Data Analysis - Input-Output with Pandas

Gabriele Pompa
gabriele.pompa@unisi.com

April 6, 2020

Contents

Resources:

- *Python for Finance (2nd ed.)*: Sec. 9.Basic I/O with Python, 9.I/O with Pandas.
- *The Python Tutorial*: Sec. 7.1.2 (the String `.format()` method) and 7.2 (reading and writing files)
- From *Pandas - Getting started tutorials:* How do I read and write tabular data?
- **Software**: DB Browser for SQLite. Download for Windows 64-bit | Mac OS | Linux (check distribution on download page)

Executive Summary

In this Notebook we will cover topics about serialization of Python objects, as well standard input output operations between Pandas and standard output formats.

The following sections are organized as follows:

- in Sec. 1 we introduce the os Python module which provides a useful interface between Python code and the underlying Operating System;
- in Sec. 2 we cover two standard serializations protocols available in Python: JSON and Pickle;
- in Sec. 3 we describe how to perform Input/Output operations between Pandas and SQL, CSV and Excel formats.

These are the basic imports that we need to work with NumPy, Pandas and to plot data using Matplotlib functionalities

[1]:

```
# for NumPy arrays
import numpy as np

# for Pandas Series and DataFrame
import pandas as pd

# for Matplotlib plotting
import matplotlib.pyplot as plt

# to do inline plots in the Notebook
%matplotlib inline
```

1 Introduction

Before talking about specific Input/Output (IO) protocols, it is important to mention that typical operating system functionalities (like creating and deleting files, folders, etc) are accessible from Python code using os module. This is a module we will include in our basic imports sectsion hereafter.

[2]:

```
import os
```

we use `os.makedirs()` function to create the `Data` folder, under our `IT_For_Business_And_Finance_2019_20` class folder, where we will put all our data files. Function `os.path.exists()` returns `True` if the folder (or file) path it receives in input already exists, otherwise `False`.

[3]:

```
dataFolderPath = "../Data"

if not os.path.exists(dataFolderPath):
    os.makedirs(dataFolderPath)
```

Notice the use of `..` syntax. The double dots `..` in file path Strings refers to *one directory above* in the directory tree. Therefore, since the notebook you are reading is located in the `IT_For_Business_And_Finance_2019_20/Notebooks` folder, `../Data` points (and is thus equivalent) to `IT_For_Business_And_Finance_2019_20/Data`.

2 Serialization Protocols in Python

When it comes to IO operations, Python is very flexible and offers several options. We'll review here two typical ways to transfer Python objects across machines: - JSON module, which implements human-readable encoding and decoding of basic Python object hierarchies. It is mostly suitable for Python Lists and Dicts. - Pickle module, which implements binary protocols for serializing and de-serializing a Python object structure. It convers a broad spectrum of Python data-structures.

2.1 JSON format: `json` module

JSON is the acronym for JavaScript Object Notation. It is a popular data interchange format.

The `json` Python module can take Python hierarchies (like nested Lists with Dicts inside etc.), *serialize* them as `.json` files (that is, convert to String representations) and then *deserialize* them (that is, reconstruct back the original Python object).

Pros: - JSON format is the standard to send data over a network connection. - `.json` files are, in general, human-readable.

Cons: - not all Python objects are serializable using `json` (e.g. NumPy arrays cannot be serialized in this way).

Let's make an example. We want to save the `refData` Dict of Python Lists

[4]:
```
refData = {
    'S&P Rating': ['A', 'BB', 'AA', 'CCC'],
    'Spread': [100, 300, 70, 700],
    'Country': ['USA', 'ITA', 'UK', 'ITA']
}

refData
```

[4]:
```
{'S&P Rating': ['A', 'BB', 'AA', 'CCC'],
 'Spread': [100, 300, 70, 700],
 'Country': ['USA', 'ITA', 'UK', 'ITA']}
```

First-of all we import the `json` module

[5]:
```
import json
```

We define the complete file path using the `os.path.join()` function, which concatenates the `dataFolderPath` to `Data` folder, together with `"refData.json"`, which is going to be the name of the `.json` file containing the serialized `refData` object.

[6]:
```
filePath = os.path.join(dataFolderPath, "refData.json")
```

To create and open a new file `filePath`, we use `open(filename, mode)` function, giving it the complete path `filePath` to the file to open and mode `'w'` to open it in write-mode. Function `open()` returns a file-object (which mediates the between IO operations and the underlying resource). We capture it in the `file` variable.

[7]:

```
with open(filePath, 'w') as file:
    %time json.dump(refData, file, indent="\t")
```

```
Wall time: 0 ns
```

Here and alsewhere we use the syntax

`%time statement`

to execute a statement and measure its execution time (Wall time) with the `%time` magic function.

Function

‘python `json.dump(obj, file_object[, indent])`

takes the `refData` object and serializes it as a text file, using the `file_object` file object. The optional argument `indent` is used to pretty-print nested levels of the `refData` object. Here we have used the `"\t"` character so that nested levels are distantiated of one tab. Take a look at `refData.json` file in `Data` folder… you can actually read it!

Notice the use of the `with` statement which: - manages the opening of the file `filePath`, calling `open()` function, - assign the file-object to the `file` variable, through the `as` keyword, - manages the closing of the file after the end of the indented block

Now that we have serialized the `refData` object as the `refData.json` file, we can assess whether the file-object is effectively now closed using the `.closed` attribute of the `file` file-object

[8]:

```
file.closed
```

[8]:

```
True
```

Let’s now reload the serialized object and retrieve the original `refData` object. Same opening through `open()`, but now in reading-mode, using mode `'r'`

[9]:

```
with open(filePath, 'r') as file:
    %time refData_reloaded = json.load(file)
```

```
Wall time: 0 ns
```

The deserialization (from text file to Python object) is managed by function

`json.load(file_object)`

which loads the contents of the file referred by `file_object` and convert them into a Python object

[10]:

```
refData_reloaded
```

[10]:

```
{'S&P Rating': ['A', 'BB', 'AA', 'CCC'],
 'Spread': [100, 300, 70, 700],
 'Country': ['USA', 'ITA', 'UK', 'ITA']}
```

Now that we have finished our IO operation, we can delete our `refData.json` file. We define a utility function to do this.

[11]:

```
def removeFile(fileName):
    """
   removeFile(fileName) function remove file 'fileName', if it exists. It also␣
 ↪prints on screen a success/failure message.

   Parameters:
       fileName (str): name of the file ('Data' folder is assumed)

   Returns:
       None
   """

    if os.path.isfile(os.path.join(dataFolderPath, fileName)):
        os.remove(os.path.join(dataFolderPath, fileName))

        # double-check if file still exists
        fileStillExists = os.path.isfile(os.path.join(dataFolderPath, fileName))

        if fileStillExists:
            print("Failure: file {}still exists...".format(fileName))
        else:
            print("Success: file {}successfully removed!".format(fileName))

    else:
        print("File {}already removed.".format(fileName))
```

Notice the use of `os`'s functions: - `os.path.isfile()` which returns `True` if the file in input exists and `False`, otherwise; - `os.remove()` which removes the file in input.

[12]:

```
removeFile(filePath)
```

```
Success: file ../Data\refData.json successfully removed!
```

Take a look in `Data` folder to see that effectively `refData.json` file is not there anymore…

Unfortunately, not all object that you work with in Python are serializable (and thus, transferrable) using the JSON format. A counter-example? NumPy arrays…

[13]:

```
# unserializableFilePath = os.path.join(dataFolderPath, "dummyArray.json")
#
# with open(unserializableFilePath, 'w') as file:
#
#    # raises a TypeError: Object of type ndarray is not JSON serializable
#    %time json.dump(np.array([1,2,3]), file)
```

2.2 `pickle` module

Contrary to JSON, pickle is a protocol which allows the serialization of arbitrarily complex Python objects. In Python, it is implemented in the `pickle` module.

Pros: - Pickle works with arbitrary Python obkects (NumPy array and Pandas Series/DataFrames too).

Cons: - Pickle format is not cross-platform. That is, a file serialized on a Mac OS might be impossible to de-serialize on a Windows machine (and viceversa). - `.pkl` files are not human-readable.

In real life, especially if you have to pass data across different machines, don't use Pickle.

Let's make an example. We want to save the `mat` NumPy array

[14]:
```
rows = int(1e6)
```

[15]:
```
mat = np.array([[i*k for i in range(1,rows+1)] for k in range(1,6)]).T
```

[16]:
```
mat
```

[16]:
```
array([[      1,       2,       3,       4,       5],
       [      2,       4,       6,       8,      10],
       [      3,       6,       9,      12,      15],
       …,
       [ 999998, 1999996, 2999994, 3999992, 4999990],
       [ 999999, 1999998, 2999997, 3999996, 4999995],
       [1000000, 2000000, 3000000, 4000000, 5000000]])
```

[17]:
```
mat.shape
```

[17]:
```
(1000000, 5)
```

[18]:
```
mat.dtype
```

[18]:
```
dtype('int32')
```

First-of all we import the `pickle` module

[19]:
```
import pickle
```

[20]:
```
filePath = os.path.join(dataFolderPath, "mat.pkl")

with open(filePath, 'wb') as file:
    %time pickle.dump(mat, file)
```

```
Wall time: 35.9 ms
```

Notice the use of `'wb'` mode when opening the file to store `mat` array. It's going to be a binary file.

Function

‘`python pickle.dump(obj, file_object)`

takes the `mat` object and serializes it as a binary file, using the `file_object` file object.

```
[21]: file.closed
```

```
[21]: True
```

Let's now reload it, using the `'rb'` mode to read the binary file `"mat.pkl"`

```
[22]: with open(filePath, 'rb') as file:
          %time mat_reloaded = pickle.load(file)
```

```
Wall time: 26.9 ms
```

```
[23]: file.closed
```

```
[23]: True
```

```
[24]: mat_reloaded
```

```
[24]: array([[      1,       2,       3,       4,       5],
              [      2,       4,       6,       8,      10],
              [      3,       6,       9,      12,      15],
              …,
              [ 999998, 1999996, 2999994, 3999992, 4999990],
              [ 999999, 1999998, 2999997, 3999996, 4999995],
              [1000000, 2000000, 3000000, 4000000, 5000000]])
```

Let's clean-up Data folder…

```
[25]: removeFile(filePath)
```

```
Success: file ../Data\mat.pkl successfully removed!
```

In case you have several object that you want to keep together in a unique file, wrap them in a Python Dict

```
[26]: mat_dict = {'mat': mat,
                  'mat_squared': mat**2}
```

```
[27]: mat_dict['mat']
```

```
[27]: array([[      1,       2,       3,       4,       5],
              [      2,       4,       6,       8,      10],
              [      3,       6,       9,      12,      15],
              …,
              [ 999998, 1999996, 2999994, 3999992, 4999990],
              [ 999999, 1999998, 2999997, 3999996, 4999995],
              [1000000, 2000000, 3000000, 4000000, 5000000]])
```

```
[28]: mat_dict['mat_squared']
```

```
[28]: array([[          1,           4,           9,          16,          25],
             [          4,          16,          36,          64,         100],
             [          9,          36,          81,         144,         225],
             …,
             [ -731379964,  1369447440,  2007514916,  1182822464, -1104629916],
             [ -729379967,  1377447428,  2025514889,  1214822416, -1054629991],
             [ -727379968,  1385447424,  2043514880,  1246822400, -1004630016]],
            dtype=int32)
```

```
[29]: filePath = os.path.join(dataFolderPath, "mat_dict.pkl")
```

```
[30]: with open(filePath, 'wb') as file:
          %time pickle.dump(mat_dict, file)
```

```
Wall time: 64.8 ms
```

```
[31]: with open(filePath, 'rb') as file:
          %time mat_dict_reloaded = pickle.load(file)
```

```
Wall time: 54.9 ms
```

```
[32]: mat_dict_reloaded['mat']
```

```
[32]: array([[      1,       2,       3,       4,       5],
             [      2,       4,       6,       8,      10],
             [      3,       6,       9,      12,      15],
             …,
             [ 999998, 1999996, 2999994, 3999992, 4999990],
             [ 999999, 1999998, 2999997, 3999996, 4999995],
             [1000000, 2000000, 3000000, 4000000, 5000000]])
```

```
[33]: mat_dict_reloaded['mat_squared']
```

```
[33]: array([[          1,           4,           9,          16,          25],
             [          4,          16,          36,          64,         100],
             [          9,          36,          81,         144,         225],
             …,
             [ -731379964,  1369447440,  2007514916,  1182822464, -1104629916],
             [ -729379967,  1377447428,  2025514889,  1214822416, -1054629991],
             [ -727379968,  1385447424,  2043514880,  1246822400, -1004630016]])
```

```
[34]: removeFile(filePath)
```

```
Success: file ../Data\mat_dict.pkl successfully removed!
```

3 IO with Pandas

Pandas supports IO operations from/to many file formats. As a rule of thumb:

- to import data into Pandas, you can use `read_*` functions (like `pd.read_sql()`, `pd.read_csv()`, `pd.read_excel()`, etc.);
- to export data from Pandas, you can use `to_*` methods of Pandas DataFrames (like for a `df` DataFrame, `df.to_sql()`, `df.to_csv()`, `df.to_excel()`, etc.);

Here we'll review some of the most common file formats:

- SQL, using the `sqlite3` module;
- CSV
- Excel

3.1 SQL

In a nutshell, SQL stands for Structured Query Language and it is a domain-specific language to manage data held in Relational Databases.

We'll first follow a step-by-step approach, typing real SQL queries and executing them using the `sqlite3` module. Then we'll explain the real-life approach using `df.to_sql()` method and `pd.read_sql()` function.

3.1.1 SQL queries from Python: `sqlite3` module

SQLite is a library (written in C programming language) that implements a SQL database engine. The built-in module `sqlite3` implements the interface between Python and SQLite, such that you can define SQL tables using Python code and store Pandas DataFrames into them.

Let's suppose we want to store in a SQL table our reference data DataFrame `df_refData`

[35]:
```
df_refData = pd.DataFrame(data={
                              'S&P Rating': ['A', 'BB', 'AA', 'CCC'],
                              'Spread': [100, 300, 70, 700],
                              'Country': ['USA', 'ITA', 'UK', 'ITA'],
                              'Market Cap': [430.0, 45.0, 161.25, 5.00]
                               },
                          index=['Firm_1', 'Firm_2', 'Firm_3', 'Firm_4'])

df_refData
```

[35]:
```
       S&P Rating  Spread Country  Market Cap
Firm_1          A     100     USA      430.00
Firm_2         BB     300     ITA       45.00
Firm_3         AA      70      UK      161.25
Firm_4        CCC     700     ITA        5.00
```

First of all we import `sqlite3`, giving it the alias name `sq3`

[36]:
```
import sqlite3 as sq3
```

Next we have to open a *connection* to the SQL engine, which creates an empty `"refData.db"` file in our `Data` folder

[37]:

```
filePath = os.path.join(dataFolderPath, "refData.db")
```

[38]:

```
con = sq3.connect(filePath)
```

The `sq3.connect()` returns a connector `con` that manages the interaction between Python code and SQLite engine

[39]:

```
type(con)
```

[39]:

```
sqlite3.Connection
```

Let's now write the SQL query to create the table `refData`, as a Python String named `query`. For details on SQL syntax, there is a good SQLite tutorial. Knowledge of SQL is not required to pass this class, but understanding at least basic syntax might be very usefull in your daily working life.

[40]:

```
query = """CREATE TABLE refData (
               Firms TEXT NOT NULL,
               SnP_Rating TEXT,
               Spread INT,
               Country TEXT,
               Market_Cap REAL
)"""

print(query)
```

```
CREATE TABLE refData (
                Firms TEXT NOT NULL,
                SnP_Rating TEXT,
                Spread INT,
                Country TEXT,
                Market_Cap REAL
)
```

So, the meaning of the SQL query which uses the `CREATE TABLE` statement and typed as the Python String `query` is that of just creating an empty table `refData` of five columns: - `Firms`: a column of text strings (`TEXT`) constrained to store non-missing values (`NOT NULL`). We will store here the index of `df_refData`; - `SnP_Rating`: column of text strings (`TEXT`) that will store the `'S&P Rating'` column; - `Spread`: column of integer numbers (`INT`) that will store the `'Spread'` column; - `Country`: column of text strings (`TEXT`) that will store the `'Country'` column; - `Market_Cap`: column of float numbers (`REAL`) that will store the `'Market Cap'` column;

For those who are not familiar with SQL data-types, notice that `TEXT`, `INT` and `REAL` are the SQLite analogous of Python `str`, `int` and `float` data-types, respectively. For details, see section SQLite Data Types

We now execute the query using the `.execute()` method of the connector

[41]:

```
con.execute(query)
```

```
[41]: <sqlite3.Cursor at 0x18eb1671960>
```

Using the `.commit()` method, we actually implement the changes due to the run of the `query` to the `"refData.db"` file.

TAKE-HOME MESSAGE: always `.commit()` after `.execute()`, otherwise no changes will be made to the table.

```
[42]: con.commit()
```

You can actually open the `"refData.db"` using DB Browser for SQLite and there, under 'Browse Data' tab, you'll see an empty table `refData` of five columns and SQLite data-types as above.

We can store `df_refData` into `refData` table row-by-row, using the `.iterrows()` method of a Pandas DataFrame, which in a for loop returns the index and a Pandas Series of the given row. Check it out

```
[43]: for index, row in df_refData.iterrows():
          print("Index: \n{}\n\nRow: \n{}\n\n".format(index, row))
```

```
Index:
Firm_1

Row:
S&P Rating      A
Spread        100
Country       USA
Market Cap    430
Name: Firm_1, dtype: object


Index:
Firm_2

Row:
S&P Rating     BB
Spread        300
Country       ITA
Market Cap     45
Name: Firm_2, dtype: object


Index:
Firm_3

Row:
S&P Rating        AA
Spread            70
Country           UK
```

```
Market Cap    161.25
Name: Firm_3, dtype: object


Index:
Firm_4

Row:
S&P Rating    CCC
Spread        700
Country       ITA
Market Cap      5
Name: Firm_4, dtype: object
```

Looping over `df_refData`'s rows, let's now store each row as a row in `refData` table

[44]:
```
for index, row in df_refData.iterrows():
    query = "INSERT INTO refData VALUES ('{}', '{}', {}, '{}', {})".\
    format(index, row['S&P Rating'], row['Spread'], row["Country"], row["Market␣
 ↪Cap"])

    print(query)
    con.execute(query)

con.commit()
```

```
INSERT INTO refData VALUES ('Firm_1', 'A', 100, 'USA', 430.0)
INSERT INTO refData VALUES ('Firm_2', 'BB', 300, 'ITA', 45.0)
INSERT INTO refData VALUES ('Firm_3', 'AA', 70, 'UK', 161.25)
INSERT INTO refData VALUES ('Firm_4', 'CCC', 700, 'ITA', 5.0)
```

The `INSERT INTO refData VALUES...` query, using the `INSERT INTO` statement, is responsible to store the values of each row. We `.execute()` one query per row, replacing the `{}` brackets in the string declaration with the appropriate values of each column of the given `row`, thanks to the `.format()` String method. Notice the use of the `''` to write in SQL TEXT values, like `'Firm_1'`.

When all the `df_refData.iterrows()` loop is over, we then `.commit()` the changes all together. Check it out with DB Browser. Of course we could have `.commit()` after the `.execute()` of every row addition, but this would have been inefficient (and I discourage you to do that), since the `.commit()` is, in general, an time-consuming operation.

Now that we have stored values into the `refData` table, we can retrieve them using the standard `SELECT *` query. Rember always to `.commit()` the changes after you have `.execute()` your query, otherwise the selection of data won't be effective.

[45]:
```
query = "SELECT * FROM refData"
cursor = con.execute(query)
con.commit()
```

```
cursor
```

```
[45]: <sqlite3.Cursor at 0x18eb633a2d0>
```

The `SELECT * FROM refData` query is an SQL query to using the SELECT statement to query all (*) the contents `FROM` the `refData` table. We capture the output of `.execute()` selection as a `cursor`, which we can use to then fetch the selected data, using the `.fetchall()` method of the `cursor`

```
[46]: data=cursor.fetchall()
      data
```

```
[46]: [('Firm_1', 'A', 100, 'USA', 430.0),
       ('Firm_2', 'BB', 300, 'ITA', 45.0),
       ('Firm_3', 'AA', 70, 'UK', 161.25),
       ('Firm_4', 'CCC', 700, 'ITA', 5.0)]
```

```
[47]: type(data)
```

```
[47]: list
```

As expected, the values are there, but the `data` returned are not the original `df_refData` Pandas DataFrame, but in the form of a Python List. With some List comprehension effort, we can reconstruct our original DataFrame… but, as you could imagine, there is a much more efficient method to do all this IO operation

```
[48]: df_refData_reloaded = pd.DataFrame(data=[t[1:] for t in data],
                                         index=[t[0] for t in data],
                                         columns=['S&P Rating', 'Spread', 'Country',␣
        ↪'Market Cap'])

      df_refData_reloaded
```

```
[48]:        S&P Rating  Spread Country  Market Cap
      Firm_1          A     100     USA      430.00
      Firm_2         BB     300     ITA       45.00
      Firm_3         AA      70      UK      161.25
      Firm_4        CCC     700     ITA        5.00
```

Of course you can make conditional selections using the `WHERE` SQL statement. Here, we select rows corresponding to Firms having a `Market_Cap` above 100

```
[49]: query = "SELECT * FROM refData WHERE Market_Cap > 100"
      cursor = con.execute(query)
      con.commit()
      blueChips_data=cursor.fetchall()
      blueChips_data
```

[49]:
```
[('Firm_1', 'A', 100, 'USA', 430.0), ('Firm_3', 'AA', 70, 'UK', 161.25)]
```

In the next section we'll see that Pandas provides much more efficient methods to save and retrieve data with and SQL engine.

When you stop working with a SQL engine, close the connection

[50]:
```
con.close()
```

We clean-up our `Data` folder deleting the `"refData.db"` file (if still open, shut down DB Browser otherwise you won't be able to remove the file)

[51]:
```
removeFile(filePath)
```

```
Success: file ../Data\refData.db successfully removed!
```

3.1.2 Pandas and SQL: `.to_sql()` and `pd.read_sql()`

Thanks to the `.to_sql()` DataFrame's method and the function `.read_sql()`, Pandas allows us to replace all the

- `refData` table creation (`.execute()` of `CREATE TABLE refData...` query);
- the (highly inefficient) row-by-row data insertion (`.execute()` of `INSERT INTO refData VALUES...` query);
- all the `.commit()` to make the changes to `refData` table effective

with a call to `df_refData.to_sql()` method, and

- the selection of the data (`.execute()` of `SELECT * FROM refData` and `.fetchall()` selected data);
- as well conditional selection of the data (`.execute()` of `SELECT * FROM refData WHERE Market_Cap > 100` and `.fetchall()` selected data)
- all the `.commit()` to make the selection query effective

with a call to `pd.read_sql()` function.

All we have to do, at the very beginning, is opening a database connection. Let's quicly do it as we have already seen

[52]:
```
import sqlite3 as sq3

filePath = os.path.join(dataFolderPath, "refData.db")
con = sq3.connect(filePath)
type(con)
```

[52]:
```
sqlite3.Connection
```

Let's store the `df_refData` DataFrame into the newly created SQL table `refData` (physically saved into the `"refData.db"` file. You can inspected the newly created table opening the `"refData.db"` with DB Browser.

To do this we use the `.to_sql()` method whose essential syntax is

```
DataFrame.to_sql(name, con[, index_label])
```

This method saves the content of the DataFrame into a table using parameter: - name is going to be the name of the table; - con has to be an already opened database connection;

Moreover, DataFrame's index is going to be stored as an additional column into the db (the left-most column) and the parameter

- index_label is an optional parameter which specifies the label of the index column

```
[53]: df_refData.to_sql(name="refData", con=con, index_label="Firms")
```

```
C:\Users\gabri\Anaconda3\envs\ITForBusAndFin2020_env\lib\site-
packages\pandas\core\generic.py:2712: UserWarning: The spaces in these column
names will not be changed. In pandas versions < 0.14, spaces were converted to
underscores.
  method=method,
```

Let's now retrieve all the data stored in the refData table as the df_refData_reloaded DataFrame, with the idea of reconstructing a reloaded version of the original df_refData DataFrame

To do this we use the pd.read_sql() function whose essential syntax is

```
pd.read_sql(sql, con[, index_col])
```

which saves the content of the DataFrame into a table using parameter: - sql is a SQL query to be executed; - con has to be an already opened database connection;

Moreover, we can specify the column of the db to be interpreted as index in the reconstructed DataFrame and

- index_col is a parameter which specifies the name of the column to be used as index of the reconstructed DataFrame

```
[54]: query = "SELECT * FROM refData"

      df_refData_reloaded = pd.read_sql(sql=query, con=con, index_col="Firms")

      df_refData_reloaded
```

```
[54]:        S&P Rating  Spread Country  Market Cap
      Firms
      Firm_1          A     100     USA      430.00
      Firm_2         BB     300     ITA       45.00
      Firm_3         AA      70      UK      161.25
      Firm_4        CCC     700     ITA        5.00
```

To make conditional selections we can either specify a SQL query using the WHERE statement (notice the use of backticks `` to select a column with and empty space in the column name)

```
[55]: query = "SELECT * FROM refData WHERE `Market Cap` > 100"

      %time df_refData_blueChips = pd.read_sql(sql=query, con=con, index_col="Firms")
```

```

df_refData_blueChips
```

```
Wall time: 4.99 ms
```

[55]:

```
       S&P Rating  Spread Country  Market Cap
Firms
Firm_1          A     100     USA      430.00
Firm_3         AA      70      UK      161.25
```

or you can read in-memory the whole table as a reconstructed DataFrame and then using Pandas conditional row selection to retrieve your data

[56]:

```
%%time

query = "SELECT * FROM refData"

df_refData_reloaded = pd.read_sql(sql=query, con=con, index_col="Firms")

df_refData_blueChips = df_refData_reloaded[df_refData_reloaded["Market Cap"] >␣
 ↪100]
```

```
Wall time: 5.99 ms
```

Here and alsewhere we use the syntax

```
%%time

statement(s)_in_a_cell
```

to execute a all the statements in a cell and measure its execution time (Wall time) with the `%%time` magic function in cell mode.

[57]:

```
df_refData_blueChips
```

[57]:

```
       S&P Rating  Spread Country  Market Cap
Firms
Firm_1          A     100     USA      430.00
Firm_3         AA      70      UK      161.25
```

TAKE-HOME MESSAGE: in general, for small to medium-sized dbs, querying in-memory the whole DataFrame (using `SELECT *...` kind of query) and then manipulating the reconstructed DataFrame to make conditional selections etc. is in general more efficient than running conditional SQL query on the db. The opposite is true when dimensions of the db are huge and/or the conditional selection involves multiple tables or operations which would be hardly replicated in Pandas: in this case is better to run a conditional SQL query (using for example the `SELECT ... WHERE...` kind of query)

Let's now close the connection and clean-up `Data` folder

[58]:

```
con.close()
```

[59]:

```
removeFile(filePath)
```

```
Success: file ../Data\refData.db successfully removed!
```

3.1.3 Parsing Dates: pd.read_sql(..., parse_dates)

Let's make now an example of a DataFrame with Dates as index

[60]:

```
df = pd.DataFrame(data=np.array([[i**k for i in range(1,11)] for k in␣
 ↪range(1,6)]).T,
                  index=pd.date_range('2020-01-01', periods=10, freq='B'),
                  columns=['x', 'x^2', 'x^3', 'x^4', 'x^5'])
df
```

[60]:

```
             x  x^2   x^3    x^4     x^5
2020-01-01   1    1     1      1       1
2020-01-02   2    4     8     16      32
2020-01-03   3    9    27     81     243
2020-01-06   4   16    64    256    1024
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

[61]:

```
df.index
```

[61]:

```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
               '2020-01-13', '2020-01-14'],
              dtype='datetime64[ns]', freq='B')
```

[62]:

```
df.index[0]
```

[62]:

```
Timestamp('2020-01-01 00:00:00', freq='B')
```

Let's open the db connection

[63]:

```
filePath = os.path.join(dataFolderPath, "df.db")
```

[64]:

```
con = sq3.connect(filePath)
```

and let's save `df` DataFrame as a `df` table in a `"df.db"` file

[65]:

```
df.to_sql(name="df", con=con, index_label="Dates")
```

As you can see with DB Browser, the `"Dates"` column, is stored in the `df` table as a `TIMESTAMP` column. Let's what happen if we try to reconstruct the DataFrame

[66]:

```
query = "SELECT * FROM df"
df_reloaded = pd.read_sql(sql=query, con=con, index_col="Dates")
df_reloaded
```

[66]:

```
                      x  x^2   x^3    x^4     x^5
Dates
2020-01-01 00:00:00   1    1     1      1       1
2020-01-02 00:00:00   2    4     8     16      32
2020-01-03 00:00:00   3    9    27     81     243
2020-01-06 00:00:00   4   16    64    256    1024
2020-01-07 00:00:00   5   25   125    625    3125
2020-01-08 00:00:00   6   36   216   1296    7776
2020-01-09 00:00:00   7   49   343   2401   16807
2020-01-10 00:00:00   8   64   512   4096   32768
2020-01-13 00:00:00   9   81   729   6561   59049
2020-01-14 00:00:00  10  100  1000  10000  100000
```

[67]:

```
df_reloaded.index
```

[67]:

```
Index(['2020-01-01 00:00:00', '2020-01-02 00:00:00', '2020-01-03 00:00:00',
       '2020-01-06 00:00:00', '2020-01-07 00:00:00', '2020-01-08 00:00:00',
       '2020-01-09 00:00:00', '2020-01-10 00:00:00', '2020-01-13 00:00:00',
       '2020-01-14 00:00:00'],
      dtype='object', name='Dates')
```

[68]:

```
df_reloaded.index[0]
```

[68]:

```
'2020-01-01 00:00:00'
```

[69]:

```
type(df_reloaded.index[0])
```

[69]:

```
str
```

As you can see, the contents of the `"Dates"` column in `df` column are interpreted as indexes in the `df_reloaded` DataFrame. Nevertheless, these index is not a DatetimeIndex, it is made of Strings…

To parse SQL `TIMESTAMP` columns as dates you have to use the additional parameter of `pd.read_sql()` function

`pd.read_sql(..., parse_dates)`

where the parameter `parse_dates` can be the name (or a List of names) of columns in the db to be parsed as dates

[70]:

```
query = "SELECT * FROM df"
```

```
df_reloaded = pd.read_sql(sql=query, con=con, index_col="Dates",␣
 ↪parse_dates="Dates")
df_reloaded
```

[70]:

```
             x  x^2   x^3    x^4     x^5
Dates
2020-01-01   1    1     1      1       1
2020-01-02   2    4     8     16      32
2020-01-03   3    9    27     81     243
2020-01-06   4   16    64    256    1024
2020-01-07   5   25   125    625    3125
2020-01-08   6   36   216   1296    7776
2020-01-09   7   49   343   2401   16807
2020-01-10   8   64   512   4096   32768
2020-01-13   9   81   729   6561   59049
2020-01-14  10  100  1000  10000  100000
```

[71]:

```
df_reloaded.index
```

[71]:

```
DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
               '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
               '2020-01-13', '2020-01-14'],
              dtype='datetime64[ns]', name='Dates', freq=None)
```

[72]:

```
df_reloaded.index[0]
```

[72]:

```
Timestamp('2020-01-01 00:00:00')
```

cool. We have correctly reconstructed our original DataFrame.

Let's now close the connection and clean-up `Data` folder

[73]:

```
con.close()
```

[74]:

```
removeFile(filePath)
```

```
Success: file ../Data\df.db successfully removed!
```

3.2 CSV: `.to_csv()` and `pd.read_csv()`

The CSV format (for Comma Separated Values) is a delimited text file that uses a comma to separate values. Doing IO operations with this format is very simple, since Pandas has the built-in method `DataFrame.to_csv()` and the function `pd.read_csv()`.

We work first with our reference data DataFrame

[75]:

```
df_refData
```

```
[75]:        S&P Rating  Spread Country  Market Cap
      Firm_1          A     100     USA      430.00
      Firm_2         BB     300     ITA       45.00
      Firm_3         AA      70      UK      161.25
      Firm_4        CCC     700     ITA        5.00
```

```
[76]: filePath = os.path.join(dataFolderPath, "df_refData.csv")
```

To save `df_refData` as a `.csv` file we use the `.to_csv()` method whose essential syntax is

`DataFrame.to_csv(path_or_buf)`

where the `path_or_buf` parameter is as String representing the complete path of the `.csv` file we want to save.

```
[77]: %time df_refData.to_csv(path_or_buf = filePath)
```

```
Wall time: 5.98 ms
```

To reload in memory the `.csv` file and reconstruct the original DataFrame, we use the `pd.read_csv()` function whose essential syntax is

`pd.read_csv(filepath_or_buffer, index_col)`

where

- `filepath_or_buffer` parameter is as String representing the complete path of the `.csv` file we want to load.
- `index_col` is an Integer parameter which specifies the column position (e.g. the `0` for the first) to be used as index of the reconstructed DataFrame

```
[78]: %time df_refData_reloaded = pd.read_csv(filepath_or_buffer = filePath,␣
       ↪index_col = 0)
      df_refData_reloaded
```

```
Wall time: 11 ms
```

```
[78]:        S&P Rating  Spread Country  Market Cap
      Firm_1          A     100     USA      430.00
      Firm_2         BB     300     ITA       45.00
      Firm_3         AA      70      UK      161.25
      Firm_4        CCC     700     ITA        5.00
```

Let's now clean-up `Data` folder

```
[79]: removeFile(filePath)
```

```
Success: file ../Data\df_refData.csv successfully removed!
```

3.2.1 Parsing Dates: pd.read_csv(..., parse_dates)

As was the case for pd.read_sql(), in the case of a DataFrame having Dates as indexes, as df DataFrame. The parsing of the index column as an appropriate DatetimeIndex is not automatic

```
[80]: df
```

```
[80]:              x  x^2   x^3    x^4     x^5
      2020-01-01   1    1     1      1       1
      2020-01-02   2    4     8     16      32
      2020-01-03   3    9    27     81     243
      2020-01-06   4   16    64    256    1024
      2020-01-07   5   25   125    625    3125
      2020-01-08   6   36   216   1296    7776
      2020-01-09   7   49   343   2401   16807
      2020-01-10   8   64   512   4096   32768
      2020-01-13   9   81   729   6561   59049
      2020-01-14  10  100  1000  10000  100000
```

Let's save the df DataFrame in a "df.csv" file

```
[81]: filePath = os.path.join(dataFolderPath, "df.csv")
```

```
[82]: %time df.to_csv(path_or_buf = filePath)
```

```
Wall time: 3.99 ms
```

and let's reload it

```
[83]: %time df_reloaded = pd.read_csv(filepath_or_buffer = filePath, index_col = 0)
      df_reloaded
```

```
Wall time: 11 ms
```

```
[83]:              x  x^2   x^3    x^4     x^5
      2020-01-01   1    1     1      1       1
      2020-01-02   2    4     8     16      32
      2020-01-03   3    9    27     81     243
      2020-01-06   4   16    64    256    1024
      2020-01-07   5   25   125    625    3125
      2020-01-08   6   36   216   1296    7776
      2020-01-09   7   49   343   2401   16807
      2020-01-10   8   64   512   4096   32768
      2020-01-13   9   81   729   6561   59049
      2020-01-14  10  100  1000  10000  100000
```

```
[84]: df_reloaded.index
```

```
[84]: Index(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06', '2020-01-07',
             '2020-01-08', '2020-01-09', '2020-01-10', '2020-01-13', '2020-01-14'],
            dtype='object')
```

```
[85]: df_reloaded.index[0]
```

```
[85]: '2020-01-01'
```

```
[86]: type(df_reloaded.index[0])
```

```
[86]: str
```

again, the index is made of Strings, not Dates… In order to have an appropriate DatetimeIndex object as index of the reconstructed `df_reloaded` DataFrame, we have to use the optional parameter

`pd.read_csv(...[, parse_dates])`

where the parameter `parse_dates` is a Bool value which, if `True`, make the values of the index of the reconstructed DataFrame parsed as Dates

```
[87]: %time df_reloaded = pd.read_csv(filepath_or_buffer = filePath, index_col = 0,␣
       ↪parse_dates = True)
```

```
Wall time: 4.99 ms
```

```
[88]: df_reloaded
```

```
[88]:              x  x^2   x^3    x^4     x^5
      2020-01-01   1    1     1      1       1
      2020-01-02   2    4     8     16      32
      2020-01-03   3    9    27     81     243
      2020-01-06   4   16    64    256    1024
      2020-01-07   5   25   125    625    3125
      2020-01-08   6   36   216   1296    7776
      2020-01-09   7   49   343   2401   16807
      2020-01-10   8   64   512   4096   32768
      2020-01-13   9   81   729   6561   59049
      2020-01-14  10  100  1000  10000  100000
```

```
[89]: df_reloaded.index
```

```
[89]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
                     '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
                     '2020-01-13', '2020-01-14'],
                    dtype='datetime64[ns]', freq=None)
```

```
[90]: df_reloaded.index[0]
```

```
[90]: Timestamp('2020-01-01 00:00:00')
```

cool. That's what we wanted!

Let's now clean-up `Data` folder

[91]:
```
removeFile(filePath)
```

```
Success: file ../Data\df.csv successfully removed!
```

3.3 Excel: `.to_excel()` and `df.read_excel()`

Using Pandas, IO operations with Excel spreadsheet are mediated by `.to_excel()` DataFrame method and `pd.read_excel()` function. Let's redo the same steps we have done in the CSV section, but saving our `df_refData` DataFrame as a `.xlsx` spreadsheet file.

[92]:
```
df_refData
```

[92]:
```
       S&P Rating  Spread Country  Market Cap
Firm_1          A     100     USA      430.00
Firm_2         BB     300     ITA       45.00
Firm_3         AA      70      UK      161.25
Firm_4        CCC     700     ITA        5.00
```

[93]:
```
filePath = os.path.join(dataFolderPath, "df_refData.xlsx")
```

To save `df_refData` as a `.xlsx` file we use the `.to_excel()` method whose essential syntax is

`DataFrame.to_excel(excel_writer[, sheet_name])`

where the

- `excel_writer` parameter is as String representing the complete path of the `.xlsx` file we want to save;
- `sheet_name` is an optional String parameter (default `"Sheet1"`) representing the sheet name where our DataFrame will be stored.

[94]:
```
%time df_refData.to_excel(excel_writer = filePath, sheet_name =␣
 ↪"Reference_Data_Table")
```

```
Wall time: 129 ms
```

To reload in memory the `.xlsx` file and reconstruct the original DataFrame, we use the `pd.read_excel()` function whose essential syntax is

`pd.read_excel(io, index_col[, sheet_name])`

where

- `io` parameter is as String representing the complete path of the `.csv` file we want to load.
- `index_col` is an Integer parameter which specifies the column position (e.g. the `0` for the first) to be used as index of the reconstructed DataFrame;
- `sheet_name` is an optional Integer or String parameter (default 0 to refer to the first sheet) representing the sheet position (if Integer) or the sheet name (if String) from where our DataFrame will be loaded.

```
[95]: %time df_refData_reloaded = pd.read_excel(io = filePath, index_col = 0,␣
       ↪sheet_name = "Reference_Data_Table")
      df_refData_reloaded
```

```
Wall time: 68.8 ms
```

```
[95]:         S&P Rating  Spread Country  Market Cap
      Firm_1           A     100     USA      430.00
      Firm_2          BB     300     ITA       45.00
      Firm_3          AA      70      UK      161.25
      Firm_4         CCC     700     ITA        5.00
```

Let's now clean-up `Data` folder

```
[96]: removeFile(filePath)
```

```
Success: file ../Data\df_refData.xlsx successfully removed!
```

3.3.1 Parsing Dates: it's automatic

Well, all the extra work you have to do when you do IO operations in case of Dates indexes is… nothing! Let's consider as usual example our `df` DataFrame

```
[97]: df
```

```
[97]:              x  x^2   x^3    x^4     x^5
      2020-01-01   1    1     1      1       1
      2020-01-02   2    4     8     16      32
      2020-01-03   3    9    27     81     243
      2020-01-06   4   16    64    256    1024
      2020-01-07   5   25   125    625    3125
      2020-01-08   6   36   216   1296    7776
      2020-01-09   7   49   343   2401   16807
      2020-01-10   8   64   512   4096   32768
      2020-01-13   9   81   729   6561   59049
      2020-01-14  10  100  1000  10000  100000
```

```
[98]: filePath = os.path.join(dataFolderPath, "df.xlsx")
```

```
[99]: %time df.to_excel(excel_writer = filePath)
```

```
Wall time: 65.8 ms
```

```
[100]: %time df_reloaded = pd.read_excel(io = filePath, index_col = 0)
       df_reloaded
```

```
Wall time: 15 ms
```

```
[100]:              x  x^2   x^3    x^4     x^5
        2020-01-01   1    1     1      1       1
        2020-01-02   2    4     8     16      32
        2020-01-03   3    9    27     81     243
        2020-01-06   4   16    64    256    1024
        2020-01-07   5   25   125    625    3125
        2020-01-08   6   36   216   1296    7776
        2020-01-09   7   49   343   2401   16807
        2020-01-10   8   64   512   4096   32768
        2020-01-13   9   81   729   6561   59049
        2020-01-14  10  100  1000  10000  100000
```

```
[101]: df_reloaded.index
```

```
[101]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
                      '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
                      '2020-01-13', '2020-01-14'],
                     dtype='datetime64[ns]', freq=None)
```

```
[102]: df_reloaded.index[0]
```

```
[102]: Timestamp('2020-01-01 00:00:00')
```

cool. The index values are automatically parsed as Dates

Let’s now clean-up `Data` folder

```
[103]: removeFile(filePath)
```

```
Success: file ../Data\df.xlsx successfully removed!
```