GTx CSE6040x
FA20: Computing for Data Analysis

Help mrajagopal6

Course Progress Dates Discussion Wiki

Course / Module 1: Representing, Transforming, and Visualizing ... / Solution: Noteboo...

Previous Next

Sample solutions

Bookmark this page

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 si 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. Howeve naturally, invoke it from Python (https://docs.python.org/3/library/sqlite3.html). But all of the basic techniques apply to any commercial SQL ba

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]:
```
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 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 databas

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, variab columns. In the relational database world, we sometimes refer to rows as *items* or *records* and columns as *attributes*. We'll use all of these term 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 Ge issued ID numbers. You might start by creating a table named `Students` to hold this data. You can create the table using the command, `CREA` (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 remove any existing `example.db` file or destroy any existing table; you can do the latter with the SQL command, `DROP TABLE IF E` `Students`.

In [3]:
```
# If this is not the first time you run this cell,
# you need to delete the existed "Students" table first
c.execute("DROP TABLE IF EXISTS Students")

# create a table named "Students" with 2 columns: "gtid" and "name".
# the type for column "gtid" is integer and for "name" is text.
c.execute("CREATE TABLE Students (gtid INTEGER, name TEXT)")
```

Out[3]: `<sqlite3.Cursor at 0x7f3354403500>`

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 0x7f3354403500>
```

Commitment issues. The commands above modify the database. However, these are temporary modifications and aren't actually saved to the 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 poin discussed here (https://stackoverflow.com/questions/13642956/commit-behavior-and-atomicity-in-python-sqlite3-module). However, i 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')]

# '?' question marks are placeholders for the two columns in Students table
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 c `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, 'Zha'), (
k'), (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 grac

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 in terms of GPA (i.e. 4 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 0x7f3354403500>
```

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)

(Note: See `students` table above to get the GT IDs for Vuduc, Sokol, and Chau. You don't have to write any code to retrieve their GT IDs. You them in manually. However, it would be a good and extra practice for you if you can use some sql commands to retrieve their IDs.)

In [9]:
```
### 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 8803', 1.0),
SE 6040', 4.0), (991, 'ISYE 6740', 4.0), (456, 'CSE 6040', 4.0), (456, 'CSE 6740', 2.0), (456, '
3', 3.0)]
```

In [10]:

```
# 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)

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, `NUL`
- `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 t 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 t 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 wa 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 results
    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 q 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 forma

In [12]:

```
# 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'
```

```
        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]:
```
# 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(results1),

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 Write your query as a string variable named `query`, which the subsequent code will execute.

In [14]:
```
# 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]:
```
# Test cell: `left_join_test`

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

```
(Passed!)
```

Aggregations

Another common style of query is an *aggregation* (https://www.sqlite.org/lang_aggfunc.html), which is a summary of information across multipl rather than the raw records themselves.

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

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.

In [17]:
```
c.close()
conn.close()
```

What next? It's now a good time to look at a different tutorial which reviews this material and introduces some additional topics: A thorough gu database operations in Python (http://sebastianraschka.com/Articles/2014_sqlite_in_python_tutorial.html).

In [ ]:
```

```

Part 1: 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 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 via 311 calls. The full dataset is available at the NYC open (https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9). Our subset is about 6 GB and 10 mill complaints, so 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. (I 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. These include a few functions from Notebook 7.

In [1]:
```
import sys
print(sys.version) # Print Python version -- On Vocareum, it should be 3.7+

from IPython.display import display
import pandas as pd

from nb7utils import canonicalize_tibble, tibbles_are_equivalent, cast
```

```
3.7.5 (default, Dec 18 2019, 06:24:58)
[GCC 5.5.0 20171010]
```

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 [2]:
```
from nb9utils import download, get_path, auxfiles

for filename, checksum in auxfiles.items():
    download(filename, checksum=checksum, url_suffix="lab9-sql/")

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

(https://bokeh.org) BokehJS 1.4.0 successfully loaded.

```
[https://cse6040.gatech.edu/datasets/lab9-sql/df_complaints_by_city_soln.csv]
==> 'resource/asnlib/publicdata/df_complaints_by_city_soln.csv' is already available.
==> Checksum test passes: b07d65c208bd791ea21679a3551ae265
==> 'resource/asnlib/publicdata/df_complaints_by_city_soln.csv' is ready!

[https://cse6040.gatech.edu/datasets/lab9-sql/df_complaints_by_hour_soln.csv]
==> 'resource/asnlib/publicdata/df_complaints_by_hour_soln.csv' is already available.
==> Checksum test passes: f06fcd917876d51ad52ddc13b2fee69e
```

```
==> 'resource/asnlib/publicdata/df_complaints_by_hour_soln.csv' is ready!

[https://cse6040.gatech.edu/datasets/lab9-sql/df_noisy_by_hour_soln.csv]
==> 'resource/asnlib/publicdata/df_noisy_by_hour_soln.csv' is already available.
==> Checksum test passes: 30f3fa7c753d4d3f4b3edfa1f6d05bcc
==> 'resource/asnlib/publicdata/df_noisy_by_hour_soln.csv' is ready!

[https://cse6040.gatech.edu/datasets/lab9-sql/df_plot_stacked_fraction_soln.csv]
==> 'resource/asnlib/publicdata/df_plot_stacked_fraction_soln.csv' is already available.
==> Checksum test passes: ab46e3f514824529edf65767771d4622
==> 'resource/asnlib/publicdata/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 [3]:
```
from nb9utils import make_barchart, make_stacked_barchart
from bokeh.io import show
```

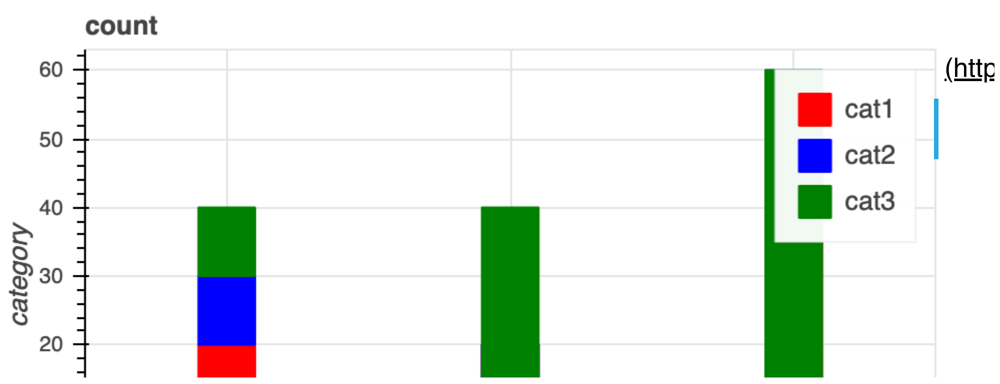
In [4]:
```
def demo_bar():
    from bokeh.plotting import figure
    from bokeh.models import ColumnDataSource
    data = [
        ['201720', 'cat1', 20],
        ['201720', 'cat2', 30],
        ['201720', 'cat3', 40],
        ['201721', 'cat1', 20],
        ['201721', 'cat2', 0],
        ['201721', 'cat3', 40],
        ['201722', 'cat1', 50],
        ['201722', 'cat2', 60],
        ['201722', 'cat3', 10],
    ]
    df = pd.DataFrame(data, columns=['week', 'category', 'count'])
    pt = df.pivot('week', 'category', 'count')
    pt.cumsum(axis=1)
    return df, pt

df_demo, pt_demo = demo_bar()
pt_demo
```

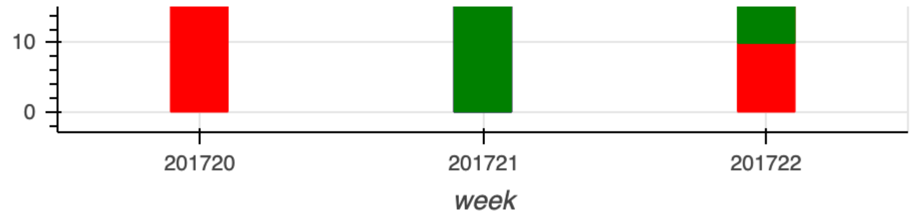
Out[4]:

category	cat1	cat2	cat3
week			
201720	20	30	40
201721	20	0	40
201722	50	60	10

In [5]:
```
def demo_stacked_bar(pt):
    from bokeh.models.ranges import FactorRange
    from bokeh.io import show
    from bokeh.plotting import figure
    p = figure(title="count",
               x_axis_label='week', y_axis_label='category',
               x_range = FactorRange(factors=list(pt.index)),
               plot_height=300, plot_width=500)
    p.vbar(x=pt.index, bottom=0, top=pt.cat1, width=0.2, color='red', legend='cat1')
    p.vbar(x=pt.index, bottom=pt.cat1, top=pt.cat2, width=0.2, color='blue', legend='cat2')
    p.vbar(x=pt.index, bottom=pt.cat2, top=pt.cat3, width=0.2, color='green', legend='cat3')
    return p

show(demo_stacked_bar(pt_demo))
```

```
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
```

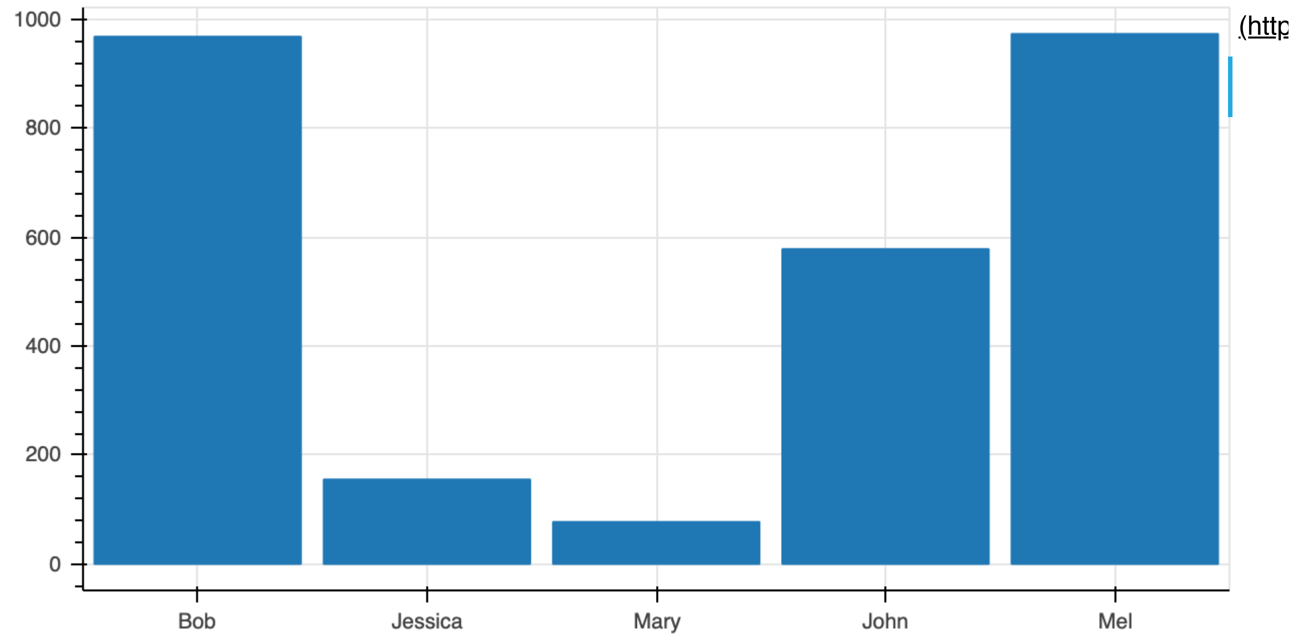




In [6]:
```
# Build a Pandas data frame
names = ['Bob','Jessica','Mary','John','Mel']
births = [968, 155, 77, 578, 973]
name_birth_pairs = list(zip(names, births))
baby_names = pd.DataFrame(data=name_birth_pairs, columns=['Names', 'Births'])
display(baby_names)
```

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

In [7]:
```
p = make_barchart(baby_names, 'Names', 'Births', kwargs_figure={'plot_width': 640, 'plot_height
show(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.

> If you are not running on Vocareum, you will need to download this file manually from the following link and place it locally in a (nested) subdirectory or folder named `resource/asnlib/publicdata`.
>
> Link to the pre-constructed NYC 311 Database on MS OneDrive (https://onedrive.live.com/download?cid=FD520DDC6BE92730&resid=FD520DDC6BE92730%21616&authkey=AEeP_4E1uh-vyDE)

In [8]:
```
from nb9utils import download_nyc311db
DB_FILENAME = download_nyc311db()
```

```
[https://onedrive.live.com/download?cid=FD520DDC6BE92730&resid=FD520DDC6BE92730%21616&authkey=AE
vyDENYC-311-2M.db]
==> 'resource/asnlib/publicdata/NYC-311-2M.db' is already available.
==> Checksum test passes: f48eba2fb06e8ece7479461ea8c6dee9
==> 'resource/asnlib/publicdata/NYC-311-2M.db' is ready!
```

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

In [9]:
```
# Connect
import sqlite3 as db
disk_engine = db.connect('file:{}?mode=ro'.format(DB_FILENAME), uri=True)
```

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 use the SQL reader built into `pandas`.

In [10]:
```
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 7.23535 seconds.
```

Out[10]:

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

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

In [11]:
```
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 0.00242829 seconds.
```

Out[11]:

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

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

In [12]:
```
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[12]:

	City
0	None
1	NEW YORK

1	NEW YORK
2	BRONX
3	STATEN ISLAND
4	ELMHURST

However, `DISTINCT` applied to strings is case-sensitive. We'll deal with that momentarily.

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

In [13]:
```
query = '''
  SELECT ComplaintType, Descriptor, Agency
    FROM data
    GROUP BY ComplaintType
'''

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

```
(200, 3)
```

Out[13]:

	ComplaintType	Descriptor	Agency
0	AGENCY	HOUSING QUALITY STANDARDS	HPD
1	APPLIANCE	ELECTRIC/GAS RANGE	HPD
2	Adopt-A-Basket	10A Adopt-A-Basket	DSNY
3	Agency Issues	Bike Share	DOT
4	Air Quality	Air: Odor/Fumes, Vehicle Idling (AD3)	DEP

`GROUP BY` aggregations. A common pattern is to combine grouping with aggregation. For example, suppose we want to count how many tim complaint occurs. Here is one way to do it.

In [14]:
```
query = '''
  SELECT ComplaintType, COUNT(*)
    FROM data
    GROUP BY ComplaintType
    LIMIT 10
'''

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

Out[14]:

	ComplaintType	COUNT(*)
0	AGENCY	2
1	APPLIANCE	11263
2	Adopt-A-Basket	50
3	Agency Issues	7428
4	Air Quality	8151

Character-case conversions. From the two preceding examples, observe that the strings employ a mix of case conventions (i.e., lowercase v vs. mixed case). A convenient way to query and "normalize" case is to apply SQL's `UPPER()` and `LOWER()` functions. Here is an example:

In [15]:
```
query = '''
  SELECT LOWER(ComplaintType), LOWER(Descriptor), LOWER(Agency)
    FROM data
    GROUP BY LOWER(ComplaintType)
    LIMIT 10
'''

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

Out[15]:

	LOWER(ComplaintType)	LOWER(Descriptor)	LOWER(Agency)
0	adopt-a-basket	10a adopt-a-basket	dsny
1	agency	housing quality standards	hpd
2	agency issues	bike share	dot
3	air quality	air: odor/fumes, vehicle idling (ad3)	dep

3	air quality	air: odor/fumes, vehicle idling (ad3)	dep
4	animal abuse	other (complaint details)	nypd

Filtered aggregations: `HAVING` clauses. A common pattern for aggregation queries (e.g., `GROUP BY` plus `COUNT()`) is to filter the grouped res cannot do that with a `WHERE` clause alone, because `WHERE` is applied *before* grouping.

As an example, recall that some `ComplaintType` values are in all uppercase whereas some use mixed case. Since we didn't inspect all of the even be some are all lowercase. Worse, you would expect some inconsistencies. For instance, it turns out that both `"Plumbing"` (mixed case) `"PLUMBING"` (all caps) appear. Here is a pair of queries that makes this point.

In [16]:
```
query0 = "SELECT DISTINCT ComplaintType FROM data"
df0 = pd.read_sql_query(query0, disk_engine)
print("Found {} unique `ComplaintType` strings.".format(len(df0)))
display(df0.head())

query1 = "SELECT DISTINCT LOWER(ComplaintType) FROM data"
df1 = pd.read_sql_query(query1, disk_engine)
print("\nFound {} unique `LOWER(ComplaintType)` strings.".format(len(df1)))
display(df1.head())

print("\n==> Therefore, there are {} cases that are duplicated. Which ones?".format(len(df0) - l
```

```
Found 200 unique `ComplaintType` strings.
```

	ComplaintType
0	Illegal Parking
1	Noise - Street/Sidewalk
2	Noise - Commercial
3	Homeless Person Assistance
4	Highway Condition

```
Found 198 unique `LOWER(ComplaintType)` strings.
```

	LOWER(ComplaintType)
0	illegal parking
1	noise - street/sidewalk
2	noise - commercial
3	homeless person assistance
4	highway condition

```
==> Therefore, there are 2 cases that are duplicated. Which ones?
```

What if we wanted a query that identifies these inconsistent capitalizations? Here is one way to do it, which demonstrates the `HAVING` clause. (**nested query**, that is, it performs one query and then selects immediately from that result.) Can you read it and figure out what it is doing and v

In [17]:
```
query2 = '''
    SELECT ComplaintType, COUNT(*)
      FROM (SELECT DISTINCT ComplaintType FROM data)
      GROUP BY LOWER(ComplaintType)
      HAVING COUNT(*) >= 2
'''
df2 = pd.read_sql_query(query2, disk_engine)
df2
```

Out[17]:

	ComplaintType	COUNT(*)
0	Elevator	2
1	PLUMBING	2

You should see that "elevator" and "plumbing" complaints use inconsistent case, which we can then verify directly using the next technique, th

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` opera it to see the two inconsistent-capitalization complaint types from above.

In [18]:
```
query = '''
    SELECT DISTINCT ComplaintType
      FROM data
      WHERE LOWER(ComplaintType) IN ("plumbing", "elevator")
'''
df = pd.read_sql_query(query, disk_engine)
```

```
df.head()
```

Out[18]:

	ComplaintType
0	PLUMBING
1	Elevator
2	Plumbing
3	ELEVATOR

Renaming columns: `AS` operator. Sometimes you might want to rename a result column. For instance, the following query counts the number 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 colum resulting data frame, you can give it a more "friendly" name using the `AS` operator.

In [19]:
```
query = '''
  SELECT Agency, COUNT(*) AS NumComplaints
    FROM data
    GROUP BY Agency
'''
df = pd.read_sql_query(query, disk_engine)
df.head()
```

Out[19]:

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

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

In [20]:
```
query = '''
  SELECT Agency, COUNT(*) AS NumComplaints
    FROM data
    GROUP BY UPPER(Agency)
    ORDER BY NumComplaints
'''
df = pd.read_sql_query(query, disk_engine)
df.tail()
```

Out[20]:

	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) prefixing the `ORDER BY` attribute with a `-` (minus) symbol. Alternatively, you can use `DESC` to achieve the same result.

In [21]:
```
query = '''
  SELECT Agency, COUNT(*) AS NumComplaints
    FROM data
    GROUP BY UPPER(Agency)
    ORDER BY -NumComplaints
'''

# Alternative: query =
'''
SELECT Agency, COUNT(*) AS NumComplaints
    FROM data
    GROUP BY UPPER(Agency)
    ORDER BY NumComplaints DESC
'''

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

Out[21]:

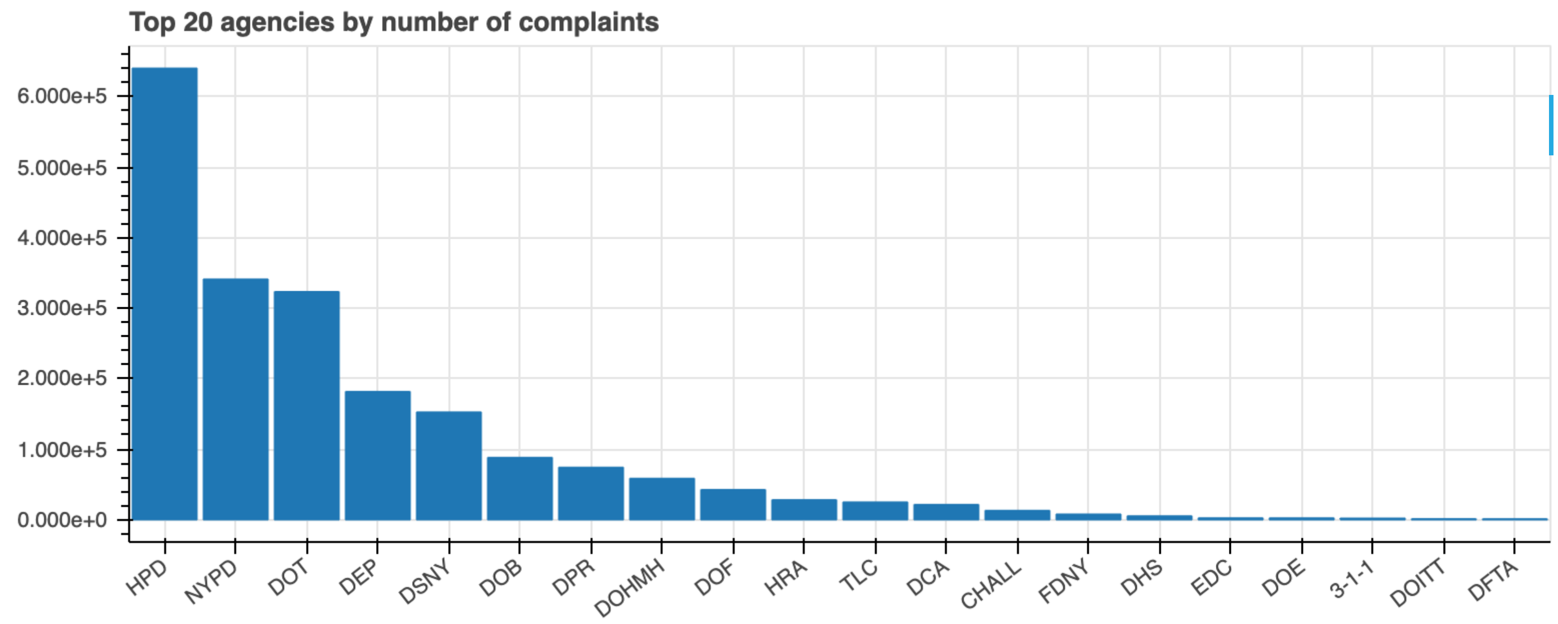
	Agency	NumComplaints
0	HPD	640096

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 [22]:
```
p = make_barchart(df[:20], 'Agency', 'NumComplaints',
                  {'title': 'Top 20 agencies by number of complaints',
                   'plot_width': 800, 'plot_height': 320})
p.xaxis.major_label_orientation = 0.66
show(p)
```



(http

Exercise 1 (2 points). Create a string, `query`, containing an SQL query that will return the number of complaints by type. The columns should b `type` and `freq`, and the results should be sorted in descending order by `freq`. Also, since we know some complaints use an inconsistent cas function convert complaints to lowercase.

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

In [23]:
```
del query # clears any existing `query` variable; you should define it, below!

# Define a variable named `query` containing your solution
### BEGIN SOLUTION
query = '''
  SELECT LOWER(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[23]:

	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 [24]:
```
# 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 named 'type
q', not {}".format(set(df_complaint_freq.columns))

soln = ['heat/hot water', 'street condition', 'street light condition', 'blocked driveway', 'il
ng', 'unsanitary condition', 'paint/plaster', 'water system', 'plumbing', 'noise', 'noise - stre
```

```
k', 'traffic signal condition', 'noise - commercial', 'door/window', 'water leak', 'dirty condi
wer', 'sanitation condition', 'dof literature request', 'electric', 'rodent', 'flooring/stairs',
 construction/plumbing', 'building/use', 'broken muni meter', 'general', 'missed collection (al
s)', 'benefit card replacement', 'derelict vehicle', 'noise - vehicle', 'damaged tree', 'consum
t', 'derelict vehicles', 'taxi complaint', 'overgrown tree/branches', 'graffiti', 'snow', 'opin
mayor', 'appliance', 'maintenance or facility', 'animal abuse', 'dead tree', 'elevator', 'hpd l
equest', 'root/sewer/sidewalk condition', 'safety', 'food establishment', 'scrie', 'air quality
 issues', 'construction', 'highway condition', 'other enforcement', 'water conservation', 'side
ion', 'indoor air quality', 'street sign - damaged', 'traffic', 'fire safety director - f58', '
rson assistance', 'homeless encampment', 'special enforcement', 'street sign - missing', 'noise
'vending', 'for hire vehicle complaint', 'food poisoning', 'special projects inspection team (s
ardous materials', 'electrical', 'dot literature request', 'litter basket / request', 'taxi rep
gal tree damage', 'dof property - reduction issue', 'unsanitary animal pvt property', 'asbestos
'vacant lot', 'dca / doh new license application request', 'street sign - dangling', 'smoking',
of park rules', 'outside building', 'animal in a park', 'noise - helicopter', 'school maintenan
nternal', 'boilers', 'industrial waste', 'sweeping/missed', 'overflowing litter baskets', 'non-
heat', 'curb condition', 'drinking', 'standing water', 'indoor sewage', 'water quality', 'eap i
f59', 'derelict bicycle', 'noise - house of worship', 'dca literature request', 'recycling enfo
'dof parking - tax exemption', 'broken parking meter', 'request for information', 'taxi compli
ashed dog', 'urinating in public', 'unsanitary pigeon condition', 'investigations and disciplin
'bridge condition', 'ferry inquiry', 'bike/roller/skate chronic', 'public payphone complaint',
'best/site safety', 'sweeping/inadequate', 'disorderly youth', 'found property', 'mold', 'senio
mplaint', 'fire alarm - reinspection', 'for hire vehicle report', 'city vehicle placard complai
s and derricks', 'ferry complaint', 'illegal animal kept as pet', 'posting advertisement', 'har
s/wasps', 'panhandling', 'scaffold safety', 'oem literature request', 'plant', 'bus stop shelte
t', 'collection truck noise', 'beach/pool/sauna complaint', 'complaint', 'compliment', 'illegal
, 'fire alarm - modification', 'dep literature request', 'drinking water', 'fire alarm - new sy
son ivy', 'bike rack condition', 'emergency response team (ert)', 'municipal parking facility',
g', 'unsanitary animal facility', 'animal facility - no permit', 'miscellaneous categories', 'm
ts', 'literature request', 'special natural area district (snad)', 'highway sign - damaged', 'p
t', 'adopt-a-basket', 'ferry permit', 'invitation', 'window guard', 'parking card', 'illegal an
'stalled sites', 'open flame permit', 'overflowing recycling baskets', 'highway sign - missing',
ssembly', 'dpr literature request', 'fire alarm - addition', 'lifeguard', 'transportation provi
nt', 'dfta literature request', 'bottled water', 'highway sign - dangling', 'dhs income savings
t', 'legal services provider complaint', 'foam ban enforcement', 'tunnel condition', 'calorie l
'fire alarm - replacement', 'x-ray machine/equipment', 'sprinkler - mechanical', 'hazmat storag
nning', 'radioactive material', 'rangehood', 'squeegee', 'srde', 'building condition', 'sg-98',
- mechanical', 'agency', 'forensic engineering', 'public assembly - temporary', 'vacant apartme
atory', 'sg-99']
assert all(soln[:25] == df_complaint_freq['type'].iloc[:25])

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

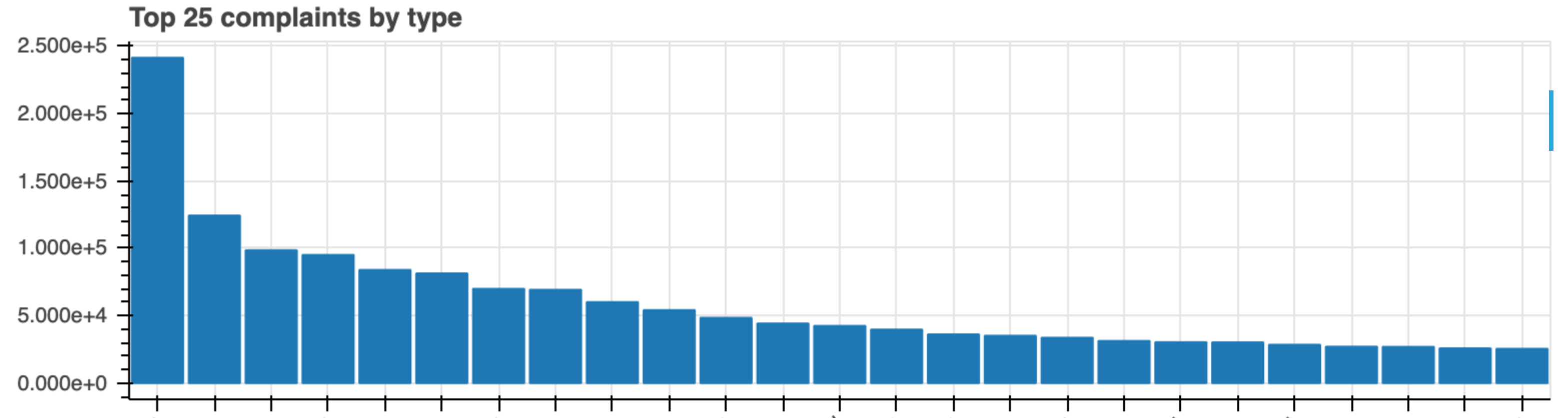
Top 10 complaints:

	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	60105
9	noise	54165

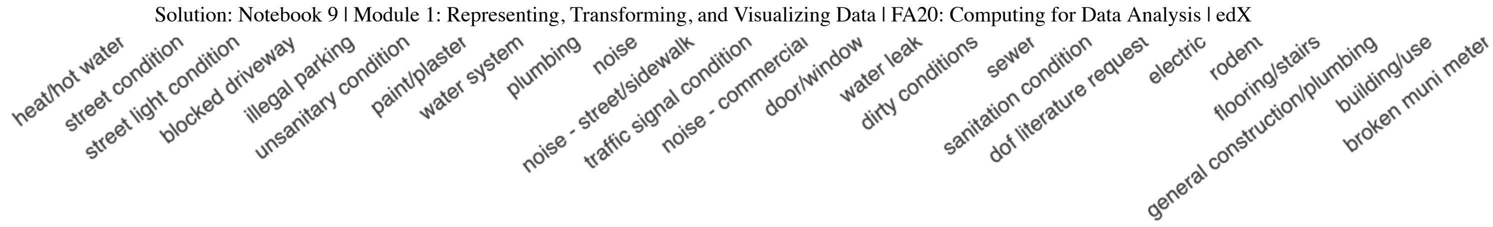
(Passed.)

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 [25]:
```
p = make_barchart(df_complaint_freq[:25], 'type', 'freq',
                  {'title': 'Top 25 complaints by type',
                   'plot_width': 800, 'plot_height': 320})
p.xaxis.major_label_orientation = 0.66
show(p)
```



(http



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 th
use the `LIKE` operator combined with the string wildcard, `%`, to look for case-insensitive substring matches.

In [26]:

```
query = '''
  SELECT LOWER(ComplaintType) AS type, COUNT(*) AS freq
    FROM data
    WHERE LOWER(ComplaintType) LIKE '%noise%'
    GROUP BY type
    ORDER BY -freq
'''

df_noisy = pd.read_sql_query(query, disk_engine)
print("Found {} queries with 'noise' in them.".format(len(df_noisy)))
df_noisy
```

```
Found 8 queries with 'noise' in them.
```

Out[26]:

	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 cc
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 n
complaints by that city.

Like complaint types, cities are not capitalized consistently. Therefore, standardize the city names by converting them to **uppercase**.

In [27]:

```
del query # define a new `query` variable, below

# Define your `query`, here:
### BEGIN SOLUTION
query = '''
  SELECT UPPER(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[27]:

	name	freq
0	BROOKLYN	579363
1	NEW YORK	385655
2	BRONX	342533
3	None	168692
4	STATEN ISLAND	92509
5	JAMAICA	46683
6	FLUSHING	35504
7	ASTORIA	31873

8	RIDGEWOOD	21618
9	WOODSIDE	15932

Brooklynites are "vocal" about their issues, evidently.

In [28]:
```
# 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.)
```

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

The next example demonstrates this clause. Note that it also filters out the 'None' cases, where the <> operator denotes "not equal to." Lastly, ensures that the returned city names are uppercase.

> The `COLLATE NOCASE` clause modifies the column next to which it appears. So if you are grouping by more than one key and want to case-insensitive, you need to write, `... GROUP BY ColumnA COLLATE NOCASE, ColumnB COLLATE NOCASE ...`.

In [29]:
```
query = '''
  SELECT UPPER(City) AS name, COUNT(*) AS freq
    FROM data
    WHERE name <> 'None'
    GROUP BY City COLLATE NOCASE
    ORDER BY -freq
    LIMIT 10
'''
df_whiny_cities2 = pd.read_sql_query(query, disk_engine)
df_whiny_cities2
```

Out[29]:

	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

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

In [30]:
```
TOP_CITIES = list(df_whiny_cities2.head(7)['name'])
TOP_CITIES
```

Out[30]:
```
['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_lis` 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"'
```

> **Tip.** Try to avoid manipulating the input `str_list` directly and returning the updated `str_list`. This may result in your function addi to the strings in your list each time the function is used (which will be more than once in this notebook!)

In [31]:
```
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 [32]:
```
# 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 == \
       '"BROOKLYN", "NEW YORK", "BRONX", "STATEN ISLAND", "Jamaica", "Flushing", "ASTORIA"'.uppe
assert TOP_CITIES == list(df_whiny_cities2.head(7)['name']), \
       "Does your implementation cause the `TOP_CITIES` variable to change? If so, you need to 

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

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

(Passed.)
```

Exercise 4 (3 points). Suppose we want to look at the number of complaints by type *and* by city **for only the top cities**, i.e., those in the list `TO` computed above. Execute an SQL query to produce a tibble named `df_complaints_by_city` with the variables {`complaint_type`, `city_` `complaint_count`}.

In your output `DataFrame`, convert all city names to uppercase and convert all complaint types to lowercase.

In [33]:
```
### BEGIN SOLUTION
# Version 0:
query0 = """SELECT LOWER(ComplaintType) AS complaint_type,
                   UPPER(City) AS city_name,
                   COUNT(*) AS complaint_count
            FROM data
            WHERE city_name IN ({})
            GROUP BY City COLLATE NOCASE, ComplaintType COLLATE NOCASE
            ORDER BY city_name, complaint_type, complaint_count""".format(strs_to_args(TOP_CITI

# Version 1:
query1 = """SELECT LOWER(ComplaintType) AS complaint_type,
                   UPPER(City) AS city_name,
                   COUNT(*) AS complaint_count
            FROM data
            WHERE city_name IN ({})
            GROUP BY city_name, complaint_type
            ORDER BY city_name, complaint_type, complaint_count""".format(strs_to_args(TOP_CITI

df_complaints_by_city = pd.read_sql_query(query1, disk_engine)
### END SOLUTION

# Previews the results of your query:
print("Found {} records.".format(len(df_complaints_by_city)))
display(df_complaints_by_city.head(10))
```

```
Found 1042 records.
```

	complaint_type	city_name	complaint_count
0	air quality	ASTORIA	142
1	animal abuse	ASTORIA	174
2	animal facility - no permit	ASTORIA	3
3	animal in a park	ASTORIA	29
4	appliance	ASTORIA	70
5	asbestos	ASTORIA	36
6	beach/pool/sauna complaint	ASTORIA	2
7	best/site safety	ASTORIA	18
8	bike rack condition	ASTORIA	3
9	bike/roller/skate chronic	ASTORIA	7

In [34]:
```
# Test cell: `df_complaints_by_city__test`
```

```
print("Reading instructor's solution...")
if False:
    df_complaints_by_city.to_csv(get_path('df_complaints_by_city_soln.csv'), index=False)
df_complaints_by_city_soln = pd.read_csv(get_path('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 Bokeh to visualize the results as a stacked bar chart.

In [35]:
```
# 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	60105
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 [36]:
```
# 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')
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

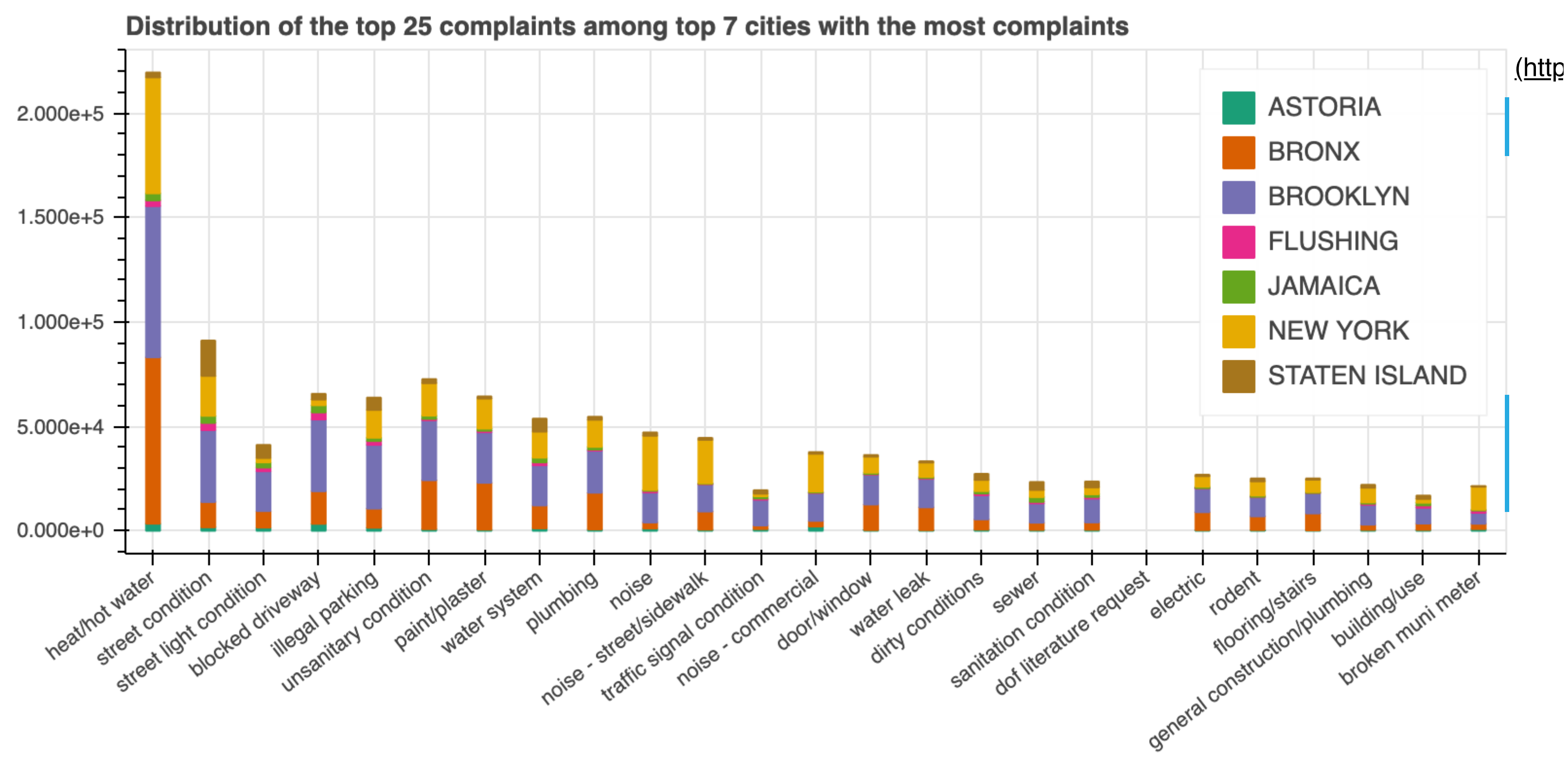
0	heat/hot water	241430	heat/hot water	ASTORIA	3396.0
1	heat/hot water	241430	heat/hot water	BRONX	79690.0
2	heat/hot water	241430	heat/hot water	BROOKLYN	72410.0
3	heat/hot water	241430	heat/hot water	FLUSHING	2741.0
4	heat/hot water	241430	heat/hot water	JAMAICA	3376.0

...

In [37]:

```
# Some code to render a Bokeh stacked bar chart

kwargs_figure = {'title': "Distribution of the top 25 complaints among top 7 cities with the mos
ts",
                 'width': 800,
                 'height': 400,
                 'tools': "hover,crosshair,pan,box_zoom,wheel_zoom,save,reset,help"}

def plot_complaints_stacked_by_city(df, y='complaint_count'):
    p = make_stacked_barchart(df, 'complaint_type', 'city_name', y,
                              x_labels=list(top_complaints['type']), bar_labels=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}")]
    return p

show(plot_complaints_stacked_by_city(df_plot))
```

```
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
```



(http

Exercise 5 (2 points). Suppose we want to create a different stacked bar plot that shows, for each complaint type t and city c, the fraction of al of type t (across all cities, not just the top ones) that occurred in city c. Store your result in a dataframe named `df_plot_fraction`. It should columns as `df_plot`, **except** that the `complaint_count` column should be replaced by one named `complaint_frac`, which holds the frac

> **Hint.** Everything you need is already in `df_plot`.
>
> **Note.** The test cell will create the chart in addition to checking your result. Note that the normalized bars will not necessarily add up to not?

In [38]:

```
### BEGIN SOLUTION
df_plot_fraction = df_plot.copy()
df_plot_fraction['complaint_frac'] = df_plot['complaint_count'] / df_plot['freq']
del df_plot_fraction['complaint_count']
### END SOLUTION
```

```
### END SOLUTION

df_plot_fraction.head()
```

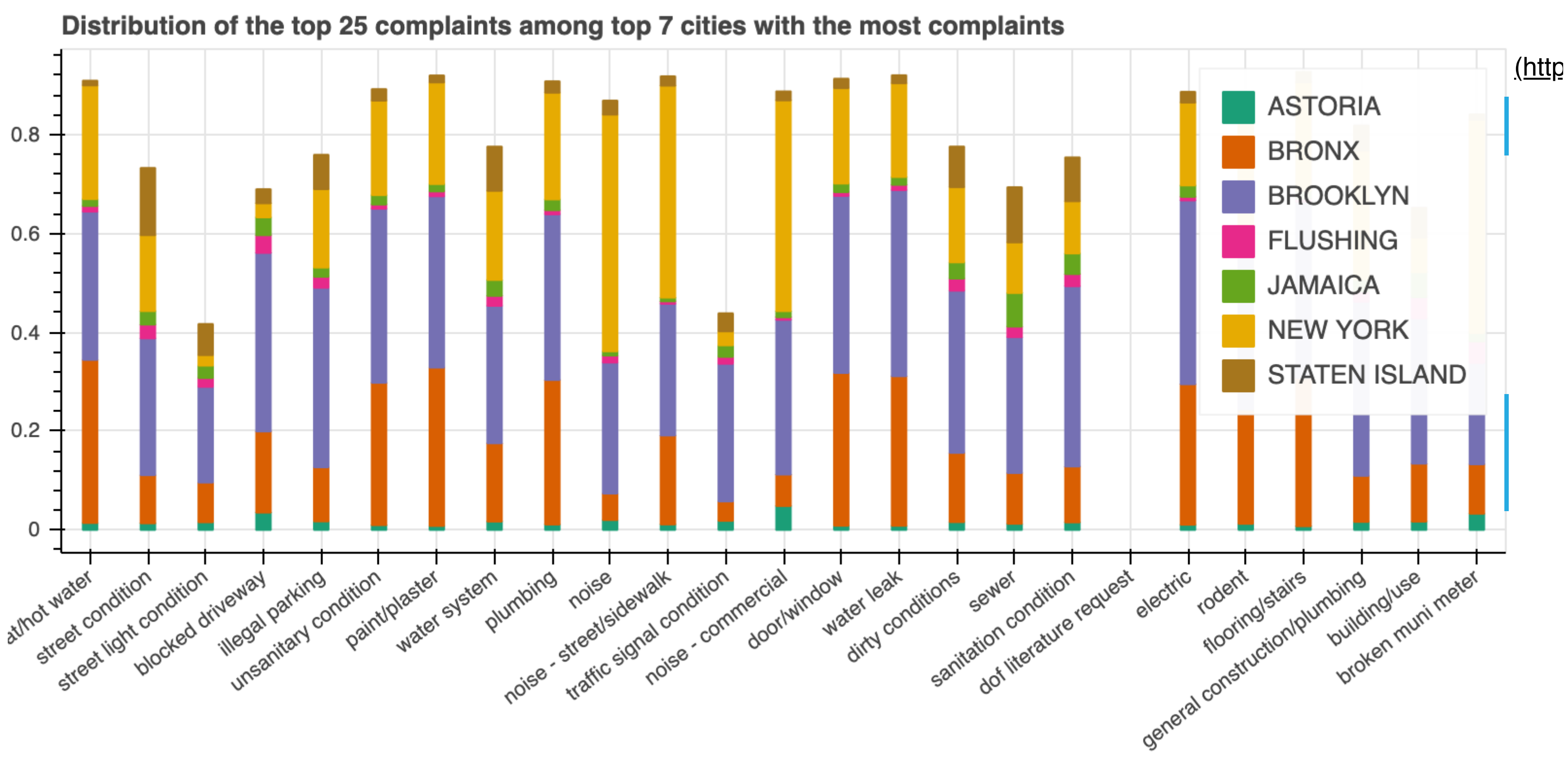
Out[38]:

	type	freq	complaint_type	city_name	complaint_frac
0	heat/hot water	241430	heat/hot water	ASTORIA	0.014066
1	heat/hot water	241430	heat/hot water	BRONX	0.330075
2	heat/hot water	241430	heat/hot water	BROOKLYN	0.299921
3	heat/hot water	241430	heat/hot water	FLUSHING	0.011353
4	heat/hot water	241430	heat/hot water	JAMAICA	0.013983

In [39]:

```
# Test cell: `norm_above_test`

df_plot_stacked_fraction = cast(df_plot_fraction, key='city_name', value='complaint_frac')

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

show(plot_complaints_stacked_by_city(df_plot_fraction, y='complaint_frac'))

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

df_plot_fraction_soln = canonicalize_tibble(pd.read_csv(get_path('df_plot_stacked_fraction_soln.
df_plot_fraction_yours = canonicalize_tibble(df_plot_stacked_fraction)

nonfloat_cols = df_plot_stacked_fraction.columns.difference(TOP_CITIES)
assert tibbles_are_equivalent(df_plot_fraction_yours[nonfloat_cols],
                              df_plot_fraction_soln[nonfloat_cols])
for c in TOP_CITIES:
    assert all(abs(df_plot_fraction_yours[c] - df_plot_fraction_soln[c]) <= 1e-13), \
        "Fractions for city {} do not match the values we are expecting.".format(c)

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

```
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
BokehDeprecationWarning: 'legend' keyword is deprecated, use explicit 'legend_label', 'legend_fi
'legend_group' keywords instead
```



(http

```
(Passed!)
```

In [40]:

```
df_plot_stacked_fraction
```

Out[40]:

	type	freq	complaint_type	ASTORIA	BRONX	BROOKLYN	FLUSHING	JAMAICA	N YO
0	heat/hot water	241430	heat/hot water	0.014066	0.330075	0.299921	0.011353	0.013983	0.230

1	street condition	124347	street condition	0.013422	0.097405	0.276854	0.027858	0.027279	0.153
2	street light condition	98577	street light condition	0.015551	0.080384	0.193554	0.017895	0.025067	0.021
3	blocked driveway	95080	blocked driveway	0.035107	0.164156	0.361285	0.035833	0.036075	0.028
4	illegal parking	83961	illegal parking	0.017329	0.109408	0.363204	0.022225	0.018735	0.158
5	unsanitary condition	81394	unsanitary condition	0.009706	0.287982	0.352213	0.008526	0.019019	0.191
6	paint/plaster	69929	paint/plaster	0.008080	0.320439	0.346637	0.009839	0.014887	0.205
7	water system	69209	water system	0.016934	0.158433	0.278071	0.020142	0.032236	0.180
8	plumbing	60105	plumbing	0.010948	0.292472	0.335163	0.008502	0.021762	0.215
9	noise	54165	noise	0.020013	0.053540	0.265153	0.014124	0.008585	0.478
10	noise - street/sidewalk	48436	noise - street/sidewalk	0.011128	0.180011	0.266455	0.004996	0.007515	0.428
11	traffic signal condition	44229	traffic signal condition	0.018382	0.039273	0.278528	0.013995	0.023356	0.028
12	noise - commercial	42422	noise - commercial	0.048395	0.063717	0.313116	0.005186	0.012329	0.426
13	door/window	39695	door/window	0.008439	0.309107	0.358383	0.007558	0.017357	0.193
14	water leak	36149	water leak	0.008382	0.302692	0.376497	0.010706	0.015879	0.189
15	dirty conditions	35122	dirty conditions	0.015887	0.140026	0.328199	0.024486	0.032828	0.152
16	sewer	33628	sewer	0.012430	0.102712	0.274949	0.021292	0.068039	0.102
17	sanitation condition	31260	sanitation condition	0.015323	0.113052	0.364811	0.024504	0.041715	0.105
18	electric	30248	electric	0.010480	0.284349	0.372025	0.007108	0.023076	0.167
19	rodent	28454	rodent	0.012371	0.231110	0.318760	0.007240	0.023266	0.239
20	flooring/stairs	27007	flooring/stairs	0.007554	0.300404	0.356093	0.006035	0.016144	0.220
21	general construction/plumbing	26861	general construction/plumbing	0.016306	0.093146	0.352556	0.020960	0.023789	0.261
22	building/use	25807	building/use	0.016856	0.117255	0.293835	0.043012	0.050180	0.070
23	broken muni meter	25428	broken muni meter	0.032838	0.099969	0.204932	0.043849	0.016635	0.432

In [41]:
```
df_plot_fraction_yours
```

Out[41]:

	ASTORIA	BRONX	BROOKLYN	FLUSHING	JAMAICA	NEW YORK	STATEN ISLAND	complaint_type	freq	
0	0.007554	0.300404	0.356093	0.006035	0.016144	0.220239	0.020032	flooring/stairs	27007	flooring/
1	0.008080	0.320439	0.346637	0.009839	0.014887	0.205623	0.014029	paint/plaster	69929	paint/pl
2	0.008382	0.302692	0.376497	0.010706	0.015879	0.189659	0.016100	water leak	36149	water le
3	0.008439	0.309107	0.358383	0.007558	0.017357	0.193097	0.018970	door/window	39695	door/wi
4	0.009706	0.287982	0.352213	0.008526	0.019019	0.191169	0.023220	unsanitary condition	81394	unsanit
5	0.010480	0.284349	0.372025	0.007108	0.023076	0.167647	0.021720	electric	30248	electric
6	0.010948	0.292472	0.335163	0.008502	0.021762	0.215872	0.022993	plumbing	60105	plumbir
7	0.011128	0.180011	0.266455	0.004996	0.007515	0.428917	0.018808	noise - street/sidewalk	48436	noise -
8	0.012371	0.231110	0.318760	0.007240	0.023266	0.239017	0.047515	rodent	28454	rodent
9	0.012430	0.102712	0.274949	0.021292	0.068039	0.102355	0.111603	sewer	33628	sewer
10	0.013422	0.097405	0.276854	0.027858	0.027279	0.153683	0.135822	street condition	124347	street c
11	0.014066	0.330075	0.299921	0.011353	0.013983	0.230067	0.009444	heat/hot water	241430	heat/ho
12	0.015323	0.113052	0.364811	0.024504	0.041715	0.105694	0.088548	sanitation condition	31260	sanitatio
13	0.015551	0.080384	0.193554	0.017895	0.025067	0.021293	0.062865	street light condition	98577	street li
14	0.015887	0.140026	0.328199	0.024486	0.032828	0.152212	0.081943	dirty conditions	35122	dirty co
15	0.016306	0.093146	0.352556	0.020960	0.023789	0.261048	0.050296	general construction/plumbing	26861	general constru
16	0.016856	0.117255	0.293835	0.043012	0.050180	0.070833	0.059480	building/use	25807	building
17	0.016934	0.158433	0.278071	0.020142	0.032236	0.180352	0.089338	water system	69209	water sy
18	0.017329	0.109408	0.363204	0.022225	0.018735	0.158609	0.069520	illegal parking	83961	illegal p
19	0.018382	0.039273	0.278528	0.013995	0.023356	0.028285	0.036469	traffic signal condition	44229	traffic si
20	0.020013	0.053540	0.265153	0.014124	0.008585	0.478962	0.028284	noise	54165	noise
21	0.032838	0.099969	0.204932	0.043849	0.016635	0.432004	0.010304	broken muni meter	25428	broken
22	0.035107	0.164156	0.361285	0.035833	0.036075	0.028502	0.028418	blocked driveway	95080	blocked
23	0.048395	0.063717	0.313116	0.005186	0.012329	0.426430	0.017986	noise - commercial	42422	noise -

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 son 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 f `MM-DD hh:mm:ss`.

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

In [42]:
```
query = '''
  SELECT LOWER(ComplaintType), CreatedDate, UPPER(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[42]:

	LOWER(ComplaintType)	CreatedDate	UPPER(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
...	...	...	...
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 [43]:
```
query = '''
  SELECT CreatedDate, STRFTIME('%H', CreatedDate) AS Hour, LOWER(ComplaintType)
    FROM data
    LIMIT 5
'''
df = pd.read_sql_query (query, disk_engine)
df
```

Out[43]:

	CreatedDate	Hour	LOWER(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 That is, the variables or column names should be {`hour`, `count`} where each observation is the total number of complaints (`count`) that occurr 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 [44]:
```
# 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 [45]:

```
# Test cell: `df_complaints_by_hour_test`

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

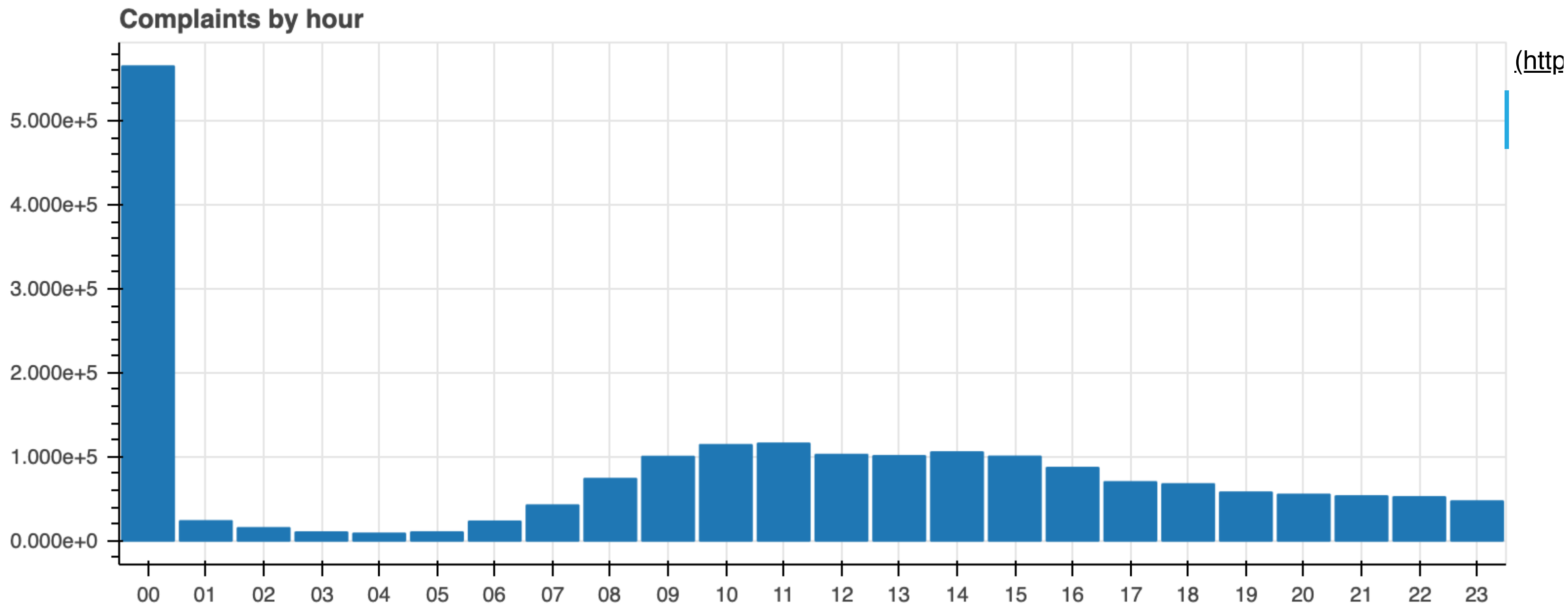
	hour	count
0	0	564703
1	1	23489
2	2	15226
3	3	10164
4	4	8692
5	5	10224
6	6	23051
7	7	42273
8	8	73811
9	9	100077

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 [46]:
```
p = make_barchart(df_complaints_by_hour, 'hour', 'count',
                  {'title': 'Complaints by hour',
                   'plot_width': 800, 'plot_height': 320})
show(p)
```



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 probably strike you as suspicious. Indeed, the reason is that there are some complaints that are dated but with no associated time, which was the data as exactly `00:00:00.000`.

In [47]:
```
query = '''
  SELECT COUNT(*)
    FROM data
    WHERE STRFTIME('%H:%M:%f', CreatedDate) = '00:00:00.000'
'''

pd.read_sql_query(query, disk_engine)
```

Out[47]:

	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 `count}` and whose observations are the number of noise complaints that occurred during a given `hour`. Consider a "noise complaint" to be an 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 [48]:
```
### BEGIN SOLUTION
query = '''
  SELECT STRFTIME('%H %M %S %s', CreatedDate) AS hour,
         COUNT(*) AS count
    FROM data
    WHERE (LOWER(ComplaintType) like '%noise%')
      AND (STRFTIME('%H %M %S %s', CreatedDate) <> '00 00 00')
```

```
    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 00 00 1411603200	1
1	00 00 00 1412121600	1
2	00 00 00 1412380800	1
3	00 00 00 1412812800	1
4	00 00 00 1412985600	1
...	...	...
165216	23 59 58 1416959998	1
165217	23 59 59 1416873599	1
165218	23 59 59 1426982399	1
165219	23 59 59 1433807999	1
165220	23 59 59 1438991999	1

165221 rows × 2 columns

In [52]:
```
# Test cell: `df_noisy_by_hour_test`

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

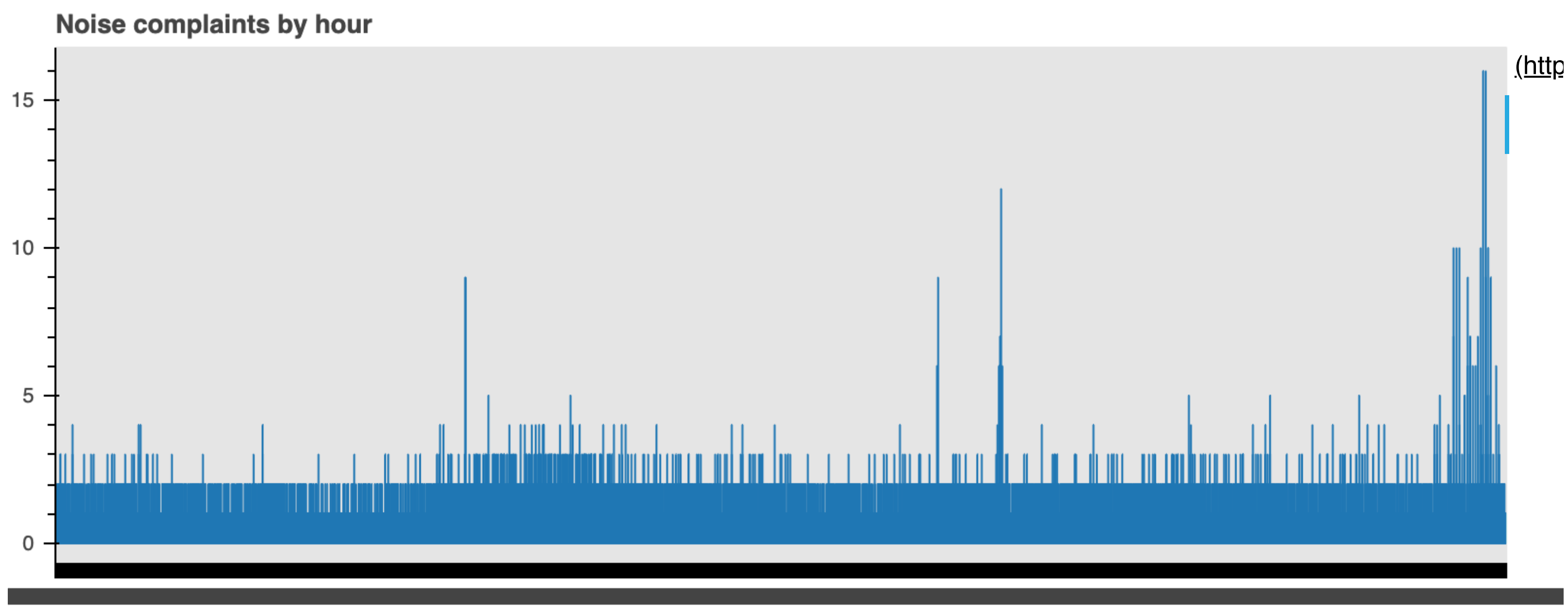
```
Reading instructor's solution...
```

	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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-52-a042707beab7> in <module>
      9 df_noisy_by_hour_norm = df_noisy_by_hour.copy()
     10 df_noisy_by_hour_norm['hour'] = \
---> 11     df_noisy_by_hour_norm['hour'].apply(int)
     12 assert tibbles_are_equivalent (df_noisy_by_hour_norm,
     13                               df_noisy_by_hour_soln)

/usr/lib/python3.7/site-packages/pandas/core/series.py in apply(self, func, convert_dtype, args,
   4198             else:
   4199                 values = self.astype(object)._values
-> 4200                 mapped = lib.map_infer(values, f, convert=convert_dtype)
   4201
   4202         if len(mapped) and isinstance(mapped[0], Series):

pandas/_libs/lib.pyx in pandas._libs.lib.map_infer()

ValueError: invalid literal for int() with base 10: '00 00 00 1411603200'
```

In [50]:
```
p = make_barchart(df_noisy_by_hour, 'hour', 'count',
                  {'title': 'Noise complaints by hour',
                   'plot_width': 800, 'plot_height': 320})
show(p)
```



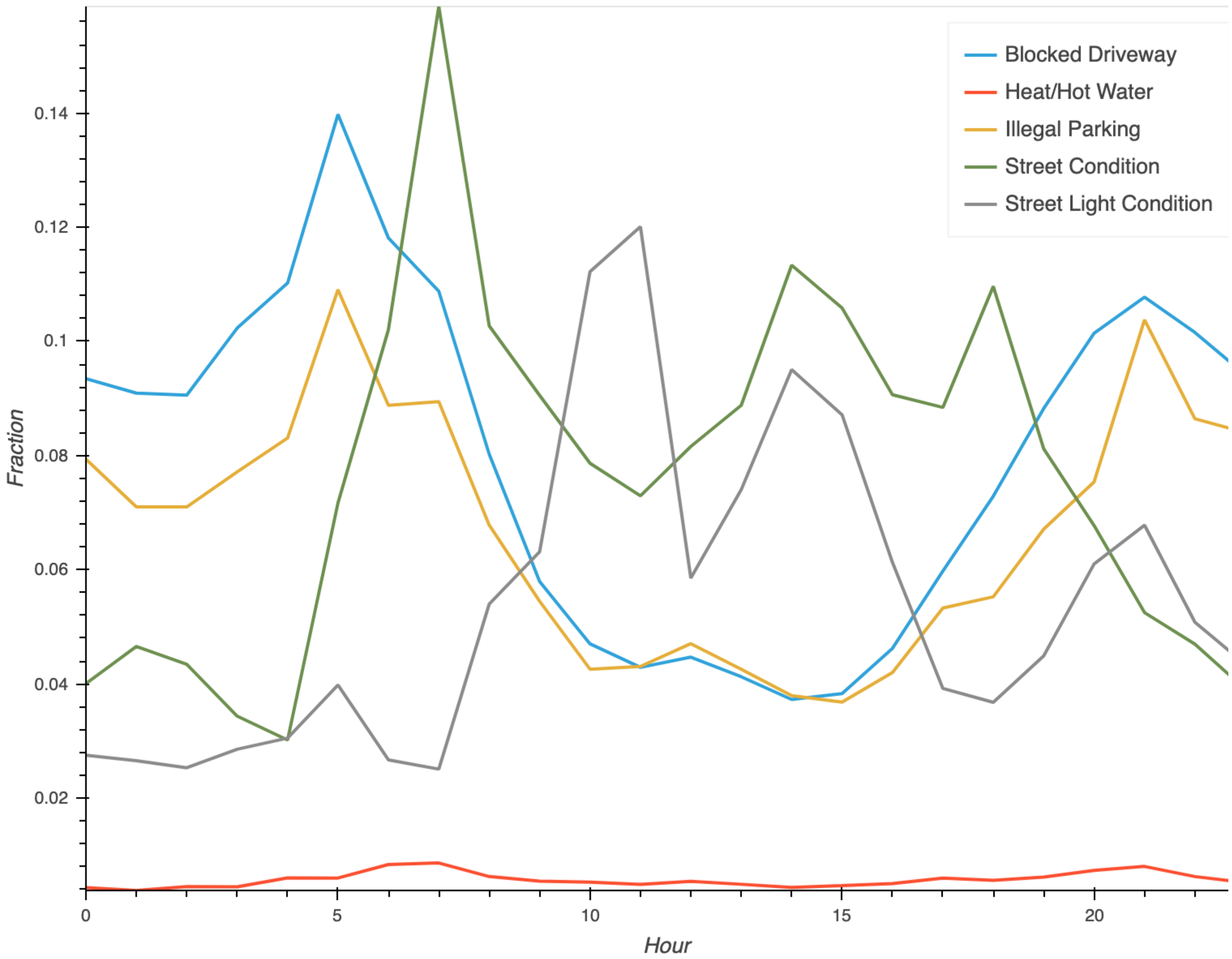
Exercise 8 (ungraded). Create a line chart to show the fraction of complaints (y-axis) associated with each hour of the day (x-axis), with each c shown as a differently colored line. Show just the top 5 complaints (`top_complaints[:5]`). Remember to exclude complaints with a zero-tim `00:00:00.000`).

> **Note.** This exercise is ungraded but if your time permits, please give it a try! Feel free to discuss your approaches to this problem on th discussion forums (but do try to do it yourself first). One library you may find useful to try out is holoviews (http://holoviews.org/index.h (http://holoviews.org/index.html))

In [51]:
```
import holoviews as hv
hv.extension('bokeh')
from holoviews import Bars

### BEGIN SOLUTION
query1 = '''
  SELECT STRFTIME('%H', CreatedDate) AS hour, LOWER(ComplaintType) AS complaint_type, COUNT(*) A
    FROM data
    WHERE CreatedDate NOT LIKE "%00:00:00.000%"
    GROUP BY hour, complaint_type
'''

query2 = '''
  SELECT COUNT(*) AS freq, STRFTIME('%H', CreatedDate) AS hour
    FROM data
    WHERE CreatedDate NOT LIKE "%00:00:00.000%"
    GROUP BY hour
'''

query3 = '''
  SELECT LOWER(ComplaintType) AS complaint_type, COUNT(*) AS num
    FROM data
    GROUP BY complaint_type
    ORDER BY -num
    LIMIT 5
'''
```

```
df_query1 = pd.read_sql_query(query1, disk_engine)
df_query2 = pd.read_sql_query(query2, disk_engine)
df_query3 = pd.read_sql_query(query3, disk_engine)

A = df_query1.merge(df_query3, on=['complaint_type'],how='inner')
B = A.merge(df_query2, on=["hour"],how='inner')
B = B[['freq','hour','complaint_type','count']]

df_cast = cast(B, key='complaint_type', value='count')

df_new = df_cast.copy()

for i in df_new.columns[2:]:
    df_new[i] = df_new[i]/df_new["freq"]

df_top5_frac = df_new.copy()
del df_top5_frac["freq"]

%opts Overlay [width=800 height=600 legend_position='top_right' xlabel="Hour" ylabel="Fraction"

hv.Curve((df_top5_frac['blocked driveway']), label='Blocked Driveway') * \
hv.Curve((df_top5_frac['heat/hot water']), label='Heat/Hot Water') * \
hv.Curve((df_top5_frac['illegal parking']), label='Illegal Parking') * \
hv.Curve((df_top5_frac['street condition']), label='Street Condition') * \
hv.Curve((df_top5_frac['street light condition']), label='Street Light Condition')
### END SOLUTION
```

Out[51]:



Learn more

- Find more open data sets on Data.gov (https://data.gov) and NYC Open Data (https://nycopendata.socrata.com)
- Learn how to setup MySql with Pandas and Plotly (http://moderndata.plot.ly/graph-data-from-mysql-database-in-python/)
- Big data workflows with HDF5 and Pandas (http://stackoverflow.com/questions/14262433/large-data-work-flows-using-pandas)

< Previous

Next Up: Practice Problems for Midterm 1: Topics 1-5 >
1 min + 1 activity

edX

[About]
[Affiliates]
[edX for Business]
[Open edX]
[Careers]
[News]

Legal

[Terms of Service & Honor Code]
[Privacy Policy]
[Accessibility Policy]
[Trademark Policy]
[Sitemap]

Connect

[Blog]
[Contact Us]
[Help Center]
[Media Kit]
[Donate]

GET IT ON Google play

Download on the App Store