# Beyond the `dict`

## Python tools for data wrangling

Imran S. Haque
9 November 2013
PyData NYC

@imranshaque
@counsyl

github.com/ihaque
github.com/counsyl

https://github.com/ihaque/pydata_nyc_2013

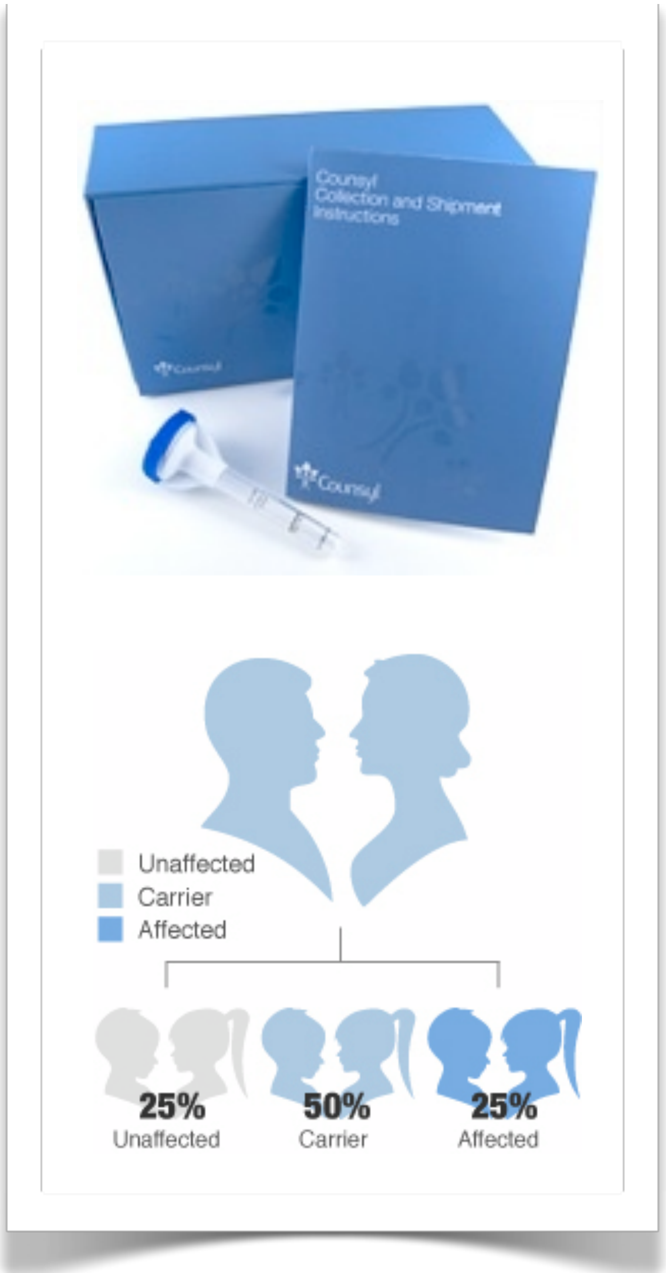
# Who Are You?

Counsyl performs carrier screening for
inherited genetic disorders
on >3% of all births in the USA.

# Who Are You?

Counsyl performs carrier screening for inherited genetic disorders on >3% of all births in the USA.

Almost all backend code at Counsyl is in Python: it can do everything from hardcore numerics to webapps.



# Who Are You?

Counsyl performs carrier screening for inherited genetic disorders on >3% of all births in the USA.

Almost all backend code at Counsyl is in Python: it can do everything from hardcore numerics to webapps.

Medical genomics requires dealing with lots of structured (numerical, tabular) and semi-structured (crappy log file) data.

Wrangling is the most tedious part of data science.

Wrangling is the most tedious part of data science.

Parsing

Wrangling is the most tedious part of data science.

Parsing Querying

Wrangling is the most tedious part of data science.

Parsing

Querying

Storing

# Today's Menu

Three courses:

- Literate parsing: `csv`, `sheets`
- Querying beyond the dict: `sqlite3`, `pony.orm`
- Efficient numerical storage: `tables`

# Today's Menu

Three courses:

- Literate parsing: **`csv`**, **`sheets`**
- Querying beyond the dict: **`sqlite3`**, **`pony.orm`**
- Efficient numerical storage: **`tables`**

Off the menu:

- **`pandas`**
- "Real" databases: **`psycopg2`**, **`MySQLdb`**
- Heavyweight ORMs: **`django`**, **`sqlalchemy`**

# Let’s parse the One True Format.

Let’s parse the One True Format.

CSV

# hapmap.txt

```
rs# refallele_freq otherallele_freq
rs6423165 0.143 0.857
rs6608381 0.464 0.536
rs6644972 0.354 0.646
...
```

# Basic CSV

```
def read_csv(filename):
  with open(filename, 'r') as csvfile:
    for line in csvfile:
      yield line.strip().split(' ')
```

```
['rs#', 'refallele_freq', 'otherallele_freq']
['rs6423165', '0.143', '0.857']
...
```

Counsyl

# Basic CSV

```
def
    w                                                    :
```


```
['rs#'                                                  freq']
['rs64
...
```

# stdlib: csv

```
import csv
def read_csv(filename):
  with open(filename, 'r') as csvfile:
    for row in csv.reader(csvfile, delimiter=' '):
      yield row
```

```
['rs#', 'refallele_freq', 'otherallele_freq']
['rs6423165', '0.143', '0.857']
...
```

Counsyl

# stdlib: csv

```
import csv
def read_csv(filename):
  with open(filename, 'r') as csvfile:
    for row in csv.DictReader(csvfile,
                              delimiter=' '):
      yield row
```

```
{'rs#': 'rs6423165',
 'refallele_freq': '0.143',
 'otherallele_freq': '0.857'}
...
```

Counsyl

# Intro to sheets

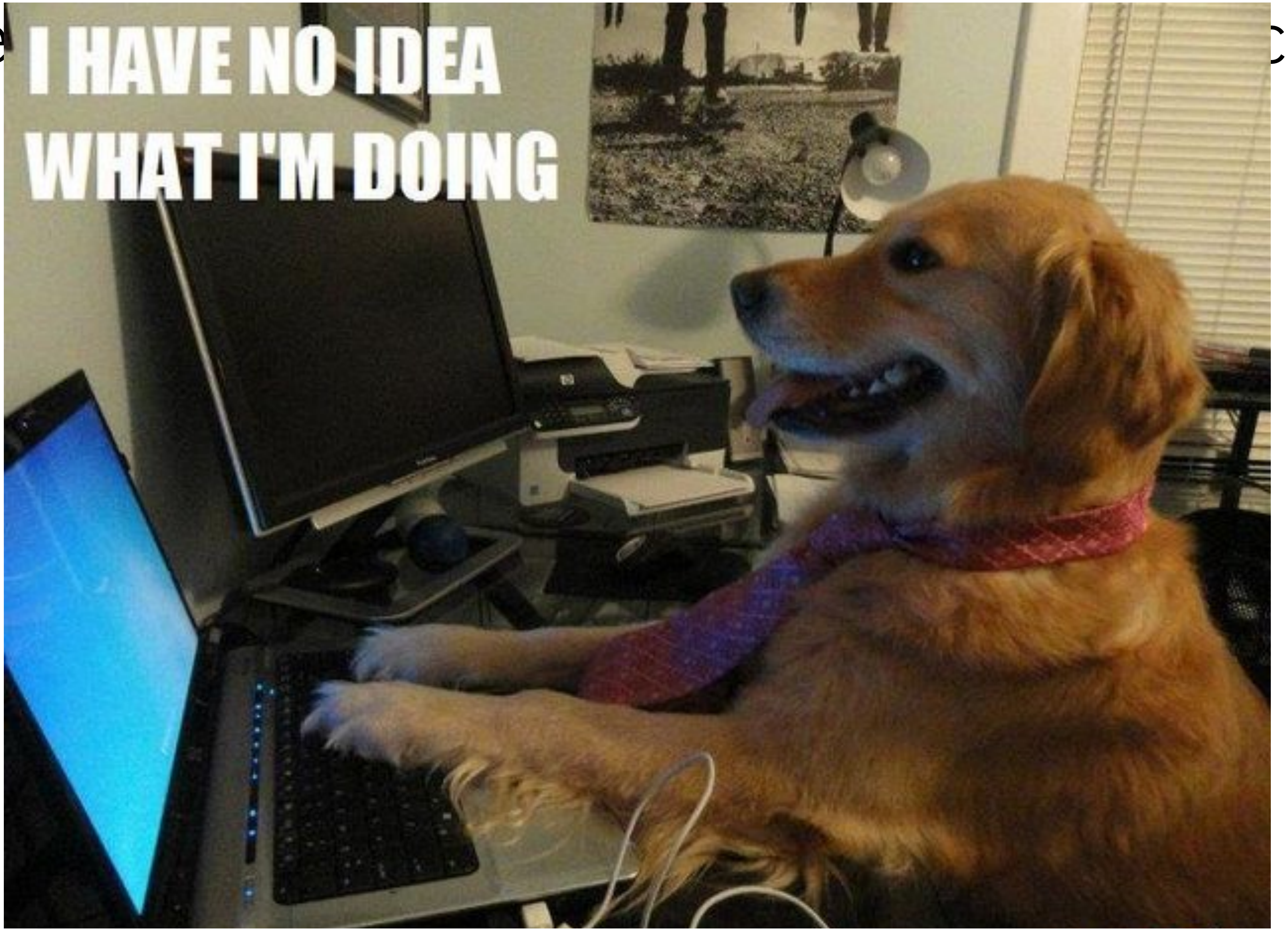
Literate schemas for text data, by Marty Alchin

*CSV* $\longleftrightarrow$ *objects*

https://github.com/gulopine/sheets

# Intro to sheets

Lite … chin




https://github.com/gulopine/sheets

# Intro to sheets

Literate schemas for text data, by Marty Alchin

*CSV ↔ objects*

```
class HapMapRow(Row):
    Dialect = Dialect(has_header_row=True,
                      delimiter=' ')
    rsid = StringColumn()
    ref_freq = FloatColumn()
    alt_freq = FloatColumn()
```

https://github.com/gulopine/sheets

Counsyl

# CSV with sheets

```
def parse_csv(filename):
  with open(filename, 'r') as csvfile:
    for row in HapMapRow.reader(csvfile):
      yield row
```

```
HapMapRow(rsid='rs6423165', ref_freq=0.143,
              alt_freq=0.857)
HapMapRow(rsid='rs6608381', ref_freq=0.464,
              alt_freq=0.536)
                  ...
```

Counsyl

# Let’s query the data.

# Simple Queries

```
next(row for row in rows
        if row.rsid == 'rs6423165').ref_freq
```

Counsyl

# Simple Queries

```
next(row for row in rows
     if row.rsid == 'rs6423165').ref_freq
```

```
rsid2ref_freq = {
    row.rsid: row.ref_freq
    for row in read_csv('hapmap.txt')
}

rsid2ref_freq['rs6423165']
```

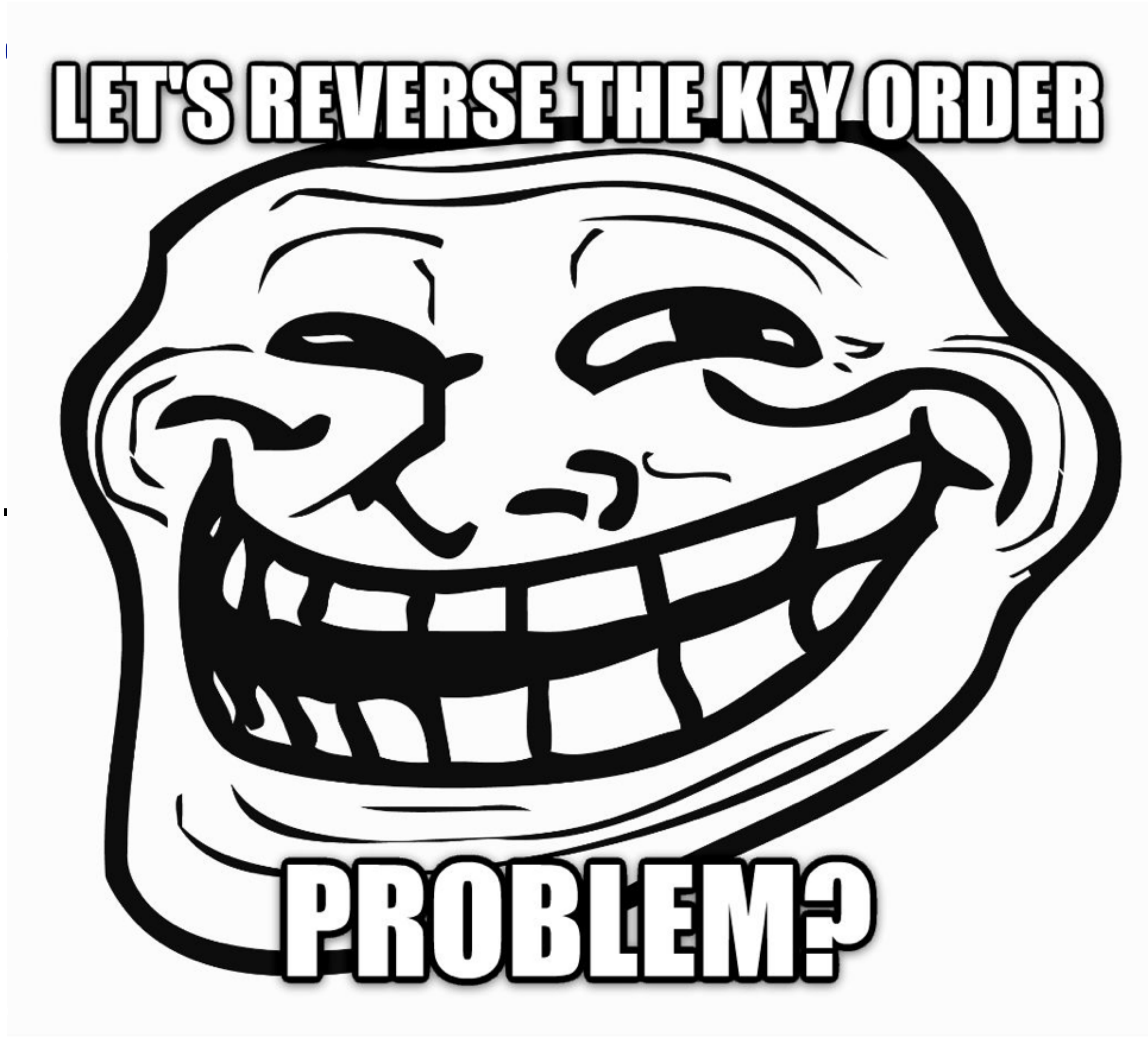
# Multiple-Key Queries

```
rows = read_csv('hapmap.txt')

pop2rsid2ref_freq = {
    pop: {
        row.rsid: row.ref_freq
        for row in rows
        if row.pop == pop
    } for pop in {row.pop for row in rows}
}

pop2rsid2ref_freq['YRI']['rs6423165']
```

Counsyl

# Multiple-Key Queries

```
rows = r
pop2rsid
    pop:
    } fo                         in rows}
}
pop2rsid                         5']
```




Counsyl

`dict` construction is equivalent to materializing *one* query plan.

This is fragile!

# Intro to sqlite3

Embedded, serverless SQL database
*in the Python standard library*

# Intro to sqlite3

Embedded, serverless SQL database

*in the Python standard library*

```python
import sqlite3
db = sqlite3.connect(':memory:')
...
for row in db.execute('SELECT * FROM ...'):
```

# sqlite3 example

```
db.execute('''CREATE TABLE hapmap
              (rsid VARCHAR,
               ref_freq REAL,
               alt_freq REAL);''')

for row in read_csv(hapmap_file):
  db.execute('''INSERT INTO hapmap
                (rsid, ref_freq, alt_freq)
                VALUES (?, ?, ?);''',
             (row.rsid, row.ref_freq, row.alt_freq))

db.execute('SELECT rsid FROM hapmap '
           'WHERE alt_freq > 0.01;')
```

Counsyl

# sqlite3 example

```
db.execute('''CREATE TABLE hapmap
              (rsid VARCHAR,
               ref_freq REAL,
               alt_freq REAL);''')

for row in read_csv(hapmap_file):
  db.execute('''INSERT INTO hapmap
                (rsid, ref_freq, alt_freq)
                VALUES (?, ?, ?);''',
             (row.rsid, row.ref_freq, row.alt_freq))

db.execute('SELECT rsid FROM hapmap '
           'WHERE alt_freq > 0.01;')
```

# sqlite3 example

```
db.execute('''CREATE TABLE hapmap
              (rsid VARCHAR,
               ref_freq REAL,
               alt_freq REAL);''')

for row in read_csv(hapmap_file):
  db.execute('''INSERT INTO hapmap
                (rsid, ref_freq, alt_freq)
                VALUES (?, ?, ?);''',
             (row.rsid, row.ref_freq, row.alt_freq))

db.execute('SELECT rsid FROM hapmap '
           'WHERE alt_freq > 0.01;')
```

# sqlite3 example

```
db.exe

for ro
   db.e
                                            freq))

db.exe
```

ORMs let you map objects into a database.

`sheets`: CSV $\leftrightarrow$ objects

ORM: objects $\leftrightarrow$ database

OR nto


C e

# pony-orm example

```
from pony.orm import Database, Required

db = Database('sqlite', ':memory:')
class HapMapAllele(db.Entity):
  rsid = Required(str)
  ref_freq = Required(float)
  alt_freq = Required(float)

...
from pony.orm import db_session, select
with db_session:
    select(allele for allele in HapMapAllele
           if allele.alt_freq > 0.01)[:]
```

# pony-orm example

```
from pony.orm import Database, Required

db = Database('sqlite', ':memory:')
class HapMapAllele(db.Entity):
  rsid = Required(str)
  ref_freq = Required(float)
  alt_freq = Required(float)


...
from pony.orm import db_session, select
with db_session:
    select(allele for allele in HapMapAllele
              if allele.alt_freq > 0.01)[:]
```

Counsyl

# pony-orm example

```
from
from

db =
clas
  rs
  re
  al

...
with
                                                lele
```

# pony-orm example

```
class HapMapAllele(db.Entity):
  rsid = Required(str)
  ref_freq = Required(float)
  alt_freq = Required(float)
```

↕ This looks familiar!

```
class HapMapRow(Row):
  Dialect = Dialect(has_header_row=True,
                    delimiter=' ')
  rsid = StringColumn()
  ref_freq = FloatColumn()
  alt_freq = FloatColumn()
```

Counsyl

# pony-blanket example

```
# ...sheets stuff up here...
from pony_blanket import csv_to_db
db, models = csv_to_db(
    {'hapmap.txt': HapMapAllele})

from pony.orm import db_session, select
with db_session:
  print len(select(x.rsid
                   for x in models[HapMapAllele]
                   if x.alt_freq > 0.01))
```

Counsyl

# pony-blanket example

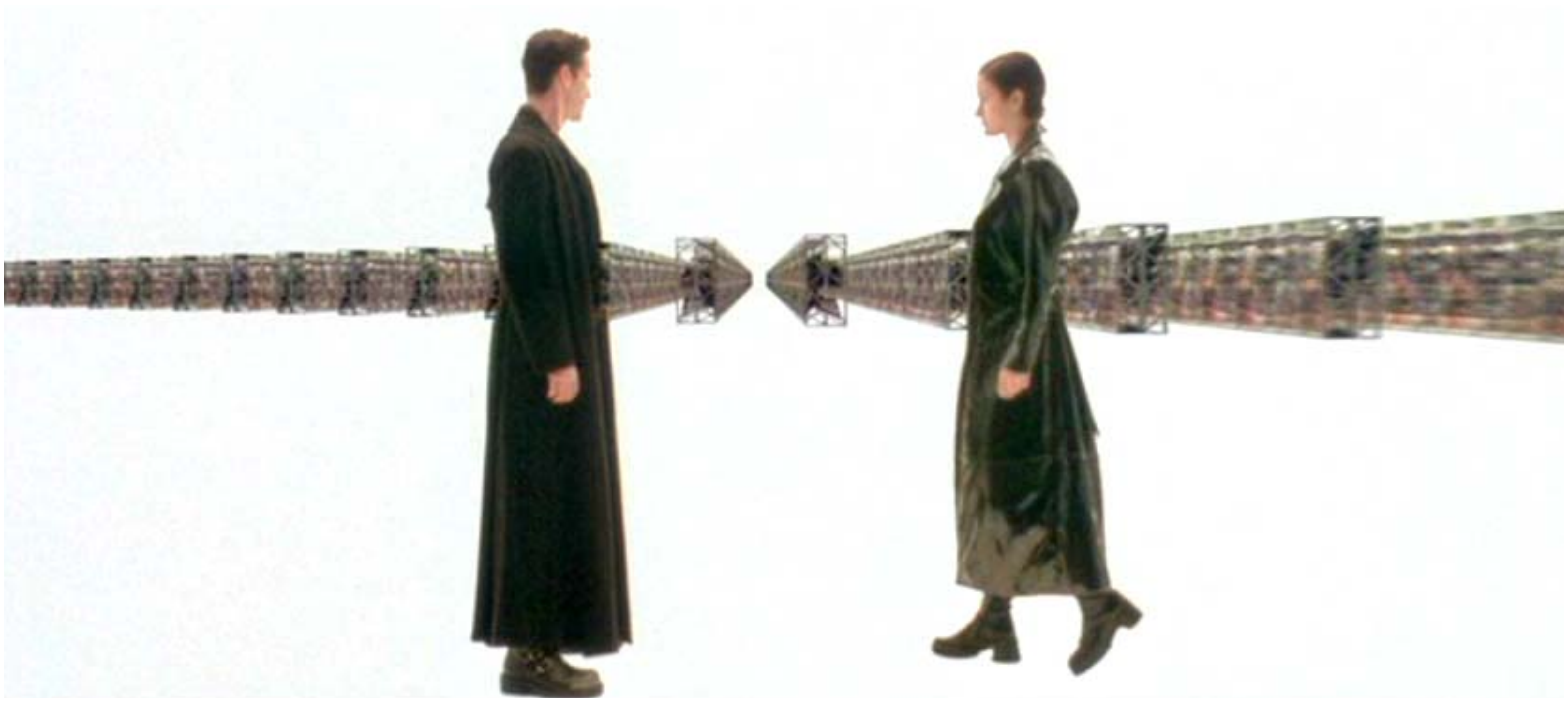
```
# ...sheets stuff up here...
from pony_blanket import csv_to_db
db, models = csv_to_db(
    {'hapmap.txt': HapMapAllele})

from pony.orm import db_session, select
with db_session:
  print len(select(x.rsid
                   for x in models[HapMapAllele]
                   if x.alt_freq > 0.01))
```

# pony-blanket example

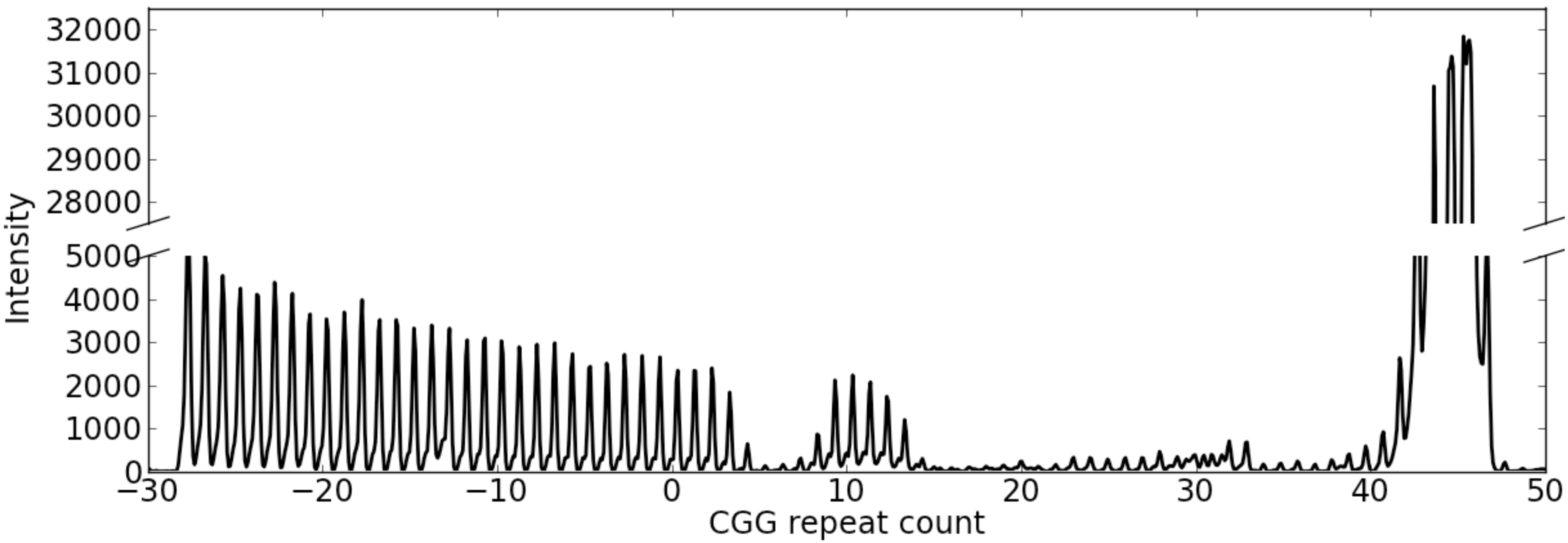
```
# ...sheets stuff up here...
from pony[redacted]
db, model[redacted]
    {'hap[redacted]

from pony[redacted]
with db_s[redacted]
  print l[redacted]
[redacted]apAllele]
                    if x.alt_freq > 0.01))
```

# Let’s talk about numbers.

# Lots of numbers.



# Like this.




$\times 10^5$

# Large-Scale Numeric Storage

- Text formats are slow to parse, bulky
- Relational DBs are not optimized for numeric storage/ops

Straw-man benchmark: read 10e6 random floats

| Format | Speed | Size | Space Overhead |
|---|---|---|---|
| CSV | 2918 ms | 144 MiB | 87.5% |
| JSON | 1410ms | 194 MiB | 153% |
| SQLite3 | 10996 ms | 166 MiB | 116% |
| HDF5 | 33 ms | 77 MiB | 0.0027% |

Counsyl

# tables: A Quick Intro

```python
# data is a numpy array

import tables
h5 = tables.openFile('mydata.h5', 'w')
h5.createArray(h5.root, 'some_data', data)
h5.close()
```

# tables: A Quick Intro

```python
with tables.openFile('mydata.h5', 'r') as h5:
    data = h5.root.some_data[:]

    sampled_data = h5.root.some_data[::10]
```

Counsyl

# tables: Compression

```
filters = tables.Filters(complevel=1,
                         complib='zlib',
                         fletcher32=True,
                         shuffle=True)

chunked = h5.createCArray(h5.root, 'chunked_data',
                          atom=tables.FloatAtom(),
                          shape=data.shape,
                          filters=filters)

chunked[:] = data
```

# tables: Metadata Storage

```
import numpy as np
row_type = np.dtype([('batch', 'S16'),
                     ('well', 'S3'),
                     ('idx', 'u8')])

metatable = h5.createTable(h5.root, 'metadata',
                           description=row_type)
```

Counsyl

# tables: Metadata Storage

```
new_row = metatable.row

for i in xrange(10):
    new_row['batch'] = 'Batch %d' % i
    new_row['well'] = 'A01'
    new_row['idx'] = i
    new_row.append()

metatable.cols.batch.createCSIndex()
metatable.cols.well.createCSIndex()

metatable.flush()
```

Counsyl

# tables: Metadata Queries

```python
def get_row_indices(batch, well):

    query = '(batch == "%s") & (well == "%s")'
    params = (batch, well)

    for row in metatable.where(query % params):
        yield row['idx']
```

Counsyl

# tables: Metadata Queries

```
def get_ro

    query                        == "%s")'
    params

    for ro                       % params):
        yie
```

# Summary

# Summary

- Declarative schemas make text parsing more readable and maintainable (`sheets`)

# Summary

- Declarative schemas make text parsing more readable and maintainable (`sheets`)
- In-memory databases make wrangling code more flexible and readable (`sqlite3`)

# Summary

- Declarative schemas make text parsing more readable and maintainable (`sheets`)
- In-memory databases make wrangling code more flexible and readable (`sqlite3`)
- Metaprogramming lets you use ORMs to do everything in Python, without DRY violations (`pony.orm`, `pony_blanket`)

# Summary

- Declarative schemas make text parsing more readable and maintainable (**`sheets`**)
- In-memory databases make wrangling code more flexible and readable (**`sqlite3`**)
- Metaprogramming lets you use ORMs to do everything in Python, without DRY violations (**`pony.orm`**, **`pony_blanket`**)
- HDF5 lets you store both numeric data and non-numeric metadata fast and small

Counsyl

</talk>

ihaque@counsyl.com

@imranshaque

https://github.com/ihaque/pydata_nyc_2013