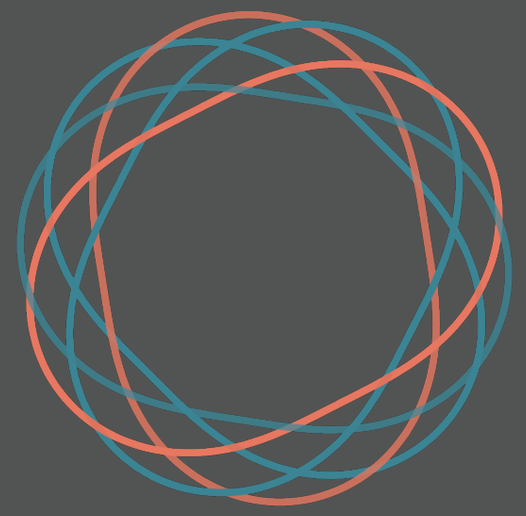
PYTHON DATA BIKESHED

Hi. I'm Rob Story.

github.com/wrobstory/pydataseattle2015

@oceankidbilly

I work @simple



Great company. Great team.
Interesting Data.

We’re hiring.

(A little Python. A lotta JVM)

Question:

I have data.
It's July 2015.
I want to group things.
or count things.
or average things.
or add things.

What library should I use?

It Depends.

(cop out)

Enter:
The Bikeshed

We are lucky to have a PyData ecosystem where there are domain-specific tools for different applications.

Analogy:

Python Data Libs :: Bikes

Think about tools in terms of
analysis velocity and ***data locality***

Wait: Why should I use Python for Data Analytics anyway?

While Python might not be the fastest language for things like web servers, it is ***very*** fast for HPC & numerics (because C*)

(and maybe Rust in the future?)

Back to choosing a lib:
First we need a dataset.

Diamonds Data:

http://vincentarelbundock.github.io/Rdatasets/datasets.html

(Yep, it's an R website.
Their community is really good at dataset aggregation)

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.2	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75

My needs are **simple.**
I don't like **dependencies.**
I'm on an old, old version of Python.

Seriously, **no dependencies.**

stdlib? **stdlib.**

```
import csv

conversion_map = {'carat': float, 'depth': float,
                  'price': int, 'table': float,
                  'x': float, 'y': float,
                  'z': float}

def converter(type_map, row):
    """Yep, we need to roll our own type conversions."""
    converted_row = {}
    for col, val in row.items():
        converter = type_map.get(col)
        if converter:
            converted_row[col] = converter(val)
        else:
            converted_row[col] = val
    return converted_row

with open('diamonds.csv', 'r') as f:
    reader = csv.DictReader(f)
    diamonds = [converter(conversion_map, r) for r in reader]
```

```python
from collections import defaultdict

def grouper(grouping_col, seq):
    """People have definitely written a faster version than what
       I'm about to write"""
    groups = {}
    for row in seq:
        group = groups.get(row[grouping_col])
        if group is not None:
            for k, v in row.items():
                if k != grouping_col:
                    group[k].append(v)
        else:
            groups[row[grouping_col]] = defaultdict(list)
    return groups
```

```sql
select max(price)
from diamonds
where carat > 1;
```

```
def get_max_price():
    max_price = 0
    for row in diamonds:
        if row['carat'] > 1 and row['price'] > max_price:
            max_price = row['price']
    return max_price
get_max_price()
```

City Bike

Reliable. Familiar. A bit slow.

Velocity: Slower

Locality: Local Memory

stdlib works!

But...what if you have 10M rows instead of 50k?

Do you really want to spend your time writing aggregation code?

Are my functions composable? Pure? Lazily evaluated? If I write Python should I care?

What happens when my analysis gets more complicated (or uses time/dates in any way...)?

Is a list of dictionaries the best abstraction for tabular data?

I like a **functional approach** to data analysis
(purity, composable, etc)

Map, reduce, filter are my friends.

I think **composing data pipelines** is
awesome, especially if it can handle
streaming data.

Toolz!

```
>>> list(tz.mapcat(lambda r: [x + "-foo" for x in r],
                   [["A", "B"], ("c", "d"), ("bar", "baz")]))

['A-foo', 'B-foo', 'c-foo', 'd-foo', 'bar-foo', 'baz-foo']
```

```
>>> tz.take(2, diamonds)

<itertools.islice at 0x10fa68260>
```

```
select count(1)
from diamonds
groupby color;
```

```
>>> tz.frequencies([r['color'] for r in diamonds])

{'D': 6775, 'E': 9797, 'F': 9542, 'G': 11292,
 'H': 8304, 'I': 5422, 'J': 2808}
```

```sql
select count(1)
from diamonds
where price > 1000
group by clarity;
```

```
>>> tzc.pipe(diamonds,
              tzc.filter(lambda r: r['price'] > 1000),
              tzc.map(lambda r: (r['clarity'],)),
              tzc.countby(lambda r: r[0]),
              dict)

{'I1': 675, 'IF': 1042, 'SI1': 9978, 'SI2': 8118,
 'VS1': 5702, 'VS2': 8647, 'VVS1': 2108, 'VVS2': 3146}
```

```sql
select count(1)
from diamonds
where price > 1000
group by clarity;
```

```
def filter_and_count(kv):
    f_and_c = tz.thread_last(kv[1],
                             (tz.filter, lambda r: r['price'] > 1000),
                             tz.count)

    return kv[0], f_and_c

tz.thread_last(diamonds,
               (tz.groupby, 'clarity'),
               (tz.itemmap, filter_and_count))
```

```sql
select count(1)
from diamonds
where price > 1000
group by clarity;
```

```
def increment(accum, row):
    if row['price'] > 1000:
        return accum + 1
    else:
        return accum

tz.reduceby('clarity',
            increment,
            diamonds, 0)
```

Fixie?

“Each toolz function consumes just iterables, dictionaries, and functions and each toolz function produces just iterables, dictionaries, and functions.“

This is great.

We’re building on data structures we already know

Bike Tools

Velocity: Slower (toolz), Faster (cytoolz)

Locality: Local Memory

Toolz is great!

But…I don't like functional programming.

I'm working with **tabular data** here- can't I have a tabular data interface?

I still have to work with time/dates.

I am sad.

A brief interlude...

Cython!

"...a superset of the Python language that additionally supports calling C functions and declaring C types"

Numexpr!

"... fast numerical expression evaluator for NumPy"

Numba!

JIT compilation to LLVM with only a few decorators needed


Made in USA 7075
130mmx53t

Numpy!

It would be a disservice to have a Python Data Toolbox presentation and not talk about Numpy.

It is the foundation on which almost *all* of the tools we have talked about today are built.

Pandas? Deeply tied to Numpy.
xray? Directly exposes Numpy.
bcolz? Has it's own array type, still uses Numpy tooling.
Blaze? Most of the internals are working with and leveraging numpy ndarray.

Numpy is still critical PyData infrastructure

/interlude

I have tabular data. Give me **DataFrames.**

I want fast/intuitive **exploratory analytics.**

I want a really, really fast CSV importer.

I want easy interfacing with SQL.

I have timeseries data. **Help. Please.**

Pandas!

```
import pandas as pd

df = pd.read_csv('diamonds.csv', index_col=0)

>>> df.describe()
```

```
              carat         depth         table         price             x  \
count  53940.000000  53940.000000  53940.000000  53940.000000  53940.000000
mean       0.797940     61.749405     57.457184   3932.799722      5.731157
std        0.474011      1.432621      2.234491   3989.439738      1.121761
min        0.200000     43.000000     43.000000    326.000000      0.000000
25%        0.400000     61.000000     56.000000    950.000000      4.710000
50%        0.700000     61.800000     57.000000   2401.000000      5.700000
75%        1.040000     62.500000     59.000000   5324.250000      6.540000
max        5.010000     79.000000     95.000000  18823.000000     10.740000

                y             z
count  53940.000000  53940.000000
mean       5.734526      3.538734
std        1.142135      0.705699
min        0.000000      0.000000
25%        4.720000      2.910000
50%        5.710000      3.530000
75%        6.540000      4.040000
max       58.900000     31.800000
```

```sql
select cut, mean(price)
from diamonds
group by cut;
```

```python
df.groupby('cut')['price'].mean()
```

```sql
select count(carat)
from diamonds
where price > 1000
group by clarity;
```

```pandas
df[df['price'] > 1000].groupby('clarity')['carat'].count()
```

```sql
select cut, price
from diamonds
where cut in ('Ideal', 'Premium')
order by price desc
limit 10;
```

```pandas
df[df['cut'].isin(['Ideal', 'Premium'])].sort('price', ascending=False)[:10]
```

```
>>> dates = pd.date_range('2015-03-25', periods=150, freq='H')
>>> dates
DatetimeIndex(['2015-03-25 00:00:00', '2015-03-25 01:00:00',
               '2015-03-25 02:00:00', '2015-03-25 03:00:00',
               '2015-03-25 04:00:00', '2015-03-25 05:00:00',
               '2015-03-25 06:00:00', '2015-03-25 07:00:00',
               '2015-03-25 08:00:00', '2015-03-25 09:00:00',
               ...
               '2015-03-30 20:00:00', '2015-03-30 21:00:00',
               '2015-03-30 22:00:00', '2015-03-30 23:00:00',
               '2015-03-31 00:00:00', '2015-03-31 01:00:00',
               '2015-03-31 02:00:00', '2015-03-31 03:00:00',
               '2015-03-31 04:00:00', '2015-03-31 05:00:00'],
              dtype='datetime64[ns]', length=150, freq='H', tz=None)
```

```
time_df = pd.DataFrame(np.random.randint(0, 500, 150),
                       index=dates, columns=["Numeric"])

>>> time_df.resample('D', how='mean')

               Numeric
2015-03-25  250.916667
2015-03-26  239.875000
2015-03-27  239.500000
2015-03-28  230.916667
2015-03-29  286.250000
2015-03-30  222.250000
2015-03-31  250.000000
```

```
missing = pd.to_datetime(['2015-03-25', '2015-03-30',
'2015-04-05'])
missing_df = pd.DataFrame(np.random.randint(0, 10, 3),
                          index=missing, columns=["Numeric"])
```

```
>>> missing_df
            Numeric
2015-03-25        3
2015-03-30        0
2015-04-05        3
```

```
>>> missing_df.asfreq('D', method='pad')
            Numeric
2015-03-25        3
2015-03-26        3
2015-03-27        3
2015-03-28        3
2015-03-29        3
2015-03-30        0
2015-03-31        0
2015-04-01        0
2015-04-02        0
2015-04-03        0
2015-04-04        0
2015-04-05        3
```

```
>>> missing_df.asfreq('D')
            Numeric
2015-03-25        3
2015-03-26      NaN
2015-03-27      NaN
2015-03-28      NaN
2015-03-29      NaN
2015-03-30        0
2015-03-31      NaN
2015-04-01      NaN
2015-04-02      NaN
2015-04-03      NaN
2015-04-04      NaN
2015-04-05        3
```

Geared Commuter

Daily use. Lots of features.

Velocity: Faster

Locality: Local Memory

Pandas rocks!

But…I'm working with a lot of **N-dimensional data**, where I need to do fast numerics on large homogeneous arrays.

That being said…I don't want to give up on all of Pandas nice **labeling and indexing** features.

In-memory tables are bumming me out.

I have **N-Dimensional homogeneous arrays**.

I want to be able to easily **aggregate data in multiple dimensions**.

I want to serialize to **NetCDF**.

I might have to deal with **OpenDAP**

XRAY!

```
arr = np.array([[1, 2, 3, 4],
                [10, 20, 30, 40],
                [100, 200, 300, 400]])
dim0_coords = ['a', 'b', 'c']
dim1_coords = ['foo', 'bar', 'baz', 'qux']

>>> xray.DataArray(arr, [('x', dim0_coords),
                         ('y', dim1_coords)])
<xray.DataArray (x: 3, y: 4)>
array([[  1,   2,   3,   4],
       [ 10,  20,  30,  40],
       [100, 200, 300, 400]])
Coordinates:
  * x        (x) |S1 'a' 'b' 'c'
  * y        (y) |S3 'foo' 'bar' 'baz' 'qux'
```

```
>>> da.sel(x=['a', 'c'])

<xray.DataArray (x: 2, y: 4)>
array([[  1,   2,   3,   4],
       [100, 200, 300, 400]])
Coordinates:
  * x        (x) |S1 'a' 'c'
  * y        (y) |S3 'foo' 'bar' 'baz' 'qux'
```

```
>>> da + 100

<xray.DataArray (x: 3, y: 4)>
array([[101, 102, 103, 104],
       [110, 120, 130, 140],
       [200, 300, 400, 500]])
Coordinates:
  * y        (y) |S3 'foo' 'bar' 'baz' 'qux'
  * x        (x) |S1 'a' 'b' 'c'
```

```
x_y = np.array([[1, 1.1, 1.2], [2, 2.1, 2.2],
                [3, 3.1, 3.2]])
z_coords = np.array(["10s", "20s"])
data_cube = np.array([[[10, 10, 10],
                       [10, 10, 10],
                       [10, 10, 10]],
                      [[20, 20, 20],
                       [20, 20, 20],
                       [20, 20, 20]]])

>>> xray.Dataset({"cube": (["z", "x", "y"], data_cube)},
                 coords={"z": z_coords,
                         "x": ["col_1", "col_2", "col_3"],
                         "y": ["row_1", "row_2", "row_3"]})

<xray.Dataset>
Dimensions:  (x: 3, y: 3, z: 2)
Coordinates:
  * y        (y) |S5 'row_1' 'row_2' 'row_3'
  * x        (x) |S5 'col_1' 'col_2' 'col_3'
  * z        (z) |S3 '10s' '20s'
Data variables:
    cube     (z, x, y) int64 10 10 10 10 10 10 20 20 20 20 ...
```

```
>>> ds.groupby("x").sum()

<xray.Dataset>
Dimensions:  (x: 3)
Coordinates:
  * x        (x) |S5 'col_1' 'col_2' 'col_3'
Data variables:
    cube     (x) int64 90 90 90
```

BMX Bike

Specialized. Multi-dimensional.

Velocity: Faster

Locality: Local Memory

xray is awesome!

But…I want to work with data with Pandas-like expressions across **many data sources**.

What if I have some data in **SQL**, some data in **CSVs**, some data in **HDF5**, some data in…

I'm still running into in-memory problems

I have **BIG DATA.** or maybe just **Medium Data**.
How about **Bigger-than-I-can-RAM-data**.

Why are my **analytical expressions** tied to my data structure?

Can I have **expressions** that map across **data structures *and* storage?**

Blaze!

```
import blaze as bz

bz_diamonds = bz.symbol('diamonds', bz.discover(df_diamonds))
```

```
>>> type(bz_diamonds)

blaze.expr.expressions.Symbol
```

```
mean_price = bz.by(bz_diamonds.cut, price=bz_diamonds.price.mean())
```

```
carat_count = bz.by(bz_diamonds[bz_diamonds.price > 1000].clarity,
                    carat=bz_diamonds.carat.count())
```

```
>>> bz.compute(mean_price, df_diamonds)

         cut         price
0       Fair   4358.757764
1       Good   3928.864452
2      Ideal   3457.541970
3    Premium   4584.257704
4  Very Good   3981.759891

>>> bz.compute(carat_count, df_diamonds)

   clarity  carat
0       I1    741
1       IF   1790
2      SI1  13065
3      SI2   9194
4      VS1   8171
5      VS2  12258
6     VVS1   3655
7     VVS2   5066
```

```
conn_str = "postgresql://postgres:postgres@localhost/pydata::diamonds"
pg_datasource = bz.odo(df_diamonds, conn_str)
```

```
>>> bz.compute(carat_count, pg_datasource)

<sqlalchemy.sql.selectable.Select at 0x1164f7e50; Select object>
```

```
>>> bz.odo(bz.compute(carat_count, pg_datasource), pd.DataFrame)

  clarity  carat
0      IF   1790
1      I1    741
2    VVS1   3655
3     VS2  12258
4     VS1   8171
5    VVS2   5066
6     SI2   9194
7     SI1  13065
```

Mountain Bike

Multi-speed, Multi-terrain

Velocity: Varied (computation engine)
Locality: Varied (data source)

YAY BLAZE!

But…I have Big/Medium/Lots of Data, and Databases aren't fast enough, and both **memory and disk space are at a premium**.

Isn't there some way to **compress my data** somehow for these in-memory computations?

Is there a way to get more out of my local machine?

I have **homogeneous array data**

I want to **compress it both in-memory/on-disk**, but have that decompression be fast enough to perform useful analytics.

bcolz!

```
import bcolz

>>> dc = bcolz.ctable.fromdataframe(df_diamonds)

carat : carray((53940,), float64)
  nbytes: 421.41 KB; cbytes: 468.10 KB; ratio: 0.90
  cparams := cparams(clevel=5, shuffle=True,
cname='blosclz')
[ 0.23  0.21  0.23 ...,  0.7   0.86  0.75]
cut : carray((53940,), |S9)
  nbytes: 474.08 KB; cbytes: 315.96 KB; ratio: 1.50
  cparams := cparams(clevel=5, shuffle=True,
cname='blosclz')
['Ideal' 'Premium' 'Good' ..., 'Very Good' 'Premium'
'Ideal']
...
```

```
dsize = dc.cbytes / 2**20.
```

```
>>> "Total size for the ctable: {} MB".format(dsize))
```

```
Total size for the ctable: 3.49322795868 MB
```

```
>>> "Compression ratio the ctable: {}".format((dc.nbytes /
float(dc.cbytes))))
```

```
Compression ratio the ctable: 1.03081835096
```

```
>>> dc["cut == 'Premium'"]

array([(0.21, 'Premium', 'E', 'SI1', 59.8, 61.0, 326, 3.89, 3.84, 2.31),
       (0.29, 'Premium', 'I', 'VS2', 62.4, 58.0, 334, 4.2, 4.23, 2.63),
       (0.22, 'Premium', 'F', 'SI1', 60.4, 61.0, 342, 3.88, 3.84, 2.33),
       ...,
       (0.71, 'Premium', 'F', 'SI1', 59.8, 62.0, 2756, 5.74, 5.73, 3.43),
       (0.72, 'Premium', 'D', 'SI1', 62.7, 59.0, 2757, 5.69, 5.73, 3.58),
       (0.86, 'Premium', 'H', 'SI2', 61.0, 58.0, 2757, 6.15, 6.12, 3.74)],
      dtype=[('carat', '<f8'), ('cut', 'S9'), ('color', 'S1'), ...)
```

```
diskdc = dc.copy(rootdir='diamonds')

>>> diskdc["(cut == 'Ideal') & (price > 1000)"]

array([(0.7, 'Ideal', 'E', 'SI1', 62.5, 57.0, 2757, 5.7, 5.72, 3.57),
       (0.7, 'Ideal', 'G', 'VS2', 61.6, 56.0, 2757, 5.7, 5.67, 3.5),
       (0.74, 'Ideal', 'G', 'SI1', 61.6, 55.0, 2760, 5.8, 5.85, 3.59),
       (0.71, 'Ideal', 'G', 'VS1', 61.4, 56.0, 2756, 5.76, 5.73, 3.53),
       (0.72, 'Ideal', 'D', 'SI1', 60.8, 57.0, 2757, 5.75, 5.76, 3.5),
       (0.75, 'Ideal', 'D', 'SI2', 62.2, 55.0, 2757, 5.83, 5.87, 3.64)],
      dtype=[('carat', '<f8'), ('cut', 'S9'), ('color', 'S1'),
             ('clarity', 'S4'), ('depth', '<f8'),
             ('table', '<f8'), ('price', '<i8'), ('x', '<f8'),
             ('y', '<f8'), ('z', '<f8')])
```

```
import os

for root, dirs, files in os.walk('diamonds'):
    level = root.replace('diamonds', '').count(os.sep)
    indent = ' ' * 4 * (level)
    print('{}{}/'.format(indent, os.path.basename(root)))
    subindent = ' ' * 4 * (level + 1)
    for f in files:
        print('{}{}'.format(subindent, f))
```

```
diamonds/
    __attrs__
    __rootdirs__
    carat/
        __attrs__
        data/
            __0.blp
            __1.blp
        meta/
            sizes
            storage
```

Recumbent

Fast, some real advantages, specific audience

Velocity: Fast

Locality: In-memory/On-disk

bcolz is a big deal!

Being able to perform **aggregation on compressed data on-disk or in-memory** is huge for medium data analytics.

But…what if I *really* want **parallel, out-of-core computing?**

Dask!

"enables **parallel computing** through **task scheduling** and **blocked algorithms**."

Recipe for all-your-cores and out-of-core computation:

1. Partition arrays/iterables/dataframes (blocked algorithms)
2. Perform parallel/scheduled computations on those partitions.
3. Put the pieces back together

Arrays: Numpy-Like

```
import dask.array as da

d_arr = da.from_array(np.random.randn(100000), chunks=100)
```

```
>>> d_arr

dask.array<x_1, shape=(100000,), chunks=((100, ...)), dtype=float64>

>>> d_arr.sum().compute()

-546.02061727798946
```

DataFrames: DataFrame-like

```
import dask.dataframe as dd

ddf = dd.read_csv('diamonds.csv')

>>> ddf[ddf['price'] > 1000].groupby('clarity')['carat'].count().compute()

clarity
I1       675
IF      1042
SI1     9978
SI2     8118
VS1     5702
VS2     8647
VVS1    2108
VVS2    3146
Name: carat, dtype: int64

>>> ddf[ddf['cut'].isin(['Ideal', 'Premium'])].sort('price', ascending=False)[:10]

AttributeError: 'DataFrame' object has no attribute 'sort'
```

Sequences: Seq-like

```
import dask.bag as db

>>> diamonds_b = db.from_sequence(diamonds)

18823

>>> tzc.pipe(diamonds_b,
             tzc.filter(lambda r: r['price'] > 1000),
             tzc.map(lambda r: (r['clarity'],)),
             tzc.countby(lambda r: r[0]),
             dict)

{'I1': 675,
 'IF': 1042,
 'SI1': 9978,
 'SI2': 8118,
 'VS1': 5702,
 'VS2': 8647,
 'VVS1': 2108,
 'VVS2': 3146}
```

Tandem

Fast, Parallel

Velocity: Fast

Locality: In-memory/On-disk/Distributed

What's Left?

(there's more?!)

Spark!

"…fast and general-purpose cluster computing system. "

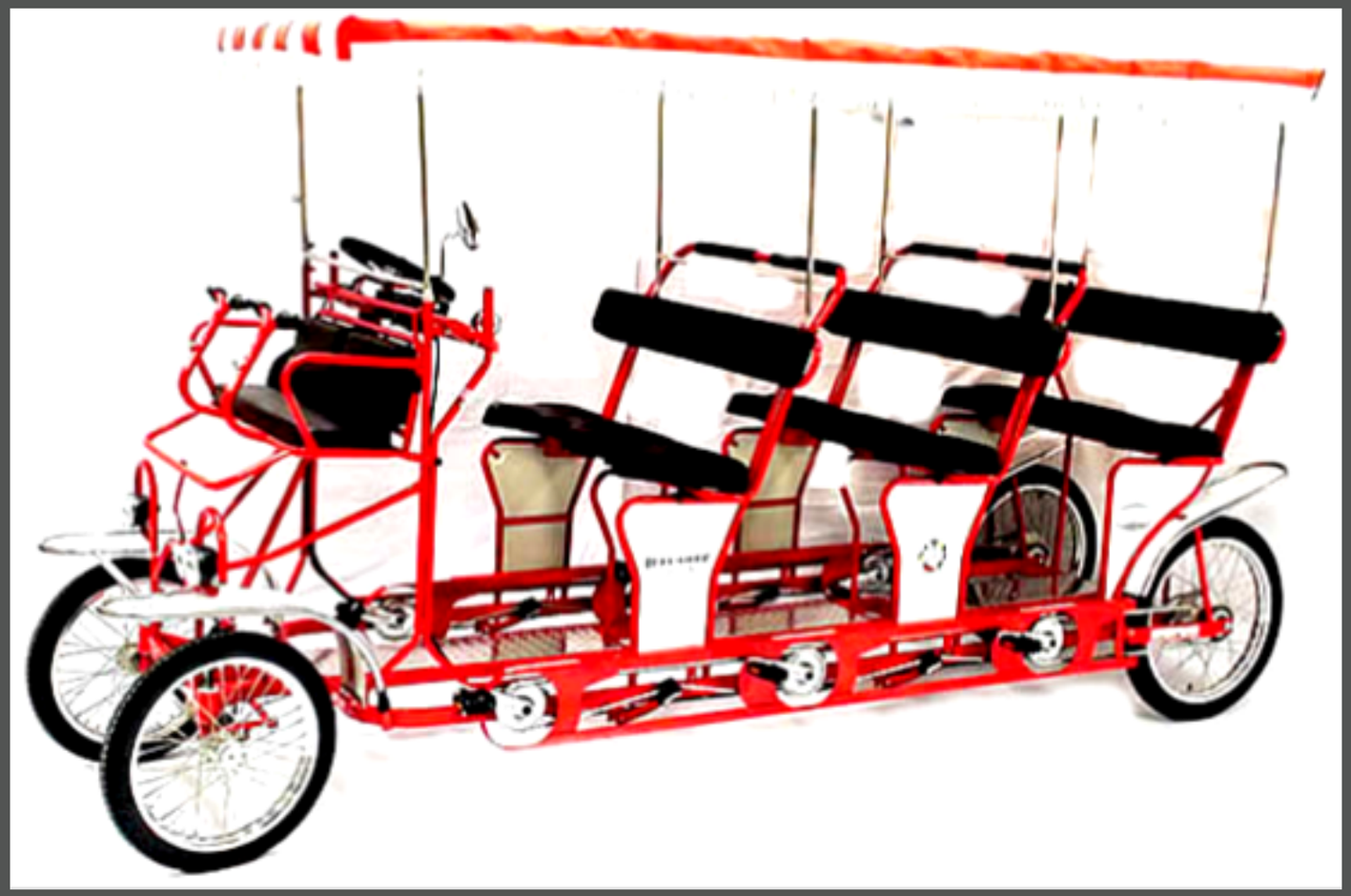
Could do an entire presentation on Spark/PySpark.

Remember toolz? Think the same **type of map/reduce/ flatMap/filter dataflow**, except **distributed.**

Also: **DataFrames! Streaming data! Machine Learning!**

Multi-person Surrey

Fast*, Parallel, Distributed



Velocity: Fast

Locality: In-memory/On-disk/Distributed

*This analogy has now completely come off the rails

JVM

Distributed/out-of-core ndarray

- **Bolt:** ndarray backed by Spark (and more?)
- **DistArray:** Distributed-Array-Protocol
- **Biggus:** Virtual large arrays + lazy eval
- **Spartan:** “Distributed Numpy”


momondo
FIND FLY OG HOTEL

SArray

on-disk/in-memory ndarray

SFrame

on-disk/in-memory DataFrame

Ibis!

"…pandas-like data expression system"

Built on **Impala** + **Hadoop** (for now)

Some things are **Tedious/Difficult** to do in SQL, much less Map/Reduce.

- **Timeseries queries (window functions)**
- **Correlated subqueries**
- **Self-joins**

Stats + ML

Scikit-learn

IMO, very clearly the benchmark in ML libraries, both in API and documentation

statsmodels

Statistical models in Python built on our PyData toolbox

What should I paint the bikeshed with?

Seaborn + Bokeh

THE END!

THANK YOU!