# WorkingWithStructuredData

February 21, 2015

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np
        %precision 4
        import os, sys, glob
```

## 0.1 Using SQLite3

Willl change this to use the same example for queries and schema design

- Subjects - Ann, Bob, Charlie
- Tests - Liver function, Complete blood count
- Test parameters - AST, ALT, RBC, platelets, WBC (may perform all or only subset of parameters)
- Diffrent number of visits, different number of tests per visit

### 0.1.1 Working example dataset

This data contains the survival time after receiving a heart transplant, the age of the patient and whether or not the survival time was censored

- Number of Observations - 69
- Number of Variables - 3

Variable name definitions:: * death - Days after surgery until death * age - age at the time of surgery * censored - indicates if an observation is censored. 1 is uncensored

```
In [2]: import statsmodels.api as sm
        heart = sm.datasets.heart.load_pandas().data
        heart.take(np.random.choice(len(heart), 6))
```

```
Out[2]:     survival  censors   age
        66       110        0  23.7
        24      1367        0  48.6
        30       897        1  46.1
        67        13        0  28.9
        49       499        0  52.2
        35       322        1  48.1
```

```
In [3]: import sqlite3
        conn = sqlite3.connect('heart.db')
```

#### 0.1.2 Creating and populating a table

```
In [4]: c = conn.cursor()

        c.execute('''CREATE TABLE IF NOT EXISTS transplant
                     (survival integer, censors integer, age real)''')

        c.executemany("insert into transplant(survival, censors, age) values (?, ?, ?)", heart.values);
```

#### 0.1.3 SQL queries

SQL Queries take the form

```
select (distinct) ... from ... (limit ...)
where ...
groupby ..
order by ...
```

where most of the query apart from the `select ... from ...` are optional.

##### Selecting all columns, first 10 rows

```
In [5]: for row in c.execute('''select * from transplant limit 5;'''):
            print row
```

```
(15, 1, 54.3)
(3, 1, 40.4)
(624, 1, 51.0)
(46, 1, 42.5)
(127, 1, 48.0)
```

##### Using where to filter rows

```
In [6]: # only find censored data for subjects < 40 years old
        for row in c.execute('''
        select * from transplant
        where censors=0 and age < 40 limit 5;'''):
            print row
```

```
(1775, 0, 33.3)
(1106, 0, 36.8)
(875, 0, 38.9)
(815, 0, 32.7)
(592, 0, 26.7)
```

##### Using SQL functions

```
In [7]: for row in c.execute('''select count(*), avg(age) from transplant where censors=0 and age < 40;
            print row
```

```
(9, 31.433333333333333)
```

**Using groupby to find number of cnesored and uncensored subjects and thier average age**

```
In [8]: query = '''
        select censors, count(*), avg(age) from transplant
        group by censors;
        '''
        for row in c.execute(query):
            print row
```

```
(0, 24, 41.729166666666664)
(1, 45, 48.484444444444456)
```

**Using having to filter grouped results**

```
In [9]: query = '''
        select censors, count(*), avg(age) from transplant
        group by censors
        having avg(age) < 45;
        '''
        for row in c.execute(query):
            print row
```

```
(0, 24, 41.729166666666664)
```

**Using order by to sort results**

```
In [10]: query = '''
         select * from transplant
         where age < 40
         order by age desc;
         '''
         for row in c.execute(query):
             print row
```

```
(875, 0, 38.9)
(1106, 0, 36.8)
(44, 1, 36.2)
(1, 0, 35.2)
(1775, 0, 33.3)
(815, 0, 32.7)
(12, 1, 29.2)
(13, 0, 28.9)
(592, 0, 26.7)
(167, 0, 26.7)
(110, 0, 23.7)
(228, 1, 19.7)
```

**Reading into a numpy structured array**

```
In [11]: result = c.execute(query).fetchall()
         arr = np.fromiter(result, dtype='i4,i4,f4')
         arr.dtype.names = ['survival', 'censors', 'age']
         print '\n'.join(map(str, arr))
```

```
(875, 0, 38.900001525878906)
(1106, 0, 36.79999923706055)
(44, 1, 36.20000076293945)
(1, 0, 35.20000076293945)
(1775, 0, 33.29999923706055)
(815, 0, 32.70000076293945)
(12, 1, 29.200000762939453)
(13, 0, 28.899999618530273)
(592, 0, 26.700000762939453)
(167, 0, 26.700000762939453)
(110, 0, 23.700000762939453)
(228, 1, 19.700000762939453)
```

#### Reading into a numpy regular array

```
In [12]: from itertools import chain
         result = c.execute(query).fetchall()
         arr = np.fromiter(chain.from_iterable(result), dtype=np.float)
         print arr.reshape(-1,3)
```

```
[[  8.7500e+02   0.0000e+00   3.8900e+01]
 [  1.1060e+03   0.0000e+00   3.6800e+01]
 [  4.4000e+01   1.0000e+00   3.6200e+01]
 [  1.0000e+00   0.0000e+00   3.5200e+01]
 [  1.7750e+03   0.0000e+00   3.3300e+01]
 [  8.1500e+02   0.0000e+00   3.2700e+01]
 [  1.2000e+01   1.0000e+00   2.9200e+01]
 [  1.3000e+01   0.0000e+00   2.8900e+01]
 [  5.9200e+02   0.0000e+00   2.6700e+01]
 [  1.6700e+02   0.0000e+00   2.6700e+01]
 [  1.1000e+02   0.0000e+00   2.3700e+01]
 [  2.2800e+02   1.0000e+00   1.9700e+01]]
```

### 0.1.4 Working wiht multiple tables in SQL

We will consturct a new database with 2 tables to illustrate the concept of joins.

```
In [13]: conn1 = sqlite3.connect('samples.db')
         c1 = conn1.cursor()

         c1.execute(
         '''
         CREATE TABLE IF NOT EXISTS t1(
           ID TEXT,
           Name TEXT,
           Value Real);
         ''')

         c1.execute('''
         CREATE TABLE IF NOT EXISTS t2(
           ID TEXT,
           Name TEXT,
           Value Real,
           Age INTEGER);
         ''');
```

```
from string import ascii_lowercase
for i in range(5):
    c1.execute('''insert into t1(ID, Name, Value) values (%d, '%s', %.2f)''' % (i, ascii_lowerc
    c1.execute('''insert into t2(ID, Name, Value, Age) values (%d, '%s', %.2f, %d)''' % (i*2, a
```

#### Cartesian product

```
In [14]: # Without specifiying a join, the result is all possible combinations
         query = '''
         select t1.ID, t2.ID from t1, t2;
         '''
         for row in c1.execute(query):
             print row
```

```
(u'0', u'0')
(u'0', u'2')
(u'0', u'4')
(u'0', u'6')
(u'0', u'8')
(u'1', u'0')
(u'1', u'2')
(u'1', u'4')
(u'1', u'6')
(u'1', u'8')
(u'2', u'0')
(u'2', u'2')
(u'2', u'4')
(u'2', u'6')
(u'2', u'8')
(u'3', u'0')
(u'3', u'2')
(u'3', u'4')
(u'3', u'6')
(u'3', u'8')
(u'4', u'0')
(u'4', u'2')
(u'4', u'4')
(u'4', u'6')
(u'4', u'8')
```

#### Inner joins

```
In [15]: # Inner join (intersection)
         query = '''
         select t1.ID, t2.ID, t1.value, t2.value, t1.value * t2.value from t1, t2
         where t1.ID = t2.ID;
         '''
         for row in c1.execute(query):
             print row
```

```
(u'0', u'0', 0.0, 5.0, 0.0)
(u'2', u'2', 4.0, 6.0, 24.0)
(u'4', u'4', 16.0, 9.0, 144.0)
```

```
In [16]: # left join keeps all values from the left table (t2)
         # and values from the right (t1) where there is a match
         query = '''
         select t1.id, t2.ID, t1.value, t2.value from t2 left join t1 on t1.ID = t2.ID
         '''
         for row in c1.execute(query):
             print row
```

```
(u'0', u'0', 0.0, 5.0)
(u'2', u'2', 4.0, 6.0)
(u'4', u'4', 16.0, 9.0)
(None, u'6', None, 14.0)
(None, u'8', None, 21.0)
```

```
In [17]: # same join but we swtich left and right tables
         query = '''
         select t1.ID, t2.ID, t1.value, t2.value from t1 left join t2 on t1.ID = t2.ID
         '''
         for row in c1.execute(query):
             print row
```

```
(u'0', u'0', 0.0, 5.0)
(u'1', None, 1.0, None)
(u'2', u'2', 4.0, 6.0)
(u'3', None, 9.0, None)
(u'4', u'4', 16.0, 9.0)
```

### Self-joins

```
In [18]: # we can join a table to itself by using aliases
         # lets add a few more rows to t1 which may have the same id and name but different values

         for i in range(5):
             c1.execute('''insert into t1(ID, Name, Value) values (%d, '%s', %.2f)''' % (i, ascii_lowerc

         for row in c1.execute('select * from t1;'):
             print row
```

```
(u'0', u'a', 0.0)
(u'1', u'b', 1.0)
(u'2', u'c', 4.0)
(u'3', u'd', 9.0)
(u'4', u'e', 16.0)
(u'0', u'a', 0.0)
(u'1', u'b', 1.0)
(u'2', u'c', 8.0)
(u'3', u'd', 27.0)
(u'4', u'e', 64.0)
```

```
In [19]: # Now use a self-join to find paired values for the same ID and name

         query = '''
         select t1a.ID, t1a.Name, t1a.value, t1b.value from t1 as t1a, t1 as t1b
         where t1a.Name = t1b.Name and t1a.Value < t1b.Value
         order by t1a.ID ASC;
```

```
'''
for row in c1.execute(query):
    print row
```

```
(u'2', u'c', 4.0, 8.0)
(u'3', u'd', 9.0, 27.0)
(u'4', u'e', 16.0, 64.0)
```

### 0.1.5 Basic concepts of database normalization

In which we convert a dataframe into a normalized database.

```
In [127]: names = ['ann', 'bob', 'ann', 'bob', 'carl', 'delia', 'ann']
          tests = ['wbc', 'wbc', 'rbc', 'rbc', 'wbc', 'rbc', 'platelets']
          values1 = [10, 11.2, 300, 204, 9.8, 340, 125]
          values2 = [10.6, 13.2, 322, 214, 10.3, 343, 145]
          df = pd.DataFrame([names, tests, values1, values2]).T
          df.columns = ['names', 'tests', 'values1', 'values2']
          df
```

Out[127]:

| | names | tests | values1 | values2 |
|---|---|---|---|---|
| 0 | ann | wbc | 10 | 10.6 |
| 1 | bob | wbc | 11.2 | 13.2 |
| 2 | ann | rbc | 300 | 322 |
| 3 | bob | rbc | 204 | 214 |
| 4 | carl | wbc | 9.8 | 10.3 |
| 5 | delia | rbc | 340 | 343 |
| 6 | ann | platelets | 125 | 145 |

```
In [129]: # names are put into their own table so there is no dubplication

          name_table = pd.DataFrame(df['names'].unique(), columns=['name'])
          name_table['name_id'] = name_table.index
          columns = ['name_id', 'name']
          name_table[columns]
```

Out[129]:

| | name_id | name |
|---|---|---|
| 0 | 0 | ann |
| 1 | 1 | bob |
| 2 | 2 | carl |
| 3 | 3 | delia |

```
In [130]: # tests are put inot their own table so there is no duplication

          test_table = pd.DataFrame(df['tests'].unique(), columns=['test'])
          test_table['test_id'] = test_table.index
          columns = ['test_id', 'test']
          test_table[columns]
```

Out[130]:

| | test_id | test |
|---|---|---|
| 0 | 0 | wbc |
| 1 | 1 | rbc |
| 2 | 2 | platelets |

```
In [132]: # the values1 and values2 correspond to visit 1 and 2, so
          # we create a visits table
```

```
visit_table = pd.DataFrame([1,2], columns=['visit'])
visit_table['visit_id'] = visit_table.index
columns = ['visit_id', 'visit']
visit_table[columns]
```

Out[132]:

| | visit_id | visit |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 2 |

In [97]:

```
# finally, we link each value to a triple(name_id, test_id, visit_id)

value_table = pd.DataFrame([
    [0,0,0,10], [1,0,0,11.2], [0,1,0,300], [1,1,0,204], [2,0,0,9.8], [3,1,0,340], [0,2,0,125],
   [0,0,1,10.6], [1,0,1,13.2], [0,1,1,322], [1,1,1,214], [2,0,1,10.3], [3,1,1,343], [0,2,1,145
], columns=['name_id', 'test_id', 'visit_id', 'value'])
value_table
```

Out[97]:

| | name_id | test_id | visit_id | value |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 10.0 |
| 1 | 1 | 0 | 0 | 11.2 |
| 2 | 0 | 1 | 0 | 300.0 |
| 3 | 1 | 1 | 0 | 204.0 |
| 4 | 2 | 0 | 0 | 9.8 |
| 5 | 3 | 1 | 0 | 340.0 |
| 6 | 0 | 2 | 0 | 125.0 |
| 7 | 0 | 0 | 1 | 10.6 |
| 8 | 1 | 0 | 1 | 13.2 |
| 9 | 0 | 1 | 1 | 322.0 |
| 10 | 1 | 1 | 1 | 214.0 |
| 11 | 2 | 0 | 1 | 10.3 |
| 12 | 3 | 1 | 1 | 343.0 |
| 13 | 0 | 2 | 1 | 145.0 |

At the end of the normalizaiton, we have gone from 1 dataframe with multiple redundancies to 4 tables with unique entries in each row. This organization helps maintain data integrity and is necesssary for effficeincy as the number of test values grows, possibly into millions of rows. As we have seen, we can use SQL queries to recreate the origianl dataformat if that is more convenient for analysis.

#### 0.1.6 Using HDF5

When your data consists of many numerical and matrices, each of which is relatively independent, relational databases offer little benefit, and it is more efficient to use HDF5 (Hierarchical Data Format) for storage. For example, your data may come from a simulation which generates a 3D matrix and a list of count data at every iteration.

In [44]:

```
import h5py

f = h5py.File('simulation.h5')
```

In [45]:

```
for i in range(10): # iterations in simulation
    xs = np.random.random((100,100,100))
    ys = np.random.randint(0,100,(i+1)*10)
    group = f.create_group('Iteration%03d' % i)
    group.create_dataset('xs', data=xs)
    group.create_dataset('ys', data=ys)
```

```
In [46]: f.keys()
```

```
Out[46]: [u'Iteration000',
          u'Iteration001',
          u'Iteration002',
          u'Iteration003',
          u'Iteration004',
          u'Iteration005',
          u'Iteration006',
          u'Iteration007',
          u'Iteration008',
          u'Iteration009']
```

```
In [47]: f['Iteration008'].keys()
```

```
Out[47]: [u'xs', u'ys']
```

```
In [48]: g8 = f['Iteration008']
         print g8['xs'][2:5,2:5,2:5]
         print g8['ys'][-10:]
```

```
[[[ 0.0367  0.2883  0.5562]
  [ 0.9494  0.5614  0.1159]
  [ 0.8887  0.7396  0.891 ]]

 [[ 0.7552  0.1539  0.216 ]
  [ 0.6671  0.4682  0.9107]
  [ 0.5565  0.5443  0.1665]]

 [[ 0.3972  0.1205  0.9487]
  [ 0.7874  0.3466  0.2818]
  [ 0.1248  0.0161  0.6898]]]
[37 69  5 15 10 44 20 73 74 24]
```

### 0.1.7 Interfacing withPandas

```
In [26]: import pandas as pd
```

```
In [27]: df = pd.read_sql('select * from transplant;', conn)
```

```
In [28]: df.take(np.random.randint(0, len(df), 6))
```

```
Out[28]:     survival  censors   age
         8         23        1  56.9
         38       815        0  32.7
         12       730        1  58.4
         58       339        0  54.4
         53       439        0  52.9
         27       994        1  48.6
```

```
In [29]: df1 = pd.read_sql('select t1.name, t2.value, t2.age from t1, t2 where t1.name = t2.name;', conn
```

```
In [30]: df1
```

```
Out[30]:   Name  Value  Age
         0    a      5    0
         1    c      6   10
```

```
2    e      9   20
3    a      5    0
4    c      6   10
5    e      9   20
```

```
In [31]: c.close()
         c1.close()
         conn.close()
         conn1.close()
```

```
In [60]: store = pd.HDFStore('dump.h5')
         store['transplant'] = df
         store['tables'] = df1
         store.close()
```

```
/Users/cliburn/anaconda/lib/python2.7/site-packages/pandas/io/pytables.py:2453: PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->unicode,key->block2_values] [items->['Name']]

  warnings.warn(ws, PerformanceWarning)
```

```
In [62]: transplant_df = pd.read_hdf('dump.h5', 'transplant')
         transplant_df.take(np.random.randint(0, len(df), 6))
```

Out[62]:

| | survival | censors | age |
|---|---|---|---|
| 50 | 305 | 0 | 49.3 |
| 3 | 46 | 1 | 42.5 |
| 0 | 15 | 1 | 54.3 |
| 22 | 1 | 1 | 41.5 |
| 47 | 63 | 1 | 56.4 |
| 19 | 1549 | 0 | 40.6 |

```
In [64]: table_df = pd.read_hdf('dump.h5', 'tables')
         table_df
```

Out[64]:

| | Name | Value | Age |
|---|---|---|---|
| 0 | a | 5 | 0 |
| 1 | c | 6 | 10 |
| 2 | e | 9 | 20 |
| 3 | a | 5 | 0 |
| 4 | c | 6 | 10 |
| 5 | e | 9 | 20 |

```
In [65]: store
```

```
Out[65]: <class 'pandas.io.pytables.HDFStore'>
         File path: dump.h5
         File is CLOSED
```

```
In [66]: store = pd.HDFStore('dump.h5')
```

```
In [67]: store.keys()
```

```
Out[67]: ['/tables', '/transplant']
```

```
In [68]: store.close()
```

```
In [ ]:
```