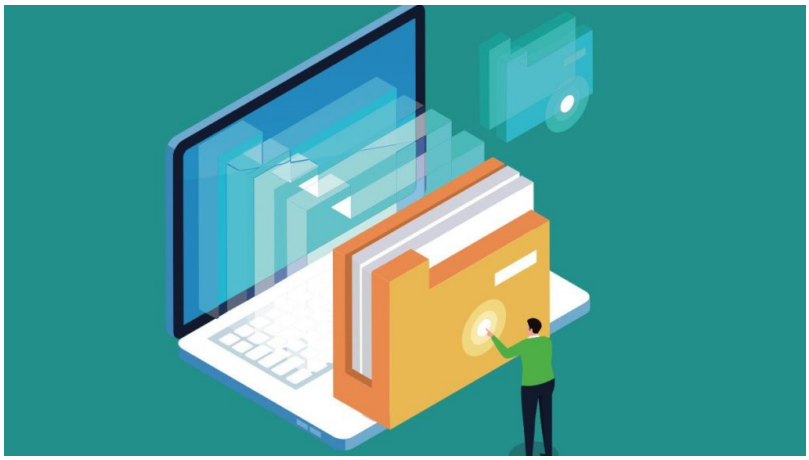
Reading data from relational databases



In this lesson you will learn how to read SQL queries and relational database tables into `DataFrame` objects using pandas. Also, we'll take a look at different techniques to persist that pandas `DataFrame` objects to database tables.

Hands on!

In [1]:

```
import pandas as pd
```

Read data from SQL database

Reading data from SQL relational databases is fairly simple and pandas support a variety of methods to deal with it.

We'll start with an example using SQLite, as it's a builtin Python package, and we don't need anything extra installed.

In [2]:

```
import sqlite3
```

In order to work with a SQLite database from Python, we first have to connect to it. We can do that using the connect function, which returns a `Connection` object.

We'll use the following database structure:

In [3]:

```
conn = sqlite3.connect('chinook.db')
```

Once we have a `Connection` object, we can then create a `Cursor` object. Cursors allow us to execute SQL queries against a database:

In [4]:

```
cur = conn.cursor()
```

The `Cursor` created has a method `execute`, which will receive SQL parameters to run against the database.

The code below will fetch the first `5` rows from the `employees` table:

In [5]:

```
cur.execute('SELECT * FROM employees LIMIT 5;')
```

Out[5]:

```
<sqlite3.Cursor at 0x7f8b83eef7a0>
```

You may have noticed that we didn't assign the result of the above query to a variable. This is because we need to run another command to actually fetch the results.

We can use the `fetchall` method to fetch all of the results of a query:

In [6]:

```
results = cur.fetchall()
```

In [7]:

```
results
```

Out[7]:

```
[(1,
  'Adams',
  'Andrew',
  'General Manager',
  None,
  '1962-02-18 00:00:00',
  '2002-08-14 00:00:00',
  '11120 Jasper Ave NW',
  'Edmonton',
  'AB',
  'Canada',
  'T5K 2N1',
  '+1 (780) 428-9482',
  '+1 (780) 428-3457',
  'andrew@chinookcorp.com'),
 (2,
  'Edwards',
  'Nancy',
  'Sales Manager',
  1,
  '1958-12-08 00:00:00',
  '2002-05-01 00:00:00',
  '825 8 Ave SW',
  'Calgary',
  'AB',
  'Canada',
  'T2P 2T3',
  '+1 (403) 262-3443',
  '+1 (403) 262-3322',
  'nancy@chinookcorp.com'),
 (3,
  'Peacock',
  'Jane',
  'Sales Support Agent',
  2,
  '1973-08-29 00:00:00',
  '2002-04-01 00:00:00',
  '1111 6 Ave SW',
  'Calgary',
  'AB',
  'Canada',
  'T2P 5M5',
  '+1 (403) 262-3443',
  '+1 (403) 262-6712',
  'jane@chinookcorp.com'),
 (4,
  'Park',
  'Margaret',
  'Sales Support Agent',
  2,
  '1947-09-19 00:00:00',
  '2003-05-03 00:00:00',
  '683 10 Street SW',
  'Calgary',
  'AB',
  'Canada',
  'T2P 5G3',
  '+1 (403) 263-4423',
  '+1 (403) 263-4289',
```

```
 'margaret@chinookcorp.com'),
(5,
 'Johnson',
 'Steve',
 'Sales Support Agent',
 2,
 '1965-03-03 00:00:00',
 '2003-10-17 00:00:00',
 '7727B 41 Ave',
 'Calgary',
 'AB',
 'Canada',
 'T3B 1Y7',
 '1 (780) 836-9987',
 '1 (780) 836-9543',
 'steve@chinookcorp.com')]
```

As you can see, the results are returned as a list of tuples. Each tuple corresponds to a row in the database that we accessed. Dealing with data this way is painful.

We'd need to manually add column headers, and manually parse the data. Luckily, the pandas library has an easier way, which we'll look at in the next section.

In [8]:

```
df = pd.DataFrame(results)
```

In [9]:

```
df.head()
```

Out[9]:

	0	1	2	3	4	5	6	7	8	9	10	
0	1	Adams	Andrew	General Manager	NaN	1962-02-18 00:00:00	2002-08-14 00:00:00	11120 Jasper Ave NW	Edmonton	AB	Canada	T 2
1	2	Edwards	Nancy	Sales Manager	1.0	1958-12-08 00:00:00	2002-05-01 00:00:00	825 8 Ave SW	Calgary	AB	Canada	T 2
2	3	Peacock	Jane	Sales Support Agent	2.0	1973-08-29 00:00:00	2002-04-01 00:00:00	1111 6 Ave SW	Calgary	AB	Canada	T 5I
3	4	Park	Margaret	Sales Support Agent	2.0	1947-09-19 00:00:00	2003-05-03 00:00:00	683 10 Street SW	Calgary	AB	Canada	T 5(
4	5	Johnson	Steve	Sales Support Agent	2.0	1965-03-03 00:00:00	2003-10-17 00:00:00	7727B 41 Ave	Calgary	AB	Canada	T 1

Before we move on, it's good practice to close `Connection` objects and `Cursor` objects that are open. This prevents the SQLite database from being locked. When a SQLite database is locked, you may be unable to update the database, and may get errors. We can close the Cursor and the Connection like this:

In [10]:

```
cur.close()
conn.close()
```

Using pandas `read_sql` method

We can use the pandas `read_sql` function to read the results of a SQL query directly into a pandas `DataFrame`. The code below will execute the same query that we just did, but it will return a `DataFrame`. It has several advantages over the query we did above:

- It doesn't require us to create a `Cursor` object or call `fetchall` at the end.
- It automatically reads in the names of the headers from the table.
- It creates a `DataFrame`, so we can quickly explore the data.

In [11]:

```
conn = sqlite3.connect('chinook.db')
```

In [12]:

```
df = pd.read_sql('SELECT * FROM employees;', conn)
```

In [13]:

```
df.head()
```

Out[13]:

	EmployeeId	LastName	FirstName	Title	ReportsTo	BirthDate	HireDate	Address	
0	1	Adams	Andrew	General Manager	NaN	1962-02-18 00:00:00	2002-08-14 00:00:00	11120 Jasper Ave NW	Edı
1	2	Edwards	Nancy	Sales Manager	1.0	1958-12-08 00:00:00	2002-05-01 00:00:00	825 8 Ave SW	(
2	3	Peacock	Jane	Sales Support Agent	2.0	1973-08-29 00:00:00	2002-04-01 00:00:00	1111 6 Ave SW	(
3	4	Park	Margaret	Sales Support Agent	2.0	1947-09-19 00:00:00	2003-05-03 00:00:00	683 10 Street SW	(
4	5	Johnson	Steve	Sales Support Agent	2.0	1965-03-03 00:00:00	2003-10-17 00:00:00	7727B 41 Ave	(

In [14]:

```
df = pd.read_sql('SELECT * FROM employees;', conn,
                 index_col='EmployeeId',
                 parse_dates=['BirthDate', 'HireDate'])
```

In [15]:

```
df.head()
```

Out[15]:

EmployeeId	LastName	FirstName	Title	ReportsTo	BirthDate	HireDate	Address	
1	Adams	Andrew	General Manager	NaN	1962-02-18	2002-08-14	11120 Jasper Ave NW	Edmor
2	Edwards	Nancy	Sales Manager	1.0	1958-12-08	2002-05-01	825 8 Ave SW	Calg
3	Peacock	Jane	Sales Support Agent	2.0	1973-08-29	2002-04-01	1111 6 Ave SW	Calg
4	Park	Margaret	Sales Support Agent	2.0	1947-09-19	2003-05-03	683 10 Street SW	Calg
5	Johnson	Steve	Sales Support Agent	2.0	1965-03-03	2003-10-17	7727B 41 Ave	Calg

In [16]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8 entries, 1 to 8
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype         
---  ------      --------------  -----         
 0   LastName    8 non-null      object        
 1   FirstName   8 non-null      object        
 2   Title       8 non-null      object        
 3   ReportsTo   7 non-null      float64       
 4   BirthDate   8 non-null      datetime64[ns]
 5   HireDate    8 non-null      datetime64[ns]
 6   Address     8 non-null      object        
 7   City        8 non-null      object        
 8   State       8 non-null      object        
 9   Country     8 non-null      object        
 10  PostalCode  8 non-null      object        
 11  Phone       8 non-null      object        
 12  Fax         8 non-null      object        
 13  Email       8 non-null      object        
dtypes: datetime64[ns](2), float64(1), object(11)
memory usage: 960.0+ bytes
```

In [ ]:

```
df['ReportsTo'].isna().sum()
```

In [ ]:

```
df['ReportsTo'].mean()
```

In [ ]:

```
df['ReportsTo'] > 1.75
```

In [ ]:

```
df['City'] = df['City'].astype('category')
```

In [ ]:

```
df.info()
```

Using pandas `read_sql_query` method

It turns out that the `read_sql` method we saw above is just a wrapper around `read_sql_query` and `read_sql_table`.

We can get the same result using `read_sql_query` method:

In [17]:

```
conn = sqlite3.connect('chinook.db')
```

In [18]:

```
df = pd.read_sql_query('SELECT * FROM employees LIMIT 5;', conn)
```

In [19]:

```
df.head()
```

Out[19]:

	EmployeeId	LastName	FirstName	Title	ReportsTo	BirthDate	HireDate	Address	
0	1	Adams	Andrew	General Manager	NaN	1962-02-18 00:00:00	2002-08-14 00:00:00	11120 Jasper Ave NW	Edı
1	2	Edwards	Nancy	Sales Manager	1.0	1958-12-08 00:00:00	2002-05-01 00:00:00	825 8 Ave SW	(
2	3	Peacock	Jane	Sales Support Agent	2.0	1973-08-29 00:00:00	2002-04-01 00:00:00	1111 6 Ave SW	(
3	4	Park	Margaret	Sales Support Agent	2.0	1947-09-19 00:00:00	2003-05-03 00:00:00	683 10 Street SW	(
4	5	Johnson	Steve	Sales Support Agent	2.0	1965-03-03 00:00:00	2003-10-17 00:00:00	7727B 41 Ave	(

In [ ]:

```
df = pd.read_sql_query('SELECT * FROM employees;', conn,
                       index_col='EmployeeId',
                       parse_dates=['BirthDate', 'HireDate'])
```

In [ ]:

```
df.head()
```

Using `read_sql_table` method

`read_sql_table` is a useful function, but it works only with SQLAlchemy (https://www.sqlalchemy.org/), a Python SQL Toolkit and Object Relational Mapper.

This is just a demonstration of its usage where we read the whole `employees` table.

In [20]:

```
from sqlalchemy import create_engine
```

In [21]:

```
engine = create_engine('sqlite:///chinook.db')

connection = engine.connect()
```

In [22]:

```
df = pd.read_sql_table('employees', con=connection)
```

In [23]:

```
df.head()
```

Out[23]:

	EmployeeId	LastName	FirstName	Title	ReportsTo	BirthDate	HireDate	Address	
0	1	Adams	Andrew	General Manager	NaN	1962-02-18	2002-08-14	11120 Jasper Ave NW	Edı
1	2	Edwards	Nancy	Sales Manager	1.0	1958-12-08	2002-05-01	825 8 Ave SW	(
2	3	Peacock	Jane	Sales Support Agent	2.0	1973-08-29	2002-04-01	1111 6 Ave SW	(
3	4	Park	Margaret	Sales Support Agent	2.0	1947-09-19	2003-05-03	683 10 Street SW	(
4	5	Johnson	Steve	Sales Support Agent	2.0	1965-03-03	2003-10-17	7727B 41 Ave	(

In [24]:

```
df = pd.read_sql_table('employees', con=connection,
                       index_col='EmployeeId',
                       parse_dates=['BirthDate', 'HireDate'])
```

In [ ]:

```
df.head()
```

In [ ]:

```
connection.close()
```

Create tables from `DataFrame` objects

Finally we can persist `DataFrame` objects we've working on in a database using the pandas `to_sql` method.

Although it is easy to implement, it could be a very slow process.

In [ ]:

```
df.head()
```

Drop the table if needed

In [25]:

```
df.to_sql?
```

```
Signature:
df.to_sql(
    name: str,
    con,
    schema=None,
    if_exists: str = 'fail',
    index: bool = True,
    index_label=None,
    chunksize=None,
    dtype=None,
    method=None,
) -> None
Docstring:
Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1]_ are supported. Tables can be
newly created, appended to, or overwritten.

Parameters
----------
name : str
    Name of SQL table.
con : sqlalchemy.engine.Engine or sqlite3.Connection
    Using SQLAlchemy makes it possible to use any DB supported by th
at
    library. Legacy support is provided for sqlite3.Connection objec
ts. The user
    is responsible for engine disposal and connection closure for th
e SQLAlchemy
    connectable See `here                 <https://docs.sqlalchemy.o
rg/en/13/core/connections.html>`_

schema : str, optional
    Specify the schema (if database flavor supports this). If None,
 use
    default schema.
if_exists : {'fail', 'replace', 'append'}, default 'fail'
    How to behave if the table already exists.

    * fail: Raise a ValueError.
    * replace: Drop the table before inserting new values.
    * append: Insert new values to the existing table.

index : bool, default True
    Write DataFrame index as a column. Uses `index_label` as the col
umn
    name in the table.
index_label : str or sequence, default None
    Column label for index column(s). If None is given (default) and
    `index` is True, then the index names are used.
    A sequence should be given if the DataFrame uses MultiIndex.
chunksize : int, optional
    Specify the number of rows in each batch to be written at a tim
e.
    By default, all rows will be written at once.
dtype : dict or scalar, optional
    Specifying the datatype for columns. If a dictionary is used, th
e
    keys should be the column names and the values should be the
    SQLAlchemy types or strings for the sqlite3 legacy mode. If a
    scalar is provided, it will be applied to all columns.
```

```
method : {None, 'multi', callable}, optional
    Controls the SQL insertion clause used:

    * None : Uses standard SQL ``INSERT`` clause (one per row).
    * 'multi': Pass multiple values in a single ``INSERT`` clause.
    * callable with signature ``(pd_table, conn, keys, data_iter)``.

    Details and a sample callable implementation can be found in the
    section :ref:`insert method <io.sql.method>`.

    .. versionadded:: 0.24.0

Raises
------
ValueError
    When the table already exists and `if_exists` is 'fail' (the
    default).

See Also
--------
read_sql : Read a DataFrame from a table.

Notes
-----
Timezone aware datetime columns will be written as
``Timestamp with timezone`` type with SQLAlchemy if supported by the
database. Otherwise, the datetimes will be stored as timezone unawar
e
timestamps local to the original timezone.

.. versionadded:: 0.24.0

References
----------
.. [1] http://docs.sqlalchemy.org
.. [2] https://www.python.org/dev/peps/pep-0249/

Examples
--------

Create an in-memory SQLite database.

>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)

Create a table from scratch with 3 rows.

>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3

>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]

>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
```

```
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]

Overwrite the table with just ``df1``.

>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]

Specify the dtype (especially useful for integers with missing value
s).
Notice that while pandas is forced to store the data as floating poi
nt,
the database supports nullable integers. When fetching the data with
Python, we get back integer scalars.

>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0

>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})

>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
File:      /usr/local/lib/python3.8/site-packages/pandas/core/generi
c.py
Type:      method
```

In [ ]:

```
cur = conn.cursor()
```

In [ ]:

```
cur.execute('DROP TABLE IF EXISTS employees2;')
```

In [ ]:

```
cur.close()
```

In [ ]:

```
df.to_sql('employees2', conn)
```

In [ ]:

```
pd.read_sql_query('SELECT * FROM employees2;', conn).head()
```

Custom behavior

The `if_exists` parameter define how to behave if the table already exists and adds a ton of flexibility, letting you decide wheather to `replace` current table data, `append` new data at the end, or simply `fail` if table already exists.

In [ ]:

```
pd.DataFrame().to_sql('employees2',
                      conn,
                      if_exists='replace')
```

In [ ]:

```
pd.read_sql_query('SELECT * FROM employees2;', conn).head()
```

In [ ]:

```
df.to_sql('employees2',
          conn,
          if_exists='replace')
```

In [ ]:

```
pd.read_sql_query('SELECT * FROM employees2;', conn).head()
```

In [ ]:

```
conn.close()
```