the results of the query and close the connection

```
df = pd.DataFrame(cursor, columns = ["ID", "name", "tot_cred"])
cursor.close()
cnx.close()
```

Load DataFrame from PostgreSQL Database

- Create Connector/Python to PostgreSQL using psycopg2 library

```
import psycopg2
conn = psycopg2.connect(database='db_name', user='user_name',
                        password='password', host='127.0.0.1', port=5432)
cursor = conn.cursor()
```

- Create SQL query as a string and send the query to the DBMS

```
query = ("SELECT ID, name, tot_cred   FROM student WHERE tot_cred > 24")
cursor.execute(query)
```

- Process the results of the query and close the connection

```
df = pd.DataFrame(cursor, columns = ["ID", "name", "tot_cred"])
cursor.close()
cnx.close()
```

Load DataFrame from SQLite Database

- Create Connector/Python to SQLite using sqlite3 library

```
import sqlite3
conn = sqlite3.connect(db_file)
cursor = conn.cursor()
```

- Create SQL query as a string and send the query to the DBMS
- Process the results of the query and close the connection
- Example

```
import sqlite3
conn = sqlite3.connect(``chinook.db”)
cursor = conn.cursor()
table = ``albums”
query = ``SELECT * FROM” + table
cur.execute(query)
rows = cur.fetchall()
   for row in rows:
      print(row)
```

Working with Dataframes

- Consider the movies dataset extracted from imdb dataset
- Start by reading the csv file

```
df = pd.read_csv(filepath_or_buffer = 'movies.csv', delimiter=',',
      doublequote=True, quotechar='"',na_values = ['na', '-', '.', ''],
      quoting=csv.QUOTE_ALL, encoding = "ISO-8859-1")
```

- Extract sub-table of the dataframe

```
df.info()                    # index & data types
n = 4
dfh = df.head(n)             # get first n rows
dft = df.tail(n)             # get last n rows
dfs = df.describe()          # summary stats cols
top_left_corner_df = df.iloc[:5, :5]
```

Extracting Data from Dataframes

- Extarct row number 0

```
row1 = df.iloc[0,:]     #You may ignore adding the :
row1 = df.iloc[0]
```

- Extract the column with the names of directors

```
df.director_name     #  OR
df["director_name"]
```

Extracting Data from Dataframes (Cont.)

- Extract set of rows (corresponds to selection in relational algebra)

```
Rows_set1 = df.iloc[[5:10], ]          # Extracts rows 5,6,7,8, and 9
Rows_set2 = df.iloc[[5,6,8,10], ]     # Extracts rows 5,6,8, and 10
```

- Extract set of columns (corresponds to projection in relational algebra)

```
cols_set1 = df[df.columns[5:10]][:]          # Extracts columns 5,6,7,8, and 9
cols_set2 = df[df.columns[[5,7,9]]][:]     # Extracts rows 5,7, and 9
col_set3 = df[['actor_3_facebook_likes', 'actor_1_facebook_likes', 'content_rating']]
```

- Note that: df.columns is a vector that contains the attributes' names

Extracting Data from Dataframes (Cont.)

- Extract set of rows with a condition

```
df.loc[df['content_rating'] == 'PG-13', ['actor_1_facebook_likes',
        'actor_3_facebook_likes', 'budget']]
```

- You can do the same thing using iloc

```
df.iloc[(df['content_rating'] == 'PG-13').values, [1, 3]]
```

- Note that: iloc requires numerical values for the indexes

Profiling the Dataframes

- Display number of columns

```
print(len(df.columns))
```

- Display number of rows

```
print(len(df))      # OR print(len(df[df.columns[0]])
```

- Find the number of non-null values in each column (attribute)

```
df.count()
```

Profiling the Dataframes

- Display number of distinct values in an attribute

```
for col in  df.columns:
    print(col, ' has (', len(df[col].unique()), ') unique values')
```

- Display the data type of each attribute

```
dataTypeSeries = df.dtypes
for col_idx in range(len(df.columns)):
            print(df.columns[col_idx], 'has type (', dataTypeSeries[col_idx], ')')
```

Profiling the Dataframes – Aggregate Queries

- Find max, min, and average of numerical attributes

```
dataTypeSeries = df.dtypes
for col_idx in range(len(df.columns)):
      if (not (dataTypeSeries[col_idx] == 'object')):
          print(df.columns[col_idx], 'has Min = ', df[df.columns[col_idx]].min(),
              'Max = ', df[df.columns[col_idx]].max(),
              'Average = ', df[df.columns[col_idx]].mean())
```

Set Operations on Dataframes

- Assume the following dataframes

```
dd1 = pd.DataFrame( { 'id': ['1', '2', '3', '4', '5'], 'Feature1': ['A', 'C', 'E', 'G', 'I'],
        'Feature2': ['B', 'D', 'F', 'H', 'J']})
```

```
dd2 = pd.DataFrame( { 'id': ['1', '2', '6', '7', '8'], 'Feature1': ['A', 'C', 'O', 'Q', 'S'],
        'Feature2': ['B', 'D', 'P', 'R', 'T']})
```

- The *concat* function concatenates the dataframes allowing repetition

```
union_df = pd.concat([dd1, dd2])                # concatenate row-wise (default)
union_df = pd.concat([dd1, dd2], axis = 1)      # concatenate column-wise
```

Join Operation on Dataframes

- The *merge* function joins dataframes on selected attribute

```
df_merge_col = pd.merge(dd1, dd2, on='id')
```

- If the joining attribute has different names in both dataframes

```
df_merge_col = pd.merge(dd1, dd2, left_on='att_dd1', right_on = 'att_dd2')
```

- Summarize what you learned today in 2-minutes

The information in this presentation has been compiled with the utmost care,
but no rights can be derived from its contents.