Data Collection and Modeling

Course 11 - HLPL Data Access

Introduction

Data collections stored in files or databases are not of much use unless they are accessible for various users and applications; up to this point we have seen how to store data in files and databases, and access this data from SQL. When data needs to be accessed by non experienced users, usually we develop applications that offer some user interface(s) for CRUD operations on data, but also generating reports and visualizations. Applications are written using High Level Programming Language(s) (HLPL) and, for data access, we use some drivers/adapters.

Data Access using Python

Many times data is stored and accessed in a combination of:

- Files, local or remote/shared
- APIs
- Databases
- Datasets (publicly available)

Files

There are several scenarios when data is shared using files; from Python there are several ways to work with files:

- using specific libraries/modules:
  - csv: csv library
  - excel: xlrd library, openpyxl library, xlwings library
  - text: standard wrapper over OS API, fileinput library, pickle library,
  - json: json library
  - xml: xml library (xml.dom.minidom)
- using pandas library

Files with pandas

“pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language”

Pandas is a library that offers tools/functions for working with data sets and analyze, clean, manipulate and explore data

Pandas can organize data in:

- Series: one dimension array that stores data of any type (homogeneous), size immutable
- DataFrames: 2 dimensional array (table with rows and columns), size mutable

Files with pandas

In order to work with pandas, it needs to be imported:

```
#importing pandas library

import pandas as pd
```

In what follows next we focus on DataFrames as they are more appropriate for several use cases; a DataFrame can be created by explicitly calling a constructor (`pd.DataFrame`) or as a result of a I/O function:

- `pd.read_excel`
- `pd.read_csv`
- `pd.read_sql`

CSV Files with pandas

When importing a csv file into pandas, there can be specified:

- File path: the path where the file is stored
- Header: the row used for specifying column names in DataFrame (default 0)
- Column delimiter: specify a custom delimiter (default ,)
- Index column: set the columns used as index of the DataFrame(default None)
- Select columns: specify columns to import as names or indices
- Omit rows: number of rows to skip at start, at end, nr rows to read
- Missing values: specify which values to be considered NaN
- Change values: convert column values as specified
- Compress/decompress file: (default infer)

CSV Files with pandas

```
#importing pandas library

import pandas as pd

#reading csv file

df=pd.read_csv('../data.csv',header=None, sep=';',nrrows=100)

df

#writing data to csv file

df.to_csv('new_data.csv')
```

Databases

Python offer several possibilities to connect to databases:

- Generic database interfaces and APIs
  - ODBC
  - ADO
- Database interfaces for RDBMSs
  - General purpose
  - Data warehouses
  - Embedded into apps
- Non-relational databases
- Native Python databases

Databases

In order to connect to a database we might use a database driver built to enable a database connections from other systems, or we might use ORMs.

Drivers:

- Relational: PyMySQL, psycopg2, mysqlclient
- No-SQL: redis-py, cassandra-python-driver, PyMongo
- Embedded: SuperSQLite

ORM:

- SQLAlchemy, PonyORM, The Django ORM

Databases

In order to work with a DBMS using a driver we need to:

- specify connection details: username, password, host, port, database name
- Connect using `connect()` method
- Create a cursor object using `cursor()` method to store records
- Use `execute()` method to run SQL queries and return results
- Extract results in the cursor using one of the methods:
  - `fetchone()`
  - `fetchmany()`
  - `fetchall()`
- Close cursor and connection: `cursor.close()` and `connection.close()`

```
import psycopg2
# Connect to an existing database
connection = psycopg2.connect(user="postgres",
password="123456", host="127.0.0.1", port="5432",
database="dcm_db")

# Create a cursor to perform database operations
cursor = connection.cursor()

# Executing a SQL query
cursor.execute("SELECT * FROM students;")
# Fetch result
record = cursor.fetchone()
print("Student: ", record, "\n")

cursor.close()
connection.close()
```

```
import psycopg2
# Connect to an existing database
connection = psycopg2.connect(user="postgres",
password="123456", host="127.0.0.1", port="5432",
database="dcm_db")

# Create a cursor to perform database operations
cursor = connection.cursor()

# Executing a SQL query
cursor.execute("INSERT INTO students values (2,’Jane
Doe’,222,’USA’, 20);")
# Persist changes
connection.commit()

cursor.close()
connection.close()
```

Databases - ORM

SQLAlchemy is an ORM that allow complex queries to be expressed easier than in other ORMs; it considers the database to be a relational algebra engine instead of just a collection of tables. Rows can be selected from tables, views, joins and other select statements; any of these units can be composed into a larger structure. SQLAlchemy's expression language builds on this concept from its core.

Databases - ORM

```
from sqlalchemy import create_engine

db_conn = "'postgresql://postgres:123456@localhost:5432/dcm_db'"
db = create_engine(db_conn)

# Create
db.execute("CREATE TABLE IF NOT EXISTS students (id int, name varchar(100), group_id int,
country_origin varchar(100), age int)")
db.execute("INSERT INTO students (id, name, group_id, country_origin, age) VALUES (1,
'John Doe', 222, 'Romania', 19)")

# Read
result_set = db.execute("SELECT * FROM students")
for r in result_set:
        print(r)

# Update
db.execute("UPDATE students SET group_id=333 WHERE id=1")

# Delete
db.execute("DELETE FROM students WHERE age=19")
```

Databases - pandas

Pandas is specially built for data preprocessing and offers a more user friendly way of working with data than SQL; when data is stored in relational databases, it has to be retrieved and stored in pandas dataframe:

- Specify a connection string
- Create engine using connection string
- Connect to db
- Load data into a DataFrame using a SELECT query
- Process data from DataFrame
- Close connection

```
# read data from a PostgreSQL table and load into a pandas DataFrame

import psycopg2
import pandas as pds
from sqlalchemy import create_engine

# Create an engine instance
db_conn = "'postgresql://postgres:123456@localhost:5432/dcm_db'"
db = create_engine(db_conn)

# Connect to PostgreSQL server
dbConnection      = db.connect()

# Read data from PostgreSQL database table and load into a DataFrame instance
dataFrame         = pds.read_sql_query("SELECT * FROM students;", dbConnection)

# Profile/summarize DataFrame data
dataFrame.describe()

# Close the database connection
dbConnection.close()
```

```
# read data from a PostgreSQL table and load into a pandas DataFrame

import psycopg2
import pandas as pds
from sqlalchemy import create_engine

# Create an engine instance
db_conn = "'postgresql://postgres:123456@localhost:5432/dcm_db'"
db = create_engine(db_conn)

# Connect to PostgreSQL server
dbConnection      = db.connect()

# Read data from PostgreSQL database table and load into a DataFrame instance
dataFrame         = pds.read_sql_table("students", dbConnection)

# display some DataFrame data
dataFrame.head()

# Close the database connection
dbConnection.close()
```

Databases - pandas

In the previous example we used read_sql_table and read_sql_query methods that can be replaced by read_sql method that is a wrapper around these 2 methods and it actually invokes one or the other based on the first argument supplied, a select query or a table name:

```
# Read data from PostgreSQL database table and load into a DataFrame instance
dataFrame          = pds.read_sql("SELECT * FROM students;", dbConnection)

will call read_sql_query

# Read data from PostgreSQL database table and load into a DataFrame instance
dataFrame          = pds.read_sql("students", dbConnection)

will call read_sql_table
```

Databases - pandas

When using `DataFrame` we usually store all the data in such an object; during the processing steps we apply all necessary changes to `DataFrame` and then we need to persist it into the database. In order to do that we can use `to_sql()` function:

```
# Create an engine instance
db_conn = "'postgresql://postgres:123456@localhost:5432/dcm_db'"
db = create_engine(db_conn)
# Connect to PostgreSQL server
dbConnection    = db.connect()

# prepare dataFrame data ...

dataFrame.to_sql("students", dbConnection, if_exists='replace', index=false)
# Close the database connection
dbConnection.close()
```

Resources

Bulk insert performance