Course › Modul... › Assign... › Sample...

Sample solutions

part0 (Score: 7.0 / 7.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 2.0 / 2.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 2.0 / 2.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel→Restart.
- Then **run all cells** -- in the menubar, select Cell→Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Lesson 0: SQLite

The de facto language for managing relational databases is the Structured Query Language, or SQL ("sequel").

Many commerical and open-source relational data management systems (RDBMS) support SQL. The one we will consider in this class is the simplest, called sqlite3 (https://www.sqlite.org/). It stores the database in a simple file and can be run in a "standalone" mode from the command-line. However, we will, naturally, invoke it from Python (https://docs.python.org/3/library/sqlite3.html). But all of the basic techniques apply to any commercial SQL backend.

With a little luck, you might by the end of this class understand this xkcd comic on SQL injection attacks (http://xkcd.com/327).

Getting started

In Python, you connect to an `sqlite3` database by creating a connection object.

Exercise 0 (ungraded). Run this code cell to get started.

In [1]: Grade cell: `who__test` Score: 1.0 / 1.0 (Top)

```
import sqlite3 as db

# Connect to a database (or create one if it doesn't exist)
conn = db.connect('example.db')
```

The `sqlite` engine maintains a database as a file; in this example, the name of that file is `example.db`.

> **Important usage note!** If the named file does **not** yet exist, this code creates it. However, if the database has been created before, this same code will open it. This fact can be important when you are debugging. For example, if your code depends on the database not existing initially, then you may need to remove the file first.

You issue commands to the database through an object called a cursor.

```
In [2]: # Create a 'cursor' for executing commands
        c = conn.cursor()
```

A cursor tracks the current state of the database, and you will mostly be using the cursor to issue commands that modify or query the database.

Tables and Basic Queries

The central object of a relational database is a table. It's identical to what you called a "tibble" in the tidy data lab: observations as rows, variables as columns. In the relational database world, we sometimes refer to as items or records and columns as attributes. We'll use all of these terms interchangeably in this course.

Let's look at a concrete example. Suppose we wish to maintain a database of Georgia Tech students, whose attributes are their names and Georgia Tech-issued ID numbers. You might start by creating a table named `Students` to hold this data. You can create the table using the command, `create table` (https://www.sqlite.org/lang_createtable.html).

> Note: If you try to create a table that already exists, it will **fail**. If you are trying to carry out these exercises from scratch, you may need to remove any existing `example.db` file or destroy any existing table; you can do the latter with the SQL command, `drop table if exists Students`.

```
In [3]: c.execute("drop table if exists Students")
        c.execute("create table Students (gtid integer, name text)")

Out[3]: <sqlite3.Cursor at 0x10f375d50>
```

To populate the table with items, you can use the command, `insert into` (https://www.sqlite.org/lang_insert.html).

```
In [4]: c.execute("insert into Students values (123, 'Vuduc')")
        c.execute("insert into Students values (456, 'Chau')")
        c.execute("insert into Students values (381, 'Bader')")
        c.execute("insert into Students values (991, 'Sokol')")

Out[4]: <sqlite3.Cursor at 0x10f375d50>
```

Commitment issues. The commands above modify the database. However, these are temporary modifications and aren't actually saved to the databases until you say so. (Aside: Why would you want such behavior?) The way to do that is to issue a commit operation from the connection object.

> There are some subtleties related to when you actually need to commit, since the SQLite database engine does commit at certain points as discussed here (https://stackoverflow.com/questions/13642956/commit-behavior-and-atomicity-in-python-sqlite3-module). However, it's probably simpler if you remember to encode commits when you intend for them to take effect.

```
In [5]: conn.commit()
```

Another common operation is to perform a bunch of insertions into a table from a list of tuples. In this case, you can use `executemany()`.

```
In [6]: # An important (and secure!) idiom
        more_students = [(723, 'Rozga'),
                         (882, 'Zha'),
                         (401, 'Park'),
                         (377, 'Vetter'),
                         (904, 'Brown')]

        c.executemany('insert into Students values (?, ?)', more_students)
        conn.commit()
```

Given a table, the most common operation is a query, which asks for some subset or transformation of the data. The simplest kind of query is called a `select` (https://www.sqlite.org/lang_select.html).

The following example selects all rows (items) from the `Students` table.

In [7]:
```
c.execute("select * from Students")
results = c.fetchall()
print("Your results:", len(results), "\nThe entries of Students:\n", results)
```

```
Your results: 9
The entries of Students:
 [(123, 'Vuduc'), (456, 'Chau'), (381, 'Bader'), (991, 'Sokol'), (723, 'Rozga'), (882, 'Z
ha'), (401, 'Park'), (377, 'Vetter'), (904, 'Brown')]
```

Exercise 1 (2 points). Suppose we wish to maintain a second table, called `Takes`, which records classes that students have taken and the grades they earn.

In particular, each row of `Takes` stores a student by his/her GT ID, the course he/she took, and the grade he/she earned. More formally, suppose this table is defined as follows:

In [8]:
```
# Run this cell
c.execute('drop table if exists Takes')
c.execute('create table Takes (gtid integer, course text, grade real)')
```

Out[8]:
```
<sqlite3.Cursor at 0x10f375d50>
```

Write a command to insert the following records into the `Takes` table.

- Vuduc: CSE 6040 - A (4.0), ISYE 6644 - B (3.0), MGMT 8803 - D (1.0)
- Sokol: CSE 6040 - A (4.0), ISYE 6740 - A (4.0)
- Chau: CSE 6040 - A (4.0), CSE 6740 - C (2.0), MGMT 8803 - B (3.0)

In [9]: Student's answer (Top)

```
### BEGIN SOLUTION
takes_data = [
    (123, 'CSE 6040', 4.0),
    (123, 'ISYE 6644', 3.0),
    (123, 'MGMT 8803', 1.0),
    (991, 'CSE 6040', 4.0),
    (991, 'ISYE 6740', 4.0),
    (456, 'CSE 6040', 4.0),
    (456, 'CSE 6740', 2.0),
    (456, 'MGMT 8803', 3.0)
]
c.executemany('insert into Takes values (?, ?, ?)', takes_data)
conn.commit()
### END SOLUTION

# Displays the results of your code
c.execute('select * from Takes')
results = c.fetchall()
print("Your results:", len(results), "\nThe entries of Takes:", results)
```

```
Your results: 8
The entries of Takes: [(123, 'CSE 6040', 4.0), (123, 'ISYE 6644', 3.0), (123, 'MGMT 880
3', 1.0), (991, 'CSE 6040', 4.0), (991, 'ISYE 6740', 4.0), (456, 'CSE 6040', 4.0), (456,
'CSE 6740', 2.0), (456, 'MGMT 8803', 3.0)]
```

In [10]: Grade cell: `insert_many__test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `insert_many__test`

# Close the database and reopen it
conn.close()
conn = db.connect('example.db')
c = conn.cursor()
c.execute('select * from Takes')
results = c.fetchall()

if len(results) == 0:
    print("*** No matching records. Did you remember to commit the results? ***")
assert len(results) == 8, "The `Takes` table has {} when it should have {}.".format(len
(results), 8)

assert (123, 'CSE 6040', 4.0) in results
assert (123, 'CSE 6040', 4.0) in results
assert (123, 'ISYE 6644', 3.0) in results
assert (123, 'MGMT 8803', 1.0) in results
assert (991, 'CSE 6040', 4.0) in results
assert (991, 'ISYE 6740', 4.0) in results
assert (456, 'CSE 6040', 4.0) in results
assert (456, "CSE 6740", 2.0) in results
assert (456, "MGMT 8803", 3.0) in results

print("\n(Passed.)")
```

```
(Passed.)
```

Lesson 1: Join queries

The main type of query that combines information from multiple tables is the join query. Recall from our discussion of tibbles these four types:

- `inner-join(A, B)`: Keep rows of `A` and `B` only where `A` and `B` match
- `outer-join(A, B)`: Keep all rows of `A` and `B`, but merge matching rows and fill in missing values with some default (`NaN` in Pandas, `NULL` in SQL)
- `left-join(A, B)`: Keep all rows of `A` but only merge matches from `B`.
- `right-join(A, B)`: Keep all rows of `B` but only merge matches from `A`.

If you are a visual person, see this page (https://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins) for illustrations of the different join types.

In SQL, you can use the `where` clause of a `select` statement to specify how to match rows from the tables being joined. For example, recall that the `Takes` table stores classes taken by each student. However, these classes are recorded by a student's GT ID. Suppose we want a report where we want each student's name rather than his/her ID. We can get the matching name from the `Students` table. Here is a query to accomplish this matching:

In [11]:

```
# See all (name, course, grade) tuples
query = '''
    select Students.name, Takes.course, Takes.grade
        from Students, Takes
        where Students.gtid=Takes.gtid
'''

for match in c.execute(query): # Note this alternative idiom for iterating over query res
ults
    print(match)
```

```
('Vuduc', 'CSE 6040', 4.0)
('Vuduc', 'ISYE 6644', 3.0)
('Vuduc', 'MGMT 8803', 1.0)
('Chau', 'CSE 6040', 4.0)
('Chau', 'CSE 6740', 2.0)
('Chau', 'MGMT 8803', 3.0)
('Sokol', 'CSE 6040', 4.0)
('Sokol', 'ISYE 6740', 4.0)
```

Exercise 2 (2 points). Define a query to select only the names and grades of students who took CSE 6040. The code below will execute your query and store the results in a list `results1` of tuples, where each tuple is a `(name, grade)` pair; thus, you should structure your query to match this format.

In [12]:

Student's answer (Top)

```
# Define `query` with your query:
### BEGIN SOLUTION
query = '''
    SELECT Students.name, Takes.grade
        FROM Students, Takes
        WHERE Students.gtid=Takes.gtid AND Takes.course = 'CSE 6040'
'''
### END SOLUTION

c.execute(query)
results1 = c.fetchall()
results1
```

Out[12]:

```
[('Vuduc', 4.0), ('Sokol', 4.0), ('Chau', 4.0)]
```

In [13]: Grade cell: `join1__test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `join1__test`

print ("Your results:", results1)

assert type(results1) is list
assert len(results1) == 3, "Your query produced {} results instead of {}.".format(len(r
esults1), 3)

assert set(results1) == {('Vuduc', 4.0), ('Sokol', 4.0), ('Chau', 4.0)}

print("\n(Passed.)")
```

```
Your results: [('Vuduc', 4.0), ('Sokol', 4.0), ('Chau', 4.0)]

(Passed.)
```

For contrast, let's do a quick exercise that executes a left join (http://www.sqlitetutorial.net/sqlite-left-join/).

Exercise 3 (2 points). Execute a left join that uses `Students` as the left table, `Takes` as the right table, and selects a student's name and course grade. Write your query as a string variable named `query`, which the subsequent code will execute.

In [14]: Student's answer (Top)

```
# Define `query` string here:
### BEGIN SOLUTION
query = '''
    SELECT Students.name, Takes.grade
        FROM Students LEFT JOIN Takes ON
         Students.gtid=Takes.gtid
'''
### END SOLUTION

# Executes your `query` string:
c.execute(query)
matches = c.fetchall()
for i, match in enumerate(matches):
    print(i, "->", match)
```

```
0 -> ('Vuduc', 1.0)
1 -> ('Vuduc', 3.0)
2 -> ('Vuduc', 4.0)
3 -> ('Chau', 2.0)
4 -> ('Chau', 3.0)
5 -> ('Chau', 4.0)
6 -> ('Bader', None)
7 -> ('Sokol', 4.0)
8 -> ('Sokol', 4.0)
9 -> ('Rozga', None)
10 -> ('Zha', None)
11 -> ('Park', None)
12 -> ('Vetter', None)
13 -> ('Brown', None)
```

In [15]: Grade cell: `left_join_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `left_join_test`

assert set(matches) == {('Vuduc', 4.0), ('Chau', 2.0), ('Park', None), ('Vuduc', 1.0),
('Chau', 3.0), ('Zha', None), ('Brown', None), ('Vetter', None), ('Vuduc', 3.0), ('Bade
r', None), ('Rozga', None), ('Chau', 4.0), ('Sokol', 4.0)}
print("\n(Passed!)")
```

```
(Passed!)
```

Aggregations

Another common style of query is an aggregation, which is a summary of information across multiple records, rather than the raw records themselves.

For instance, suppose we want to compute the GPA for each unique GT ID from the `Takes` table. Here is a query that does it:

```
In [16]: query = '''
             select gtid, avg(grade)
                 from Takes
                 group by gtid
         '''

         for match in c.execute(query):
             print(match)
```

```
(123, 2.6666666666666665)
(456, 3.0)
(991, 4.0)
```

Some other useful SQL aggregators include `min`, `max`, `sum`, and `count`.

Cleanup

As one final bit of information, it's good practice to shutdown the cursor and connection, the same way you close files.

part1 (Score: 15.0 / 15.0)

1. Test cell (Score: 0.0 / 0.0)
2. Test cell (Score: 2.0 / 2.0)
3. Test cell (Score: 2.0 / 2.0)
4. Test cell (Score: 1.0 / 1.0)
5. Test cell (Score: 3.0 / 3.0)
6. Test cell (Score: 2.0 / 2.0)
7. Test cell (Score: 3.0 / 3.0)
8. Test cell (Score: 2.0 / 2.0)
9. Coding free-response (Score: 0.0 / 0.0)

Important note! Before you turn in this lab notebook, make sure everything runs as expected:

- First, **restart the kernel** -- in the menubar, select Kernel → Restart.
- Then **run all cells** -- in the menubar, select Cell → Run All.

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE."

Lesson 2: NYC 311 calls

This notebook derives from a demo by the makers of plot.ly (https://plot.ly/ipython-notebooks/big-data-analytics-with-pandas-and-sqlite/). We've adapted it to use Bokeh (and HoloViews) (http://bokeh.pydata.org/en/latest/).

You will start with a large database of complaints filed by residents of New York City since 2010 via 311 calls. The full dataset is available at the NYC open data portal (https://nycopendata.socrata.com/data). At about 6 GB and 10 million complaints, you can infer that a) you might not want to read it all into memory at once, and b) NYC residents have a lot to complain about. (Maybe only conclusion "a" is valid.) The notebook then combines the use of `sqlite`, `pandas`, and `bokeh`.

Module setup

Before diving in, run the following cells to preload some functions you'll need later.

```
In [1]: from IPython.display import display
        import pandas as pd
```

We'll also need some functionality from earlier notebooks.

In [2]:

```
def canonicalize_tibble(X):
    var_names = sorted(X.columns)
    Y = X[var_names].copy()
    Y.sort_values(by=var_names, inplace=True)
    Y.reset_index(drop=True, inplace=True)
    return Y

def tibbles_are_equivalent (A, B):
    A_canonical = canonicalize_tibble(A)
    B_canonical = canonicalize_tibble(B)
    cmp = A_canonical.eq(B_canonical)
    return cmp.all().all()

def cast(df, key, value, join_how='outer'):
    """Casts the input data frame into a tibble,
    given the key column and value column.
    """
    assert type(df) is pd.DataFrame
    assert key in df.columns and value in df.columns
    assert join_how in ['outer', 'inner']

    fixed_vars = df.columns.difference([key, value])
    tibble = pd.DataFrame(columns=fixed_vars) # empty frame
    new_vars = df[key].unique()
    for v in new_vars:
        df_v = df[df[key] == v]
        del df_v[key]
        df_v = df_v.rename(columns={value: v})
        tibble = tibble.merge(df_v,
                              on=list(fixed_vars),
                              how=join_how)
    return tibble
```

Lastly, some of the test cells will need some auxiliary files, which the following code cell will check for and, if they are missing, download.

In [3]:

```
import requests
import os
import hashlib
import io

def download(file, url_suffix=None, checksum=None):
    if url_suffix is None:
        url_suffix = file

    if os.path.exists('.voc'):
        url = 'https://cse6040.gatech.edu/datasets/{}'.format(url_suffix)
    else:
        url = 'https://github.com/cse6040/labs-fa17/raw/master/{}'.format(url_suffix)
    if os.path.exists(file):
        print("[{}]\n==> '{}' is already available.".format(url, file))
    else:
        print("[{}] Downloading...".format(url))
        r = requests.get(url)
        with open(file, 'w', encoding=r.encoding) as f:
            f.write(r.text)

    if checksum is not None:
        with io.open(file, 'r', encoding='utf-8', errors='replace') as f:
            body = f.read()
            body_checksum = hashlib.md5(body.encode('utf-8')).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".forma
t(file, body_checksum, checksum)
            print("==> Checksum test passes: {}".format(checksum))

    print("==> '{}' is ready!\n".format(file))

auxfiles = {'df_complaints_by_city_soln.csv': '2a82e5856d5a267db9aafc26f16c3ae1',
            'df_complaints_by_hour_soln.csv': 'f06fcd917876d51ad52ddc13b2fee69e',
            'df_noisy_by_hour_soln.csv': '30f3fa7c753d4d3f4b3edfa1f6d05bcc',
            'df_plot_stacked_fraction_soln.csv': '2ca04a3eb24ccc37ddd0f8f5917fb27a'}

for filename, checksum in auxfiles.items():
    download(filename, url_suffix='{}/{}'.format('lab9-sql', filename), checksum=checksum
    download(filename, url_suffix='{}/{}'.format('lab9-sql', filename), checksum=checksum
)

print("(Auxiliary files appear to be ready.)")
```

```
[https://github.com/cse6040/labs-fa17/raw/master/lab9-sql/df_noisy_by_hour_soln.csv]
==> 'df_noisy_by_hour_soln.csv' is already available.
==> Checksum test passes: 30f3fa7c753d4d3f4b3edfa1f6d05bcc
==> 'df_noisy_by_hour_soln.csv' is ready!

[https://github.com/cse6040/labs-fa17/raw/master/lab9-sql/df_complaints_by_city_soln.csv]
==> 'df_complaints_by_city_soln.csv' is already available.
==> Checksum test passes: 2a82e5856d5a267db9aafc26f16c3ae1
==> 'df_complaints_by_city_soln.csv' is ready!

[https://github.com/cse6040/labs-fa17/raw/master/lab9-sql/df_complaints_by_hour_soln.csv]
==> 'df_complaints_by_hour_soln.csv' is already available.
==> Checksum test passes: f06fcd917876d51ad52ddc13b2fee69e
==> 'df_complaints_by_hour_soln.csv' is ready!

[https://github.com/cse6040/labs-fa17/raw/master/lab9-sql/df_plot_stacked_fraction_soln.c
sv]
==> 'df_plot_stacked_fraction_soln.csv' is already available.
==> Checksum test passes: 2ca04a3eb24ccc37ddd0f8f5917fb27a
==> 'df_plot_stacked_fraction_soln.csv' is ready!

(Auxiliary files appear to be ready.)
```

Viz setup

This notebook includes some simple visualizations. This section just ensures you have the right software setup to follow along.

In [4]:

```
# Build a Pandas data frame
names = ['Bob','Jessica','Mary','John','Mel']
births = [968, 155, 77, 578, 973]
name_birth_pairs = list(zip(names, births))
baby_names = pd.DataFrame(data=name_birth_pairs, columns=['Names', 'Births'])
display(baby_names)
```

	Names	Births
0	Bob	968
1	Jessica	155
2	Mary	77
3	John	578
4	Mel	973

In [5]:

```
import holoviews as hv # Replacement for bokeh.charts / bkcharts
hv.extension('bokeh')
from holoviews import Bars
```

In [6]:

```
%%opts Bars [width=640 height=320]
Bars(baby_names, kdims=['Names'], vdims=['Births'], color='Names')
```

Out[6]:

In addition to the HoloViews interface (above), some of the visualizations will use the Bokeh mid-level interface.

In [7]:

```
from bokeh.io import show, output_notebook
output_notebook()
```

(https://bokeh.pydata.org) Loading BokehJS ...

In [8]:

```
# Adapted from: https://bokeh.pydata.org/en/latest/docs/user_guide/categorical.html#userg
uide-categorical
from bokeh.models import ColumnDataSource
from bokeh.plotting import figure
from bokeh.core.properties import value
```

```
def make_stacked_bar(df, x_var, bar_vars, kwargs_figure={}):
    assert type(x_var) is str, "x-variable should be a string but isn't."
    assert all([b in df.columns for b in bar_vars]), "Data frame is missing one or more c
olumns: {}".format(bar_vars)

    from bokeh.palettes import brewer
    assert len(bar_vars) in brewer['Dark2'], "Not enough colors."

    x = list(df[x_var])
    colors = brewer['Dark2'][len(bar_vars)]
    legend = [value(b) for b in bar_vars]
    source = ColumnDataSource(data=df)

    p = figure(x_range=x, **kwargs_figure)
    p.vbar_stack(bar_vars, x=x_var, width=0.9,
             fill_color=colors, line_color=None,
             legend=legend,
             source=source)
    return p
```

Data setup

You'll also need the NYC 311 calls dataset. What we've provided is actually a small subset (about 250+ MiB) of the full data as of 2015.

In [9]:

```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

if on_vocareum():
    DB_FILENAME = None # TBD
else:
    DB_FILENAME = 'NYC-311-2M.db'

if not os.path.exists(DB_FILENAME):
    url = 'https://onedrive.live.com/download?cid=FD520DDC6BE92730&resid=FD520DDC6BE9273
0%21616&authkey=AEeP_4E1uh-vyDE'
    print("Downloading: {} ...".format(url))
    r = requests.get(url)
    with open(file, 'w', encoding=r.encoding) as f:
        f.write(r.text)

DB_CHECKSUM = 'f48eba2fb06e8ece7479461ea8c6dee9'
with io.open(DB_FILENAME, 'rb') as f:
    body = f.read()
    body_checksum = hashlib.md5(body).hexdigest()
    assert body_checksum == DB_CHECKSUM, \
        "Database file '{}' has an incorrect checksum: '{}' instead of '{}'".format(DB_FI
LENAME,
                                                                                  body_
checksum,
                                                                                  DB_CH
ECKSUM)

print("'{}' is ready!".format(DB_FILENAME))
print("\n(All data appears to be ready.)")
```

```
'NYC-311-2M.db' is ready!

(All data appears to be ready.)
```

Connecting. Let's open up a connection to this dataset.

In [10]:

```
# Connect
import sqlite3 as db
disk_engine = db.connect(DB_FILENAME)
```

Preview the data. This sample database has just a single table, named `data`. Let's query it and see how long it takes to read. To carry out the query, we will use the SQL reader built into `pandas`.

In [11]:

```
import time
```

```
In [11]: import time

         print ("Reading ...")
         start_time = time.time ()

         # Perform SQL query through the disk_engine connection.
         # The return value is a pandas data frame.
         df = pd.read_sql_query ('select * from data', disk_engine)

         elapsed_time = time.time () - start_time
         print ("==> Took %g seconds." % elapsed_time)

         # Dump the first few rows
         df.head()
```

```
Reading ...

==> Took 29.6305 seconds.
```

Out[11]:

	index	CreatedDate	ClosedDate	Agency	ComplaintType	Descriptor	City
0	1	2015-09-15 02:14:04.000000	None	NYPD	Illegal Parking	Blocked Hydrant	None
1	2	2015-09-15 02:12:49.000000	None	NYPD	Noise - Street/Sidewalk	Loud Talking	NEW YORK
2	3	2015-09-15 02:11:19.000000	None	NYPD	Noise - Street/Sidewalk	Loud Talking	NEW YORK
3	4	2015-09-15 02:09:46.000000	None	NYPD	Noise - Commercial	Loud Talking	BRONX
4	5	2015-09-15 02:08:01.000000	2015-09-15 02:08:18.000000	DHS	Homeless Person Assistance	Status Call	NEW YORK

Partial queries: `LIMIT` clause. The preceding command was overkill for what we wanted, which was just to preview the table. Instead, we could have used the `LIMIT` option to ask for just a few results.

```
In [12]: query = '''
           select *
             from data
             limit 5
         '''
         start_time = time.time ()
         df = pd.read_sql_query (query, disk_engine)
         elapsed_time = time.time () - start_time
         print ("==> LIMIT version took %g seconds." % elapsed_time)

         df
```

```
==> LIMIT version took 0.00597191 seconds.
```

Out[12]:

	index	CreatedDate	ClosedDate	Agency	ComplaintType	Descriptor	City
0	1	2015-09-15 02:14:04.000000	None	NYPD	Illegal Parking	Blocked Hydrant	None
1	2	2015-09-15 02:12:49.000000	None	NYPD	Noise - Street/Sidewalk	Loud Talking	NEW YORK
2	3	2015-09-15 02:11:19.000000	None	NYPD	Noise - Street/Sidewalk	Loud Talking	NEW YORK
3	4	2015-09-15 02:09:46.000000	None	NYPD	Noise - Commercial	Loud Talking	BRONX
4	5	2015-09-15 02:08:01.000000	2015-09-15 02:08:18.000000	DHS	Homeless Person Assistance	Status Call	NEW YORK

Grouping Information: GROUP BY operator. The GROUP BY operator lets you group information using a particular column or multiple columns of the table. The output generated is more of a pivot table.

```
In [13]: query = '''
           select ComplaintType, Descriptor, Agency
             from data
             GROUP BY ComplaintType
             limit 10
```

```
'''

df = pd.read_sql_query(query, disk_engine)
df.head()
```

Out[13]:

	ComplaintType	Descriptor	Agency
0	AGENCY	HOUSING QUALITY STANDARDS	HPD
1	APPLIANCE	REFRIGERATOR	HPD
2	Adopt-A-Basket	10A Adopt-A-Basket	DSNY
3	Agency Issues	Call Center Compliment	3-1-1
4	Air Quality	Air: Odor/Fumes, Vehicle Idling (AD3)	DEP

In [14]:

```
query = '''
  select ComplaintType, Descriptor, Agency
    from data
    limit 10
'''

df = pd.read_sql_query(query, disk_engine)
df.head()
```

Out[14]:

	ComplaintType	Descriptor	Agency
0	Illegal Parking	Blocked Hydrant	NYPD
1	Noise - Street/Sidewalk	Loud Talking	NYPD
2	Noise - Street/Sidewalk	Loud Talking	NYPD
3	Noise - Commercial	Loud Talking	NYPD
4	Homeless Person Assistance	Status Call	DHS

Set membership: `IN` operator. Another common idiom is to ask for rows whose attributes fall within a set, for which you can use the `IN` operator.

In [15]:

```
query = '''
  select ComplaintType, Descriptor, Agency
    from data
    where Agency IN ("NYPD", "DOB")
    limit 10
'''

df = pd.read_sql_query (query, disk_engine)
df.head ()
```

Out[15]:

	ComplaintType	Descriptor	Agency
0	Illegal Parking	Blocked Hydrant	NYPD
1	Noise - Street/Sidewalk	Loud Talking	NYPD
2	Noise - Street/Sidewalk	Loud Talking	NYPD
3	Noise - Commercial	Loud Talking	NYPD
4	Blocked Driveway	Partial Access	NYPD

Finding unique values: `DISTINCT` qualifier. Yet another common idiom is to ask for the unique values of some attribute, for which you can use the `DISTINCT` qualifier.

In [16]:

```
query = 'select DISTINCT City FROM data'
df = pd.read_sql_query(query, disk_engine)

print("Found {} unique cities. The first few are:".format(len(df)))
df.head()
```

```
Found 547 unique cities. The first few are:
```

Out[16]:

	City
0	None
1	NEW YORK

2	BRONX
3	STATEN ISLAND
4	ELMHURST

Renaming columns: `AS` operator. Sometimes you might want to rename a result column. For instance, the following query counts the number of complaints by "Agency," using the `COUNT(*)` function and `GROUP BY` clause, which we discussed in an earlier lab. If you wish to refer to the counts column of the resulting data frame, you can give it a more "friendly" name using the `AS` operator.

In [17]:

```
query = '''
  select Agency, count(*) as NumComplaints
    from data
    group by Agency
'''
df = pd.read_sql_query(query, disk_engine)
df.head()
```

Out[17]:

	Agency	NumComplaints
0	3-1-1	1289
1	ACS	3
2	AJC	6
3	CAU	1
4	CCRB	1

Ordering results: `ORDER` clause. You can also order the results. For instance, suppose we want to execute the previous query by number of complaints.

In [18]:

```
query = '''
  select Agency, count(*) as NumComplaints
    from data
    group by Agency
    order by NumComplaints
'''
df = pd.read_sql_query(query, disk_engine)
df.tail()
```

Out[18]:

	Agency	NumComplaints
45	DSNY	152004
46	DEP	181121
47	DOT	322969
48	NYPD	340694
49	HPD	640096

Note that the above example prints the bottom (tail) of the data frame. You could have also asked for the query results in reverse (descending) order, by prefixing the `ORDER BY` attribute with a `-` (minus) symbol.

In [19]:

```
query = '''
  select Agency, count(*) as NumComplaints
    from data
    group by Agency
    order by -NumComplaints
'''
df = pd.read_sql_query(query, disk_engine)
df.head()
```

Out[19]:

	Agency	NumComplaints
0	HPD	640096
1	NYPD	340694
2	DOT	322969
3	DEP	181121
4	DSNY	152004

And of course we can plot all of this data!

Exercise 0 (ungraded). Run the following code cell, which will create an interactive bar chart from the data in the previous query.

In [20]: Grade cell: `exercise_0` Score: 0.0 / 0.0 (Top)

```
%%opts Bars [width=640 height=320]
chart = Bars(df[:20], kdims=['Agency'], vdims=[('NumComplaints', '# complaints')],
             label='Top 20 agencies by number of complaints')
chart(plot={'xrotation': 25})
```

Out[20]:

Exercise 1 (2 points). Create a string, `query`, containing an SQL query that will return the number of complaints by type. The columns should be named `type` and `freq`, and the results should be sorted in descending order by `freq`.

> What is the most common type of complaint? What, if anything, does it tell you about NYC?

In [21]: Student's answer (Top)

```
del query

# Define a variable named `query` containing your solution
### BEGIN SOLUTION
query = '''
  select ComplaintType as type, count(*) as freq
    from data
    group by type
    order by -freq
'''
### END SOLUTION

# Runs your `query`:
df_complaint_freq = pd.read_sql_query(query, disk_engine)
df_complaint_freq.head()
```

Out[21]:

	type	freq
0	HEAT/HOT WATER	241430
1	Street Condition	124347
2	Street Light Condition	98577
3	Blocked Driveway	95080
4	Illegal Parking	83961

In [22]: Grade cell: `complaints_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `complaints_test`

print("Top 10 complaints:")
display(df_complaint_freq.head(10))

assert set(df_complaint_freq.columns) == {'type', 'freq'}, "Output columns should be na
med 'type' and 'freq', not {}".format(set(df_complaint_freq.columns))

soln = ['HEAT/HOT WATER', 'Street Condition', 'Street Light Condition', 'Blocked Drivew
ay', 'Illegal Parking', 'UNSANITARY CONDITION', 'PAINT/PLASTER', 'Water System', 'PLUMB
ING', 'Noise', 'Noise - Street/Sidewalk', 'Traffic Signal Condition', 'Noise - Commerci
al', 'DOOR/WINDOW', 'WATER LEAK', 'Dirty Conditions', 'Sewer', 'Sanitation Condition',
'DOF Literature Request', 'ELECTRIC', 'Rodent', 'FLOORING/STAIRS', 'General Constructio
n/Plumbing', 'Building/Use', 'Broken Muni Meter', 'GENERAL', 'Missed Collection (All Ma
terials)', 'Benefit Card Replacement', 'Derelict Vehicle', 'Noise - Vehicle', 'Damaged
 Tree', 'Consumer Complaint', 'Derelict Vehicles', 'Taxi Complaint', 'Overgrown Tree/Br
anches', 'Graffiti', 'Snow', 'Opinion for the Mayor', 'APPLIANCE', 'Maintenance or Faci
lity', 'Animal Abuse', 'Dead Tree', 'HPD Literature Request', 'Root/Sewer/Sidewalk Cond
ition', 'SAFETY', 'Elevator', 'Food Establishment', 'SCRIE', 'Air Quality', 'Agency Iss
ues', 'Construction', 'Highway Condition', 'Other Enforcement', 'Water Conservation',
'Sidewalk Condition', 'Indoor Air Quality', 'Street Sign - Damaged', 'Traffic', 'Plumbi
'Sidewalk Condition', 'Indoor Air Quality', 'Street Sign - Damaged', 'Traffic', 'Plumbi
ng', 'Fire Safety Director - F58', 'Homeless Person Assistance', 'Homeless Encampment',
 'Special Enforcement', 'Street Sign - Missing', 'Noise - Park', 'Vending', 'For Hire V
ehicle Complaint', 'Food Poisoning', 'Special Projects Inspection Team (SPIT)', 'Hazard
ous Materials', 'Electrical', 'DOT Literature Request', 'Litter Basket / Request', 'Tax
i Report', 'Illegal Tree Damage', 'DOF Property - Reduction Issue', 'Unsanitary Animal
 Pvt Property', 'Asbestos', 'Lead', 'Vacant Lot', 'DCA / DOH New License Application Re
quest', 'Street Sign - Dangling', 'Smoking', 'Violation of Park Rules', 'OUTSIDE BUILDI
NG', 'Animal in a Park', 'Noise - Helicopter', 'School Maintenance', 'DPR Internal', 'B
oilers', 'Industrial Waste', 'Sweeping/Missed', 'Overflowing Litter Baskets', 'Non-Resi
dential Heat', 'Curb Condition', 'Drinking', 'Standing Water', 'Indoor Sewage', 'Water
 Quality', 'EAP Inspection - F59', 'Derelict Bicycle', 'Noise - House of Worship', 'DCA
 Literature Request', 'Recycling Enforcement', 'ELEVATOR', 'DOF Parking - Tax Exemptio
n', 'Broken Parking Meter', 'Request for Information', 'Taxi Compliment', 'Unleashed Do
g', 'Urinating in Public', 'Unsanitary Pigeon Condition', 'Investigations and Disciplin
e (IAD)', 'Bridge Condition', 'Ferry Inquiry', 'Bike/Roller/Skate Chronic', 'Public Pay
phone Complaint', 'Vector', 'BEST/Site Safety', 'Sweeping/Inadequate', 'Disorderly Yout
h', 'Found Property', 'Mold', 'Senior Center Complaint', 'Fire Alarm - Reinspection',
'For Hire Vehicle Report', 'City Vehicle Placard Complaint', 'Cranes and Derricks', 'Fe
rry Complaint', 'Illegal Animal Kept as Pet', 'Posting Advertisement', 'Harboring Bees/
Wasps', 'Panhandling', 'Scaffold Safety', 'OEM Literature Request', 'Plant', 'Bus Stop
 Shelter Placement', 'Collection Truck Noise', 'Beach/Pool/Sauna Complaint', 'Complain
t', 'Compliment', 'Illegal Fireworks', 'Fire Alarm - Modification', 'DEP Literature Req
uest', 'Drinking Water', 'Fire Alarm - New System', 'Poison Ivy', 'Bike Rack Condition'
, 'Emergency Response Team (ERT)', 'Municipal Parking Facility', 'Tattooing', 'Unsanita
ry Animal Facility', 'Animal Facility - No Permit', 'Miscellaneous Categories', 'Misc.
 Comments', 'Literature Request', 'Special Natural Area District (SNAD)', 'Highway Sign
 - Damaged', 'Public Toilet', 'Adopt-A-Basket', 'Ferry Permit', 'Invitation', 'Window G
uard', 'Parking Card', 'Illegal Animal Sold', 'Stalled Sites', 'Open Flame Permit', 'Ov
erflowing Recycling Baskets', 'Highway Sign - Missing', 'Public Assembly', 'DPR Literat
ure Request', 'Fire Alarm - Addition', 'Lifeguard', 'Transportation Provider Complaint'
, 'DFTA Literature Request', 'Bottled Water', 'Highway Sign - Dangling', 'DHS Income Sa
vings Requirement', 'Legal Services Provider Complaint', 'Foam Ban Enforcement', 'Tunne
l Condition', 'Calorie Labeling', 'Fire Alarm - Replacement', 'X-Ray Machine/Equipment'
, 'Sprinkler - Mechanical', 'Hazmat Storage/Use', 'Tanning', 'Radioactive Material', 'R
angehood', 'SRDE', 'Squeegee', 'Building Condition', 'SG-98', 'Standpipe - Mechanical',
 'AGENCY', 'Forensic Engineering', 'Public Assembly - Temporary', 'VACANT APARTMENT',
'Laboratory', 'SG-99']
assert all(soln == df_complaint_freq['type'])

print("\n(Passed.)")
```

```
Top 10 complaints:
```

	type	freq
0	HEAT/HOT WATER	241430
1	Street Condition	124347
2	Street Light Condition	98577
3	Blocked Driveway	95080
4	Illegal Parking	83961
5	UNSANITARY CONDITION	81394
6	PAINT/PLASTER	69929
7	Water System	69209
8	PLUMBING	55445
9	Noise	54165

```
(Passed.)
```

Let's also visualize the result, as a bar chart showing complaint types on the x-axis and the number of complaints on the y-axis.

In [23]:

```
%%opts Bars [width=800 height=320]
chart = Bars(df_complaint_freq[:25], kdims=['type'], vdims=[('freq', '# complaints')],
             label='Top 25 complaints by type')
chart(plot={'xrotation': 25})
```

Out[23]:

Lesson 3: More SQL stuff

Simple substring matching: the `LIKE` operator. Suppose we just want to look at the counts for all complaints that have the word `noise` in them. You can use the `LIKE` operator combined with the string wildcard, `%`, to look for case-insensitive substring matches.

In [24]:
```
query = '''
  select ComplaintType as type, count(*) as freq
    from data
    where ComplaintType like '%noise%'
    group by type
    order by -freq
'''

df_noisy = pd.read_sql_query(query, disk_engine)
df_noisy
```

Out[24]:

	type	freq
0	Noise	54165
1	Noise - Street/Sidewalk	48436
2	Noise - Commercial	42422
3	Noise - Vehicle	18370
4	Noise - Park	4020
5	Noise - Helicopter	1715
6	Noise - House of Worship	1143
7	Collection Truck Noise	184

Exercise 2 (2 points). Create a string variable, `query`, that contains an SQL query that will return the top 10 cities with the largest number of complaints, in descending order. It should return a table with two columns, one named `name` holding the name of the city, and one named `freq` holding the number of complaints by that city.

In [25]:

Student's answer (Top)

```
del query

# Define your `query`, here:
### BEGIN SOLUTION
query = '''
  select City as name, count(*) as freq
    from data
    group by name
    order by -freq limit 10
'''
### END SOLUTION

# Runs your `query`:
df_whiny_cities = pd.read_sql_query(query, disk_engine)
df_whiny_cities
```

Out[25]:

	name	freq
0	BROOKLYN	579363
1	NEW YORK	385655
2	BRONX	342533
3	None	168692
4	STATEN ISLAND	92509
5	Jamaica	30435
6	Flushing	20708
7	Astoria	18068
8	JAMAICA	16248
9	FLUSHING	14796

In [26]:

Grade cell: `whiny_cities__test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `whiny_cities__test`

assert df_whiny_cities['name'][0] == 'BROOKLYN'
assert df_whiny_cities['name'][1] == 'NEW YORK'
assert df_whiny_cities['name'][2] == 'BRONX'
assert df_whiny_cities['name'][3] is None
assert df_whiny_cities['name'][4] == 'STATEN ISLAND'

print ("\n(Passed partial test.)")
```

```
(Passed partial test.)
```

You should notice two odd bits: cities are treated in a case-sensitive manner and `None` appears as a city. (Presumably this setting occurs when a complaint is non-localized or the city is not otherwise specified.)

Case-insensitive grouping: `COLLATE NOCASE`. One way to carry out the preceding query in a case-insensitive way is to add a `COLLATE NOCASE` qualifier to the `GROUP BY` clause.

Let's filter out the 'None' cases as well, while we are at it.

In [27]:

```
query = '''
  SELECT City as name, COUNT(*) AS freq
    FROM data
    WHERE name <> 'None'
    GROUP BY name COLLATE NOCASE
    ORDER BY -freq
    LIMIT 10
'''
df_whiny_cities2 = pd.read_sql_query(query, disk_engine)
df_whiny_cities2
```

Out[27]:

	name	freq
0	BROOKLYN	579363
1	NEW YORK	385655
2	BRONX	342533
3	STATEN ISLAND	92509
4	Jamaica	46683
5	Flushing	35504
6	ASTORIA	31873
7	Ridgewood	21618
8	Woodside	15932
9	Corona	15740

Brooklynites are complainers, evidently.

Lastly, for later use, let's save the names of just the top seven (7) cities by numbers of complaints.

In [28]:

```
TOP_CITIES = list(df_whiny_cities2.head(7)['name'])
TOP_CITIES
```

Out[28]:

```
['BROOKLYN',
 'NEW YORK',
 'BRONX',
 'STATEN ISLAND',
 'Jamaica',
 'Flushing',
 'ASTORIA']
```

Exercise 3 (1 point). Implement a function that takes a list of strings, `str_list`, and returns a single string consisting of each value, `str_list[i]`, enclosed by double-quotes and separated by a comma-space delimiters. For example, if

```
assert str_list == ['a', 'b', 'c', 'd']
```

then

```
assert strs_to_args(str_list) == '"a", "b", "c", "d"'
```

In [29]: Student's answer (Top)

```
def strs_to_args(str_list):
    assert type (str_list) is list
    assert all ([type (s) is str for s in str_list])

    ### BEGIN SOLUTION
    quoted = ['"{}"'.format(s) for s in str_list]
    return ', '.join(quoted)
    ### END SOLUTION
```

In [30]: Grade cell: `strs_to_args__test` Score: 1.0 / 1.0 (Top)

```
# Test cell: `strs_to_args__test`

print ("Your solution, applied to TOP_CITIES:", strs_to_args(TOP_CITIES))

TOP_CITIES_as_args = strs_to_args(TOP_CITIES)
assert TOP_CITIES_as_args.lower() == \
       '"BROOKLYN", "NEW YORK", "BRONX", "STATEN ISLAND", "Jamaica", "Flushing", "ASTOR
IA"'.lower ()

print ("\n(Passed.)")
```

```
Your solution, applied to TOP_CITIES: "BROOKLYN", "NEW YORK", "BRONX", "STATEN ISLAND",
"Jamaica", "Flushing", "ASTORIA"

(Passed.)
```

Exercise 4 (3 points). Suppose we want to look at the number of complaints by type and by city. Execute an SQL query to produce a tibble named `df_complaints_by_city` with the variables {`complaint_type`, `city_name`, `complaint_count`}.

In [31]: Student's answer (Top)

```
### BEGIN SOLUTION
query = """
select complainttype as complaint_type, city as city_name, count(*) as complaint_count
from data
where city in ({})
group by City, complainttype
order by City, complaint_count
""".format(strs_to_args(TOP_CITIES))

# Previews the results of your query:
df_complaints_by_city = pd.read_sql_query(query, disk_engine)
display(df_complaints_by_city.head(10))
### END SOLUTION
```

	complaint_type	city_name	complaint_count
0	Bottled Water	ASTORIA	1
1	Bridge Condition	ASTORIA	1
2	City Vehicle Placard Complaint	ASTORIA	1
3	Open Flame Permit	ASTORIA	1
4	Panhandling	ASTORIA	1
5	Unsanitary Pigeon Condition	ASTORIA	1
6	Vector	ASTORIA	1
7	Window Guard	ASTORIA	1
8	Beach/Pool/Sauna Complaint	ASTORIA	2
9	Drinking Water	ASTORIA	2

In [32]: Grade cell: `df_complaints_by_city_test` Score: 3.0 / 3.0 (Top)

In [32]: Grade cell: df_complaints_by_city__test Score: 3.0 / 3.0 (Top)

```
# Test cell: `df_complaints_by_city__test`

print("Reading instructor's solution...")
df_complaints_by_city_soln = pd.read_csv('df_complaints_by_city_soln.csv')

print("Checking...")
assert tibbles_are_equivalent(df_complaints_by_city,
                              df_complaints_by_city_soln)

print("\n(Passed.)")
del df_complaints_by_city_soln
```

```
Reading instructor's solution...
Checking...

(Passed.)
```

Let's use HoloViews+Bokeh to visualize the results as a stacked bar chart.

In [33]:

```
# Let's consider only the top 25 complaints (by total)
top_complaints = df_complaint_freq[:25]
print("Top complaints:")
display(top_complaints)
```

```
Top complaints:
```

	type	freq
0	HEAT/HOT WATER	241430
1	Street Condition	124347
2	Street Light Condition	98577
3	Blocked Driveway	95080
4	Illegal Parking	83961
5	UNSANITARY CONDITION	81394
6	PAINT/PLASTER	69929
7	Water System	69209
8	PLUMBING	55445
9	Noise	54165
10	Noise - Street/Sidewalk	48436
11	Traffic Signal Condition	44229
12	Noise - Commercial	42422
13	DOOR/WINDOW	39695
14	WATER LEAK	36149
15	Dirty Conditions	35122
16	Sewer	33628
17	Sanitation Condition	31260
18	DOF Literature Request	30326
19	ELECTRIC	30248
20	Rodent	28454
21	FLOORING/STAIRS	27007
22	General Construction/Plumbing	26861
23	Building/Use	25807
24	Broken Muni Meter	25428

In [34]:

```
# Plot subset of data corresponding to the top complaints
df_plot = top_complaints.merge(df_complaints_by_city,
                               left_on=['type'],
                               right_on=['complaint_type'],
                               how='left')
```

```
                                        how='left')
df_plot.dropna(inplace=True)
print("Data to plot (first few rows):")
display(df_plot.head())
print("...")
```

```
Data to plot (first few rows):
```

	type	freq	complaint_type	city_name	complaint_count
0	HEAT/HOT WATER	241430	HEAT/HOT WATER	BRONX	79690
1	HEAT/HOT WATER	241430	HEAT/HOT WATER	BROOKLYN	72410
2	HEAT/HOT WATER	241430	HEAT/HOT WATER	Flushing	2741
3	HEAT/HOT WATER	241430	HEAT/HOT WATER	Jamaica	3376
4	HEAT/HOT WATER	241430	HEAT/HOT WATER	NEW YORK	55545

```
...
```

Let's visualize this as a stacked bar chart. To do so, we'll need to reshape the data frame so that the values to be stacked appear as columns. It's the perfect application for cast!

```
In [35]: df_plot_stacked = cast(df_plot, key='city_name', value='complaint_count')
         df_plot_stacked.fillna(0, inplace=True)
         display(df_plot_stacked)
```

	type	freq	complaint_type	BRONX	BROOKLYN	Flushing	Jamaica	NEW YORK	STAT ISLAI
0	HEAT/HOT WATER	241430	HEAT/HOT WATER	79690	72410	2741	3376	55545	2280
1	Street Condition	124347	Street Condition	12112	34426	2385	2329	19110	1688
2	Street Light Condition	98577	Street Light Condition	7924	19080	1764	2471	2099	6197
3	Blocked Driveway	95080	Blocked Driveway	15608	34351	0	0	2710	2702
4	Illegal Parking	83961	Illegal Parking	9186	30495	0	0	13317	5837
5	UNSANITARY CONDITION	81394	UNSANITARY CONDITION	23440	28668	694	1548	15560	1890
6	PAINT/PLASTER	69929	PAINT/PLASTER	22408	24240	688	1041	14379	981
7	Water System	69209	Water System	10965	19245	1394	2231	12482	6183
8	PLUMBING	55445	PLUMBING	16929	18919	427	1204	11529	1191
9	Noise	54165	Noise	2900	14362	765	465	25943	1532
10	Noise - Street/Sidewalk	48436	Noise - Street/Sidewalk	8719	12906	0	0	20775	911
11	Traffic Signal Condition	44229	Traffic Signal Condition	1737	12319	619	1033	1251	1613
12	Noise - Commercial	42422	Noise - Commercial	2703	13283	0	0	18090	763
13	DOOR/WINDOW	39695	DOOR/WINDOW	12270	14226	300	689	7665	753
14	WATER LEAK	36149	WATER LEAK	10942	13610	387	574	6856	582
15	Dirty Conditions	35122	Dirty Conditions	4918	11527	860	1153	5346	2878
16	Sewer	33628	Sewer	3454	9246	716	2288	3442	3753
17	Sanitation Condition	31260	Sanitation Condition	3534	11404	766	1304	3304	2768
18	ELECTRIC	30248	ELECTRIC	8601	11253	215	698	5071	657
19	Rodent	28454	Rodent	6576	9070	168	554	6801	1352
20	FLOORING/STAIRS	27007	FLOORING/STAIRS	8113	9617	163	436	5948	541
21	General Construction/Plumbing	26861	General Construction/Plumbing	2502	9470	563	639	7012	1351
22	Building/Use	25807	Building/Use	3026	7583	1110	1295	1828	1535
23	Broken Muni Meter	25428	Broken Muni Meter	2542	5211	0	0	10985	262

```
In [36]: # Some code to render a Bokeh stacked bar chart

         kwargs_figure = {'title': "Distribution of the top 25 complaints among top 7 cities with
```

```
 the most complaints",
                 'width': 800,
                 'height': 400,
                 'tools': "hover,crosshair,pan,box_zoom,wheel_zoom,save,reset,help"}
p = make_stacked_bar(df_plot_stacked, 'complaint_type', TOP_CITIES, kwargs_figure=kwargs_
figure)

p.xaxis.major_label_orientation = 0.66

from bokeh.models import HoverTool
hover_tool = p.select(dict(type=HoverTool))
hover_tool.tooltips = [("y", "$y{int}")]

show(p)
```

Exercise 5 (2 points). The preceding code created a dataframe, `df_plot_stacked`, which was then used to create the stacked bar chart shown above.

Suppose we want to create a different stacked bar plot that shows, for each complaint type t and city c, the fraction of all complaints of type t that occurred in city c. Store your result in a dataframe named `df_plot_stacked_fraction`. It should have the same columns as `df_plot_stacked`.

> The test cell will create the chart in addition to checking your result. Note that the normalized bars will not necessarily add up to 1; why not?
>
> Note: The normalized bars will not necessarily add up to 1. Why not?

In [37]: Student's answer (Top)

```
### BEGIN SOLUTION
df_plot_stacked_fraction = df_plot_stacked.copy()
for c in TOP_CITIES:
    df_plot_stacked_fraction[c] /= df_plot_stacked_fraction['freq']
### END SOLUTION

df_plot_stacked_fraction.head()
```

Out[37]:

	type	freq	complaint_type	BRONX	BROOKLYN	Flushing	Jamaica	NEW YORK	STATEN ISLAND	ASTORIA
0	HEAT/HOT WATER	241430	HEAT/HOT WATER	0.330075	0.299921	0.011353	0.013983	0.230067	0.009444	0.000000
1	Street Condition	124347	Street Condition	0.097405	0.276854	0.019180	0.018730	0.153683	0.135822	0.004335
2	Street Light Condition	98577	Street Light Condition	0.080384	0.193554	0.017895	0.025067	0.021293	0.062865	0.000000
3	Blocked Driveway	95080	Blocked Driveway	0.164156	0.361285	0.000000	0.000000	0.028502	0.028418	0.035107
4	Illegal Parking	83961	Illegal Parking	0.109408	0.363204	0.000000	0.000000	0.158609	0.069520	0.017329

In [38]: Grade cell: `norm_above_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `norm_above_test`

if False:
    df_plot_stacked_fraction.to_csv('df_plot_stacked_fraction_soln.csv', index=False)

p = make_stacked_bar(df_plot_stacked_fraction, 'complaint_type', TOP_CITIES, kwargs_fig
ure=kwargs_figure)

p.xaxis.major_label_orientation = 0.66

from bokeh.models import HoverTool
hover_tool = p.select(dict(type=HoverTool))
hover_tool.tooltips = [("y", "$y")]
```

```
show(p)

# Check numerical values
df_plot_stacked_fraction_soln = pd.read_csv('df_plot_stacked_fraction_soln.csv')
def merge_two_dicts(x, y):
    return {**x, **y}

def merge_many_dicts(dicts):
    x = {}
    for d in dicts:
        x = merge_two_dicts(x, d)
    return x

def tuple_sub(a, b):
    assert type(a) is tuple and type(b) is tuple and len(a) == len(b)
    return tuple(i - j for i, j in zip(a, b))

def all_tol(x, tol=1e-14):
    return all([abs(i) <= tol for i in x])

df_plot_stacked_fraction_soln = pd.read_csv('df_plot_stacked_fraction_soln.csv')
row_to_dict = lambda x: {x['complaint_type']: tuple(x[TOP_CITIES])}
your_soln = merge_many_dicts(df_plot_stacked_fraction.apply(row_to_dict, axis=1))
true_soln = merge_many_dicts(df_plot_stacked_fraction_soln.apply(row_to_dict, axis=1))
assert len(your_soln) == len(true_soln)
for key_true, value_true in true_soln.items():
    assert key_true in your_soln, "Your solution is missing the complaint type, '{}'".f
ormat(key_true)
    value_yours = your_soln[key_true]
    assert all_tol(tuple_sub(value_yours, value_true)), "Data for complaint '{}' of you
r solution differs from that of reference solution.".format(key_true)

print("\n(Passed!)")
```

```
(Passed!)
```

Dates and times in SQL

Recall that the input data had a column with timestamps corresponding to when someone submitted a complaint. Let's quickly summarize some of the features in SQL and Python for reasoning about these timestamps.

The `CreatedDate` column is actually a specially formatted date and time stamp, where you can query against by comparing to strings of the form, `YYYY-MM-DD hh:mm:ss`.

For example, let's look for all complaints on September 15, 2015.

In [39]:

```
query = '''
  select ComplaintType, CreatedDate, City
    from data
    where CreatedDate >= "2015-09-15 00:00:00.0"
      and CreatedDate < "2015-09-16 00:00:00.0"
    order by CreatedDate
'''
df = pd.read_sql_query (query, disk_engine)
df
```

Out[39]:

	ComplaintType	CreatedDate	City
0	Illegal Parking	2015-09-15 00:01:23.000000	None
1	Blocked Driveway	2015-09-15 00:02:29.000000	REGO PARK
2	Taxi Complaint	2015-09-15 00:02:34.000000	NEW YORK
3	Opinion for the Mayor	2015-09-15 00:03:07.000000	None
4	Opinion for the Mayor	2015-09-15 00:03:07.000000	None
5	Noise - Vehicle	2015-09-15 00:03:14.000000	BROOKLYN
6	Dirty Conditions	2015-09-15 00:04:00.000000	BROOKLYN
7	Noise - Commercial	2015-09-15 00:04:22.000000	NEW YORK
8	UNSANITARY CONDITION	2015-09-15 00:04:24.000000	NEW YORK
9	PAINT/PLASTER	2015-09-15 00:04:24.000000	NEW YORK

10	PLUMBING	2015-09-15 00:04:24.000000	BRONX
11	WATER LEAK	2015-09-15 00:04:24.000000	NEW YORK
12	Graffiti	2015-09-15 00:04:39.000000	BRONX
13	Graffiti	2015-09-15 00:05:32.000000	BRONX
14	Noise - Street/Sidewalk	2015-09-15 00:05:41.000000	BRONX
15	Noise - Commercial	2015-09-15 00:06:38.000000	NEW YORK
16	Graffiti	2015-09-15 00:06:46.000000	BRONX
17	Illegal Parking	2015-09-15 00:07:09.000000	ROSEDALE
18	Graffiti	2015-09-15 00:08:49.000000	BRONX
19	Graffiti	2015-09-15 00:10:13.000000	BRONX
20	Illegal Parking	2015-09-15 00:11:20.000000	QUEENS VILLAGE
21	Graffiti	2015-09-15 00:11:26.000000	BRONX
22	Noise - Street/Sidewalk	2015-09-15 00:11:27.000000	BROOKLYN
23	Animal in a Park	2015-09-15 00:11:36.000000	ELMHURST
24	Animal Abuse	2015-09-15 00:11:41.000000	BROOKLYN
25	Blocked Driveway	2015-09-15 00:12:33.000000	BROOKLYN
26	UNSANITARY CONDITION	2015-09-15 00:12:44.000000	BROOKLYN
27	Noise - Street/Sidewalk	2015-09-15 00:13:54.000000	BROOKLYN
28	Noise - Vehicle	2015-09-15 00:15:49.000000	NEW YORK
29	Rodent	2015-09-15 00:18:07.000000	NEW YORK
...	...	...	...
88	Unsanitary Animal Pvt Property	2015-09-15 01:24:45.000000	BROOKLYN
89	DOT Literature Request	2015-09-15 01:25:41.000000	None
90	Noise - Commercial	2015-09-15 01:26:03.000000	NEW YORK
91	Noise - Street/Sidewalk	2015-09-15 01:28:33.000000	NEW YORK
92	Noise - Street/Sidewalk	2015-09-15 01:30:09.000000	NEW YORK
93	Noise - Commercial	2015-09-15 01:31:26.000000	KEW GARDENS
94	Opinion for the Mayor	2015-09-15 01:31:55.000000	None
95	Noise - Commercial	2015-09-15 01:34:18.000000	ASTORIA
96	Noise - Commercial	2015-09-15 01:36:25.000000	NEW YORK
97	Street Condition	2015-09-15 01:37:35.000000	NEW YORK
98	Rodent	2015-09-15 01:37:42.000000	BROOKLYN
99	Noise - Commercial	2015-09-15 01:40:08.000000	BROOKLYN
100	Noise - Commercial	2015-09-15 01:45:15.000000	NEW YORK
101	Noise - Vehicle	2015-09-15 01:47:16.000000	BROOKLYN
102	Noise - Commercial	2015-09-15 01:47:56.000000	NEW YORK
103	Noise - Street/Sidewalk	2015-09-15 01:48:19.000000	BROOKLYN
104	Blocked Driveway	2015-09-15 01:51:04.000000	BRONX
105	Noise - Street/Sidewalk	2015-09-15 01:51:14.000000	BROOKLYN
106	Illegal Parking	2015-09-15 01:52:10.000000	STATEN ISLAND
107	Illegal Parking	2015-09-15 01:53:19.000000	BRONX
108	Noise - Commercial	2015-09-15 01:56:32.000000	NEW YORK
109	Food Establishment	2015-09-15 01:57:38.000000	BROOKLYN
110	Noise - Park	2015-09-15 01:57:39.000000	NEW YORK
111	Blocked Driveway	2015-09-15 01:58:05.000000	ELMHURST
112	Highway Condition	2015-09-15 02:07:01.000000	STATEN ISLAND
113	Homeless Person Assistance	2015-09-15 02:08:01.000000	NEW YORK

114	Noise - Commercial	2015-09-15 02:09:46.000000	BRONX
115	Noise - Street/Sidewalk	2015-09-15 02:11:19.000000	NEW YORK
116	Noise - Street/Sidewalk	2015-09-15 02:12:49.000000	NEW YORK
117	Illegal Parking	2015-09-15 02:14:04.000000	None

118 rows × 3 columns

This next example shows how to extract just the hour from the time stamp, using SQL's `strftime()`.

In [40]:

```
query = '''
  select CreatedDate, strftime ('%H', CreatedDate) as Hour, ComplaintType
    from data
    limit 5
'''
df = pd.read_sql_query (query, disk_engine)
df
```

Out[40]:

	CreatedDate	Hour	ComplaintType
0	2015-09-15 02:14:04.000000	02	Illegal Parking
1	2015-09-15 02:12:49.000000	02	Noise - Street/Sidewalk
2	2015-09-15 02:11:19.000000	02	Noise - Street/Sidewalk
3	2015-09-15 02:09:46.000000	02	Noise - Commercial
4	2015-09-15 02:08:01.000000	02	Homeless Person Assistance

Exercise 6 (3 points). Construct a tibble called `df_complaints_by_hour`, which contains the total number of complaints during a given hour of the day. That is, the variables should be {`hour`, `count`} where each observation is the total number of complaints (`count`) that occurred during a given `hour`.

> Interpret `hour` as follows: when `hour` is `02`, that corresponds to the open time interval [`02:00:00`, `03:00:00.0`).

In [41]: Student's answer (Top)

```
# Your task: Construct `df_complaints_by_hour` as directed.
### BEGIN SOLUTION
query = '''
  select strftime ('%H', CreatedDate) as hour, count(*) as count
    from data group by hour
'''
df_complaints_by_hour = pd.read_sql_query (query, disk_engine)
### END SOLUTION

# Displays your answer:
display(df_complaints_by_hour)
```

	hour	count
0	00	564703
1	01	23489
2	02	15226
3	03	10164
4	04	8692
5	05	10224
6	06	23051
7	07	42273
8	08	73811
9	09	100077
10	10	114079
11	11	115849

12	12	102392
13	13	100970
14	14	105425
15	15	100271
16	16	86968
17	17	69920
18	18	67467
19	19	57637
20	20	54997
21	21	53126
22	22	52076
23	23	47113

In [42]: Grade cell: `df_complaints_by_hour_test` Score: 3.0 / 3.0 (Top)

```
# Test cell: `df_complaints_by_hour_test`

print ("Reading instructor's solution...")
if False:
    df_complaints_by_hour_soln.to_csv('df_complaints_by_hour_soln.csv', index=False)
df_complaints_by_hour_soln = pd.read_csv ('df_complaints_by_hour_soln.csv')
display (df_complaints_by_hour_soln)

df_complaints_by_hour_norm = df_complaints_by_hour.copy ()
df_complaints_by_hour_norm['hour'] = \
    df_complaints_by_hour_norm['hour'].apply (int)
assert tibbles_are_equivalent (df_complaints_by_hour_norm,
                               df_complaints_by_hour_soln)
print ("\n(Passed.)")
```

```
Reading instructor's solution...
```

	hour	count
0	0	564703
1	1	23489
2	2	15226
3	3	10164
4	4	8692
5	5	10224
6	6	23051
7	7	42273
8	8	73811
9	9	100077
10	10	114079
11	11	115849
12	12	102392
13	13	100970
14	14	105425
15	15	100271
16	16	86968
17	17	69920
18	18	67467
19	19	57637
20	20	54997

21	21	53126
22	22	52076
23	23	47113

```
(Passed.)
```

Let's take a quick look at the hour-by-hour breakdown above.

In [43]:

```
%%opts Bars [width=640 height=320]
Bars(df_complaints_by_hour, kdims=['hour'], vdims=['count'])
```

Out[43]:

An unusual aspect of these data are the excessively large number of reports associated with hour 0 (midnight up to but excluding 1 am), which would probably strike you as suspicious. Indeed, the reason is that there are some complaints that are dated but with no associated time, which was recorded in the data as exactly `00:00:00.000`.

In [44]:

```
query = '''
  select count(*)
    from data
    where strftime ('%H:%M:%f', CreatedDate) = '00:00:00.000'
'''

pd.read_sql_query (query, disk_engine)
```

Out[44]:

	count(*)
0	532285

Exercise 7 (2 points). What is the most common hour for noise complaints? Compute a tibble called `df_noisy_by_hour` whose variables are {`hour`, `count`} and whose observations are the number of noise complaints that occurred during a given `hour`. Consider a "noise complaint" to be any complaint string containing the word `noise`. Be sure to filter out any dates without an associated time, i.e., a timestamp of `00:00:00.000`.

In [45]:

Student's answer (Top)

```
### BEGIN SOLUTION
query = '''
  select strftime('%H', CreatedDate) as hour,
         count(*) as count
    from data
    where (ComplaintType like '%noise%')
      and (strftime('%H:%M:%f', CreatedDate) <> '00:00:00.000')
    group by hour
    order by hour
'''
df_noisy_by_hour = pd.read_sql_query(query, disk_engine)
### END SOLUTION

display(df_noisy_by_hour)
```

	hour	count
0	00	15349
1	01	11284
2	02	7170
3	03	4241
4	04	3083
5	05	2084
6	06	2832
7	07	3708
8	08	4553
9	09	5122
10	10	4672

11	11	4745
12	12	4316
13	13	4364
14	14	4505
15	15	4576
16	16	4957
17	17	5126
18	18	6797
19	19	7958
20	20	9790
21	21	12659
22	22	17155
23	23	19343

In [46]: Grade cell: `df_noisy_by_hour_test` Score: 2.0 / 2.0 (Top)

```
# Test cell: `df_noisy_by_hour_test`

print ("Reading instructor's solution...")
if False:
    df_noisy_by_hour.to_csv('df_noisy_by_hour_soln.csv', index=False)
df_noisy_by_hour_soln = pd.read_csv ('df_noisy_by_hour_soln.csv')
display(df_noisy_by_hour_soln)

df_noisy_by_hour_norm = df_noisy_by_hour.copy()
df_noisy_by_hour_norm['hour'] = \
    df_noisy_by_hour_norm['hour'].apply(int)
assert tibbles_are_equivalent (df_noisy_by_hour_norm,
                               df_noisy_by_hour_soln)
print ("\n(Passed.)")
```

```
Reading instructor's solution...
```

	hour	count
0	0	15349
1	1	11284
2	2	7170
3	3	4241
4	4	3083
5	5	2084
6	6	2832
7	7	3708
8	8	4553
9	9	5122
10	10	4672
11	11	4745
12	12	4316
13	13	4364
14	14	4505
15	15	4576
16	16	4957
17	17	5126
18	18	6797
19	19	7958

20	20	9790
21	21	12659
22	22	17155
23	23	19343

```
(Passed.)
```

In [47]:

```
%%opts Bars [width=640 height=320]
Bars(df_noisy_by_hour, kdims=['hour'], vdims=['count'])
```

Out[47]:

Exercise 8 (ungraded). Create a line chart to show the fraction of complaints (y-axis) associated with each hour of the day (x-axis), with each complaint type shown as a differently colored line. Show just the top 5 complaints (`top_complaints[:5]`). Remember to exclude complaints with a zero-timestamp (i.e., `00:00:00.000`).

> **Note.** This exercise is ungraded but we recommend spending some time giving it a try! Feel free to discuss your approaches to this problem on the discussion forums (but do try to do it yourself first).

In [48]: Student's answer Score: 0.0 / 0.0 (Top)

```
### BEGIN SOLUTION
from bokeh.charts import Line
top_5_str = strs_to_args (list (top_complaints[:5]['type']))

query_normalizing_factor = '''
  select ComplaintType as type,
         count (*) as total
    from data
    where (type in ({top}))
      and (strftime ('%H:%M:%f', CreatedDate) <> '00:00:00.000')
    group by type
'''.format (top=top_5_str)
df_timestamped_complaints_by_type = pd.read_sql_query (query_normalizing_factor,
                                                       disk_engine)
display (df_timestamped_complaints_by_type)

query = '''
  select ComplaintType as type,
         strftime ('%H', CreatedDate) as hour,
         count (*) as count
    from data
    where (type in ({top}))
      and (strftime ('%H:%M:%f', CreatedDate) <> '00:00:00.000')
    group by type, hour
    order by type, hour
'''.format (top=top_5_str)

df_complaints_by_type_and_hour = pd.read_sql_query (query, disk_engine)
df_complaints_by_type_and_hour['hour'] = \
    df_complaints_by_type_and_hour['hour'].apply (int)
df_complaints_by_type_and_hour = \
  df_complaints_by_type_and_hour.merge (df_timestamped_complaints_by_type,
                                        on=['type'],
                                        how='left')
df_complaints_by_type_and_hour['fraction'] = \
  df_complaints_by_type_and_hour['count'] / df_complaints_by_type_and_hour['total']
display (df_complaints_by_type_and_hour.head ())

p = Line (df_complaints_by_type_and_hour,
          x='hour', y='fraction', color='type')
show (p)
### END SOLUTION
```

```
/Users/richie/anaconda/lib/python3.5/site-packages/bokeh/util/deprecation.py:34: BokehDep
recationWarning:
The bokeh.charts API has moved to a separate 'bkcharts' package.

This compatibility shim will remain until Bokeh 1.0 is released.
After that, if you want to use this API you will have to install
the bkcharts package explicitly.
```

```
  warn(message)
```

	type	total
0	Blocked Driveway	95079
1	HEAT/HOT WATER	8116
2	Illegal Parking	83961
3	Street Condition	124346
4	Street Light Condition	98560

	type	hour	count	total	fraction
0	Blocked Driveway	0	3030	95079	0.031868
1	Blocked Driveway	1	2136	95079	0.022466
2	Blocked Driveway	2	1379	95079	0.014504
3	Blocked Driveway	3	1040	95079	0.010938
4	Blocked Driveway	4	958	95079	0.010076

Learn more