CPS 5745 Interact Information Visualization

Chapter 2: Reading and Writing Data with Python

Dr. Hamza Djigal (dhamza@kean.edu)

Semester: Fall 2024

Date:September 2024

温州肯恩大学
WENZHOU-KEAN UNIVERSITY

Objective

By the end of this lesson, you will be able to:

- Create Comma-separated values (CSV) files from Python Dictionary
- Read and write CSV using Python

Working with System Files

Create a CSV file

1) Let consider the following dictionary nobel_winners.

```
1 nobel_winners = [
2 {'category': 'Physics','name': 'Albert Einstein','
    nationality': 'Swiss','sex': 'male','year':
    1921},
3 {'category': 'Physics','name': 'Paul Dirac','
    nationality': 'British','sex': 'male','year':
    1933},
4 {'category': 'Chemistry','name': 'Marie Curie','
    nationality': 'Polish','sex': 'female','year':
    1911}
5 ]
```

2) Open a new file, using **w**

```
1 f = open('data/nobel_winners.csv', 'w')
```

Working with System Files

Create a CSV file

3) Create the CSV file from the nobel_winners dictionary.

```
2  cols = nobel_winners[0].keys // data columns from
       the keys
3  with open('data/nobel_winners.csv, 'w') as f:
4      f.write(','.join(cols) + '\n')
5      for o in nobel_winners:
6          row = [str(o[col]) for col in cols]
7          f.write(','.join(row) + '\n')
```

4) **Read the CSV file:**

```
8  with open('data/nobel_winners.csv') as f:
9        for line in f.readlines():
10           print(line),
```

Working with System Files

Create a CSV file

- CSV or their Tab-Separated (TSV) are the most ubiquitous file-based data formats.
- CSV has a dedicated **DictWriter** class that will turn a dictionary into CVS rows

```
1 import csv
2
3 with open('../data/nobel_winners.csv', 'w') as
      f:
4     fieldnames = nobel_win[0].keys()
5     writer = csv.DictWriter(f, fieldnames=
          fieldnames)
6     writer.writeheader()
7     for w in nobel_win:
8         writer.writerow(w)
```

Working with System Files

Read a CSV file

```
1 with open('../data/nobel_winners.csv') as f:
2     reader = csv.reader(f)
3     for row in reader:
4         print(row)
```

Read and Wrie JSON files

to the file access. Loading JSON strings to Python containers and dumping Python containers to JSON strings.

```
1 import json
2 with open('data/nobel_winners.json', 'w') as f:
3         json.dump(nobel_win, f)
4
5 open('data/nobel_winners.json').read()
```

Working with System Files

Read and Wrie JSON files

to the file access. Loading JSON strings to Python containers and dumping Python containers to JSON strings.

```
1 import json
2 with open('data/nobel_winners.json') as f:
3     nobel_winners = json.load(f)
4 nobel_winners
```

Dealing with Dates and Times

Trying to dump a datetime object to json produces a TypeError:

```
1 from datetime import datetime
2 json.dumps(datetime.now())
```

TypeError: Object of type datetime is not JSON serializable

Working with System Files

Dealing with Dates and Times

The json encoders and decoders can serialize simple datatypes (strings or numbers), but not for dates

```
1  import datetime
2  from dateutil import parser
3  import json
4  class JSONDateTimeEncoder(json.JSONEncoder):
5      def default(self, obj):
6          if isinstance(obj, (datetime.date, datetime.
               datetime)):
7              return obj.isoformat()
8          else:
9              return json.JSONEncoder.default(self, obj)
10 def dumps(obj):
11     return json.dumps(obj, cls=JSONDateTimeEncoder)
12 now_str = dumps({'time': datetime.datetime.now()})
13 now_str
```

Working with System Files

Dealing with Dates and Times

The **strptime** method: tries to match the time string to a format string using various directives such as %Y (year with century) and %H (hour as a zero-padded decimal number).

```
1 from datetime import datetime
2 time_str = '2012/01/01 12:32:11'
3 dt = datetime.strptime(time_str, '%Y/%m/%d %H:%M:%S
      ')
4 dt
```

Output:datetime.datetime(2012, 1, 1, 12, 32, 11)

Read and Write Data from SQL

SQL

- **SQLAlchemy** is the most popular Python library for interacting with an SQL database.
- **SQLAlchemy** provides a powerful object-relational mapping (ORM) that allows you to operate on SQL tables using a high-level, Pythonic API, treating them essentially as Python classes.

Write Data to an SQLite file using SQLAlchemy

- **Database Engine**: establishes a connection with the DB in question and perform any conversions needed to the generic SQL instructions generated by SQLAlchemy and the data being returned.
- DB engines are interchangeable, i.e., you could develop your code using the file-based SQLite DB and then switch during production to an industrial DB, such as Postgresql.

Read and Write Data from SQL

Syntax of DB URL using SQLAlchemy:

```
dialect+driver://username:password@host:port/
    database
```

Syntax to connect to MySQL DB URL using SQLAlchemy:

```
engine = create_engine(\'mysql://root:
    mypsswd@localhost/nobel_prize')
```

Syntax to connect to SQLite DB URL using SQLAlchemy:

```
engine = create_engine('sqlite:///nobel_prize.db',
    echo=True)
```

echo='True' allow to see the SQL instructions generated by SQLAlchemy from the command line.

Read and Write Data from SQL

Defining the Database Tables

declarative_base creates a Base class that will be used to create table classes

```
1 from sqlalchemy.orm import declarative_base
2 Base = declarative_base()
```

Note: Declarative Extensions is now integrated into the SQLAlchemy ORM.

Read and Write Data from SQL

Defining an SQL database table

```
1  from sqlalchemy import Column, Integer, String,
       Enum
2
3  class Winner(Base):
4      __tablename__ = 'winners'
5      id = Column(Integer, primary_key=True)
6      name = Column(String)
7      category = Column(String)
8      year = Column(Integer)
9      nationality = Column(String)
10     sex = Column(Enum('male', 'female'))
11
12     def __repr__(self):
13         return "<Winner(name='%s', category='%s',
               year='%s')>" \
14              % (self.name, self.category, self.year)
```

Read and Write Data from SQL

Create the database tables

```
1 Base.metadata.create_all(engine)
```

Adding Instances with a Session

After created a DB a session is needed to interact with it.

```
1 from sqlalchemy.orm import sessionmaker
2 Session = sessionmaker(bind=engine)
3 session = Session()
```

Use the created Winner class to create instances and table rows and add them to the session:

```
1 albert = Winner(**nobel_winners[0])
2 session.add(albert)
3 session.new
```

Read and Write Data from SQL

Use the created Winner class to create instances and table rows and add them to the session:

```
albert = Winner(**nobel_winners[0])
session.add(albert)
session.new
```

- ** operator unpacks our first nobel_winners member into key-value pairs:
- All database insertions and deletions take place in Python.
- **commit** method alteres the database.
- **expunge** method removes the object added to the session.
- **expunge_all** method removes all new objects added to the session

```
session.expunge(albert)
session.new
```

Read and Write Data from SQL

Add all the members of our nobel_winners list to the session and commit them to the database

```
1 winner_rows = [Winner(**w) for w in nobel_winners]
2 session.add_all(winner_rows)
3 session.commit()
```

Querying the Database

To access data, you use the **session's query** method, the result of which can be filtered, grouped, and intersected, allowing the full range of standard SQL data retrieval.

1)Count the number of rows in our winners table:

```
1 session.query(Winner).count()
```

Read and Write Data from SQL

Querying the Database

2)Retrieve all Swiss winners:

```
1 q = session.query(Winner).filter_by(nationality='
      Swiss')
2 list(q)
```

3) Get all non-Swiss Physics winners:

```
1 q = session.query(Winner).filter(Winner.category ==
      'Physics', Winner.nationality != 'Swiss')
2 list(q)
```

4) Get a row based on ID number:

```
1 session.query(Winner).get(3)
```

Read and Write Data from SQL

Querying the Database

5) Retrieve winners ordered by year:

```
1 res = session.query(Winner).order_by('year')
2 list(res)
```

Read and Write Data from SQL

Converts an SQLAlchemy instance to a dict

1) Write a function to create a dict from an SQLAlchemy class:

```
1 def inst_to_dict(inst, delete_id=True):
2     dat = {}
3     for column in inst.__table__.columns:
4         dat[column.name] = getattr(inst, column.
              name)
5     if delete_id:
6         dat.pop('id')
7     return dat
```

2) Reconstruct our nobel_winners target list

```
1 win_rows = session.query(Winner)
2 nobel_winners = [inst_to_dict(w) for w in win_rows]
3 nobel_winners
```

Read and Write Data from SQL

Update database rows

```
1 marie = session.query(Winner).get(3)
2 marie.nationality = 'French'
3 session.dirty # instances not yet committed to DB.
4 session.commit()
```

Delete the results of a query

```
1 session.query(Winner).filter_by(name='Albert
      Einstein').delete()
2 list(session.query(Winner))
```

Drop the whole table

```
1 Winner.__table__.drop(engine)
```

Dataset

Dataset

Dataset is a module designed to make working with SQL databases a easier and more Pythonic than existing powerhouses like SQLAlchemy.

- **Dataset** is great for basic SQL-based work, particularly retrieving data you might wish to process or visualize.

```
1 marie = session.query(Winner).get(3)
2 marie.nationality = 'French'
3 session.dirty # instances not yet committed to DB.
4 session.commit()
```

Delete the results of a query

```
1 session.query(Winner).filter_by(name='Albert
    Einstein').delete()
2 list(session.query(Winner))
```