Optimizing Pandas code for performance

Sofia Heisler

mapbox

Download these slides at:

bit.ly/2Jci3BS

What's Pandas?

- Open-source library that offers data structure support and a great set of tools for data analysis
- Makes Python a formidable competitor to R and other data science tools
- Widely used in everything from simple data manipulation to complex machine learning

bit.ly/2Jci3BS

Why optimize Pandas?

- Pandas is built on top of NumPy and Cython, making it very fast when used correctly
- Correct optimizations can make the difference between minutes and milliseconds

Our working dataset

All hotels in New York state sold by Expedia

```
import pandas as pd
import numpy as np
from math import *
```

```
df = pd.read_csv('new_york_hotels.csv', encoding='cp1252')
```

```
df.head()
```

	ean_hotel_id	name	address1	city	state_province	postal_code	latitude	longitude	star_rating	high_rate	low_rate
0	269955	Hilton Garden Inn Albany/SUNY Area	1389 Washington Ave	Albany	NY	12206	42.68751	-73.81643	3.0	154.0272	124.0216
1	113431	Courtyard by Marriott Albany Thruway	1455 Washington Avenue	Albany	NY	12206	42.68971	-73.82021	3.0	179.0100	134.0000
2	108151	Radisson Hotel Albany	205 Wolf Rd	Albany	NY	12205	42.72410	-73.79822	3.0	134.1700	84.1600
3	254756	Hilton Garden Inn Albany Medical Center	62 New Scotland Ave	Albany	NY	12208	42.65157	-73.77638	3.0	308.2807	228.4597
4	198232	CrestHill Suites SUNY University Albany	1415 Washington Avenue	Albany	NY	12206	42.68873	-73.81854	3.0	169.3900	89.3900

Source: http://developer.ean.com/database/property-data

How do we know how fast a function is? Magic

- “Magic” commands available through Jupyter/IPython notebooks provide additional functionality on top of Python code to make it that much more awesome

- Magic commands start with **%** (executed on just the line) or **%%** (executed on the entire cell)

Timing functions with **%timeit**

- Use IPython's magic **%timeit** command
- Re-runs a function repeatedly and shows the average and standard deviation of runtime obtained
- Can serve as a benchmark for further optimization

bit.ly/2Jci3BS

Slow Pandas: Looping

bit.ly/2Jci3BS

Our practice function: Haversine distance

```
def haversine(lat1, lon1, lat2, lon2):
    miles_constant = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad,\
                                 [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) *\
        np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    mi = miles_constant * c
    return mi
```

bit.ly/2Jci3BS

Crude iteration, or what not to do

- Rookie mistake: “I just wanna loop over all the rows!”
- Pandas is built on NumPy, designed for vector manipulation - loops are inefficient
- Looping through individual cells using indexes is tempting but wrong

bit.ly/2Jci3BS

Looping over a function with indexes

```
def haversine_looping(df):
    distance_list = []
    for i in range(0, len(df)):
        d = haversine(40.671, -73.985,\
            df.iloc[i]['latitude'], df.iloc[i]['longitude'])
        distance_list.append(d)
    return distance_list

%%timeit
df['distance'] = haversine_looping(df)
```

```
682 ms ± 6.65 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

bit.ly/2Jci3BS

Crude iteration with iterrows

- The Pandas **iterrows** method will provide a tuple of (Index, Series) that you can loop through more efficiently

- Still slow :-(

bit.ly/2Jci3BS

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x

bit.ly/2Jci3BS

Running function with iterrows

```
%%timeit
haversine_series = []
for index, row in df.iterrows():
    haversine_series.append(haversine(40.671, -73.985,\
                    row['latitude'], row['longitude']))
df['distance'] = haversine_series
```

```
184 ms ± 6.65 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

bit.ly/2Jci3BS

Nicer looping: using apply

- **apply** applies a function along a specified axis (rows or columns)
- More efficient than **iterrows**, but still requires looping through rows
- Best used only when there is no way to vectorize a function

bit.ly/2Jci3BS

Timing looping with apply

```
%%timeit
df['distance'] =\
df.apply(lambda row: haversine(40.671, -73.985,\
         row['latitude'], row['longitude']), axis=1)
```

```
78.1 ms ± 7.55 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

bit.ly/2Jci3BS

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x
Looping with apply	**78.10**	**2.4x**

bit.ly/2Jci3BS

Peeking under the hood with %lprun

- Use the line_profiler tool to determine how much each line in the function contributes to the runtime

- To run line_profiler inside your notebook:

```
%lprun -f your_function_here
```

github.com/rkern/line_profiler

bit.ly/2Jci3BS

Apply is doing a lot of repetitive steps

```
%lprun -f haversine \
df.apply(lambda row: haversine(40.671, -73.985,\
            row['latitude'], row['longitude']), axis=1)
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def haversine(lat1, l
     2      1631         1429      0.9      3.1      miles_constant = 
     3      1631        17035     10.4     36.7      lat1, lon1, lat2,
     4      1631         1669      1.0      3.6      dlat = lat2 - lat
     5      1631         1143      0.7      2.5      dlon = lon2 - lon
     6      1631        16049      9.8     34.6      a = np.sin(dlat/2
     7      1631         6474      4.0     13.9      c = 2 * np.arcsin
     8      1631         1586      1.0      3.4      mi = miles_consta
     9      1631         1050      0.6      2.3      return mi
```

bit.ly/2Jci3BS

Vectorization

bit.ly/2Jci3BS

Doing it the pandorable way: vectorize

- The basic units of Pandas are arrays:
  - **Series** is a one-dimensional array with axis labels
  - **DataFrame** is a 2-dimensional array with labeled axes (rows and columns)
- **Vectorization** is the process of performing the operations on arrays rather than scalars

bit.ly/2Jci3BS

Why vectorize?

- Many built-in Pandas functions are built to operate directly on arrays (e.g. aggregations, string functions, etc.)

- Vectorized functions in Pandas are inherently much faster than looping functions

All calculations in our Haversine function can operate on vectors

```
def haversine(lat1, lon1, lat2, lon2):
    miles_constant = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad,\
                                 [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) *\
        np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    mi = miles_constant * c
    return mi
```

Color key:
NumPy functions
Vector-friendly functions

bit.ly/2Jci3BS

Vectorizing significantly improves performance

```
%%timeit
df['distance'] = haversine(40.671, -73.985,\
                           df['latitude'], df['longitude'])
```

```
1.79 ms ± 230 µs per loop (mean ± std. dev. of 7 runs, 100 loops
each)
```

bit.ly/2Jci3BS

The function is no longer looping

```
%lprun -f haversine haversine(40.671, -73.985,\
                              df['latitude'], df['longitude'])
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def haversine(lat1, lon
     2         1            2      2.0      0.0      miles_constant = 39
     3         1          529    529.0      8.8      lat1, lon1, lat2, l
     4         1          362    362.0      6.0      dlat = lat2 - lat1
     5         1          232    232.0      3.9      dlon = lon2 - lon1
     6         1         3511   3511.0     58.5      a = np.sin(dlat/2)*
     7         1          869    869.0     14.5      c = 2 * np.arcsin(n
     8         1          494    494.0      8.2      mi = miles_constant
     9         1            2      2.0      0.0      return mi
```

bit.ly/2Jci3BS

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x
Looping with apply	78.10	2.4x
Vectorization with Pandas	**1.79**	**43.6x**

bit.ly/2Jci3BS

Vectorization with NumPy arrays

bit.ly/2Jci3BS

Why NumPy?

- NumPy is a “fundamental package for scientific computing in Python”
- NumPy operations are executed “under the hood” in optimized, pre-compiled C code on **ndarray**s
- Cuts out a lot of the overhead incurred by operations on Pandas series in Python (indexing, data type checking, etc.)

bit.ly/2Jci3BS

Converting code to operate on NumPy arrays instead of Pandas series

```
%%timeit
df['distance'] = haversine(40.671, -73.985,\
         df['latitude'].values, df['longitude'].values)
```

```
370 µs ± 18 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

bit.ly/2Jci3BS

Optimizing with NumPy arrays

Runtime is down from 682 ms to 370 μs.

That's more than 1800-fold improvement!

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x
Looping with apply	78.10	2.4x
Vectorization with Pandas series	1.79	43.6x
Vectorization with NumPy arrays	**0.37**	**4.8x**

bit.ly/2Jci3BS

Okay, but I really wanted to use a loop…

bit.ly/2Jci3BS

Okay, but I really want to use a loop…

- There are a few reasons why you might actually want to use a loop:
    - Your function is complex and cannot operate on vectors
    - Trying to vectorize your function would result in significant memory overhead
    - You're just plain stubborn

Using Cython to speed up loops

bit.ly/2Jci3BS

Speeding up code with Cython

- Cython language is a superset of Python that supports calling C functions and declaring C types

- Almost any piece of Python code is also valid Cython code

- Cython compiler will convert Python code into C code which makes equivalent calls to the Python/C API.

Re-defining the function in the Cython compiler

```
%load_ext cython

%%cython
cpdef haversine_cy(lat1, lon1, lat2, lon2):
    miles_constant = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad,\
                              [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) *\
        np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    mi = miles_constant * c
    return mi
```

bit.ly/2Jci3BS

Re-defining the function in the Cython compiler

```
%%timeit
df['distance'] =\
    df.apply(lambda row: haversine_cy(40.671, -73.985,\
            row['latitude'], row['longitude']), axis=1)
```

```
76.5 ms ± 6.42 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

Scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x
Looping with apply	78.10	2.4x
Running row-wise function through Cython	**76.50**	**1.0x**
Vectorization with Pandas series	1.79	43.6x
Vectorization with NumPy arrays	0.37	4.8x

bit.ly/2Jci3BS

Evaluating results of conversion to Cython

Adding the **-a** option to **%%cython** magic command shows how much of the code has *not* actually been converted to C by default… and it's a lot!

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
 01:
 02: # Haversine cythonized (no other edits)
+03: import numpy as np
+04: cpdef haversine_cy(lat1, lon1, lat2, lon2):
+05:     miles_constant = 3959
+06:     lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
+07:     dlat = lat2 - lat1
+08:     dlon = lon2 - lon1
+09:     a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
+10:     c = 2 * np.arcsin(np.sqrt(a))
+11:     mi = miles_constant * c
+12:     return mi
```

bit.ly/2Jci3BS

Speeding up code with Cython

- As long as Cython is still using Python methods, we won't see a significant improvement
- Make the function more Cython-friendly:
  - Add explicit typing to the function
  - Replace Python/NumPy libraries with C-specific math libraries

Better cythonizing through static typing and C libraries

```
%%cython -a
from libc.math cimport sin, cos, acos, asin, sqrt

cdef deg2rad_cy(float deg):
    cdef float rad
    rad = 0.01745329252*deg
    return rad

cpdef haversine_cy_dtyped(float lat1, float lon1, float lat2, float lon2):
    cdef:
        float dlon
        float dlat
        float a
        float c
        float mi

    lat1, lon1, lat2, lon2 = map(deg2rad_cy, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    mi = 3959 * c
    return mi
```

bit.ly/2Jci3BS

Timing the cythonized function

```
%%timeit
df['distance'] =\
df.apply(lambda row: haversine_cy_dtyped(40.671, -73.985,\
           row['latitude'], row['longitude']), axis=1)
```

```
50.1 ms ± 2.74 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

bit.ly/2Jci3BS

Scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x
Looping with apply	78.10	2.4x
Running row-wise function through Cython compiler	76.50	1.0x
Looping with Cythoninzed function	**50.10**	**1.6x**
Vectorization with Pandas series	1.79	28x
Vectorization with NumPy arrays	0.37	4.8x

bit.ly/2Jci3BS

Our code is looking a lot more Cythonized, too

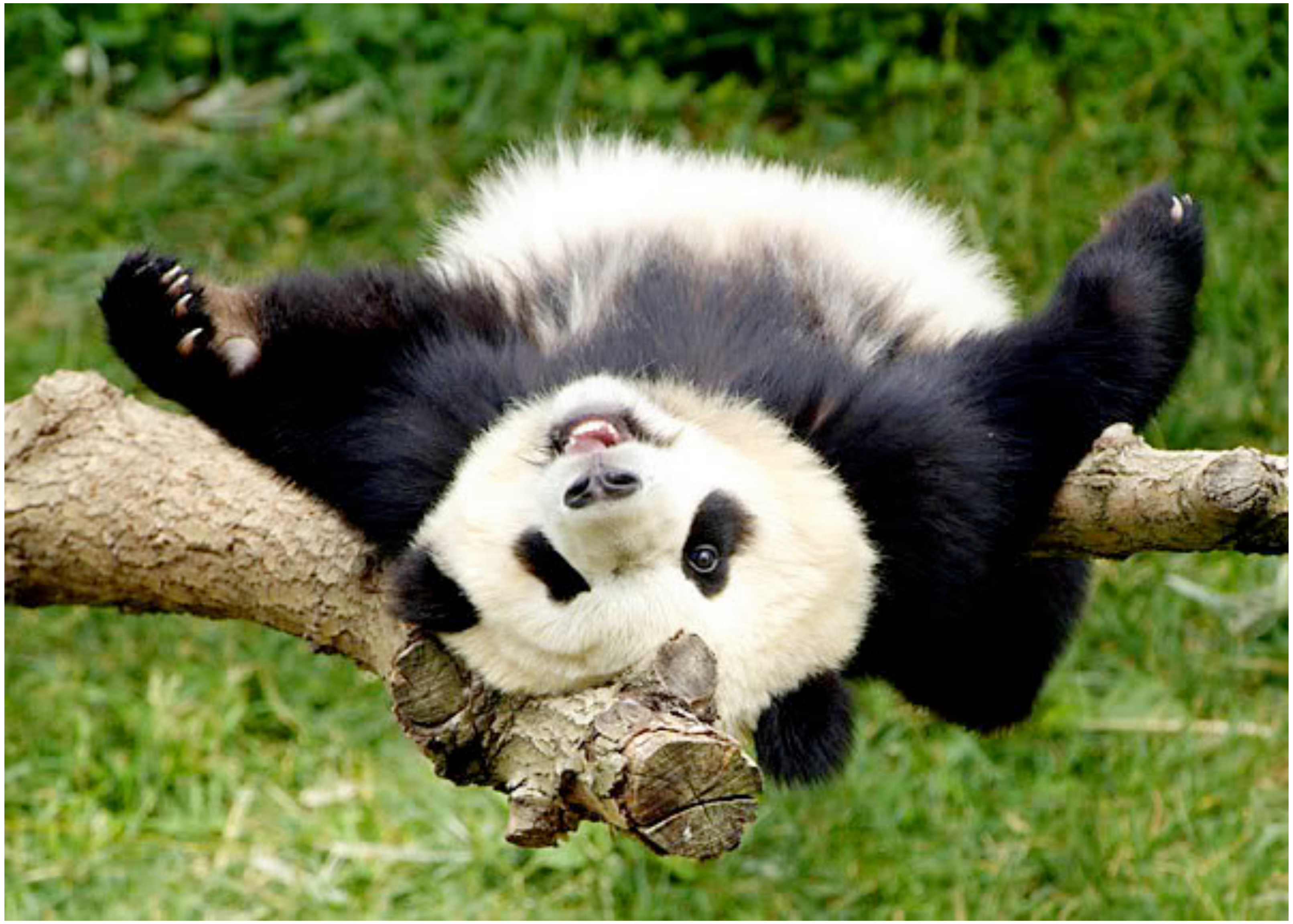
Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a '+' to see the C code that Cython generated for it.

```
# Haversine cythonized
from libc.math cimport sin, cos, acos, asin, sqrt

cpdef deg2rad_cy(float deg):
    cdef float rad
    rad = 0.01745329252*deg
    return rad

cpdef haversine_cy_dtyped(float lat1, float lon1, float lat2, float lon2):
    cdef:
        float dlon
        float dlat
        float a
        float c
        float mi

    lat1, lon1, lat2, lon2 = map(deg2rad_cy, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    mi = 3959 * c
    return mi
```

bit.ly/2Jci3BS

Summing it up



Source: http://i.imgur.com/z40kUTW.jpg

bit.ly/2Jci3BS

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping through indexes	682.00	
Looping with iterrows	184.00	3.7x
Looping with apply	78.10	2.4x
Looping with Cython	50.10	1.6x
Vectorization with Pandas series	1.79	28x
Vectorization with NumPy arrays	0.37	4.8x

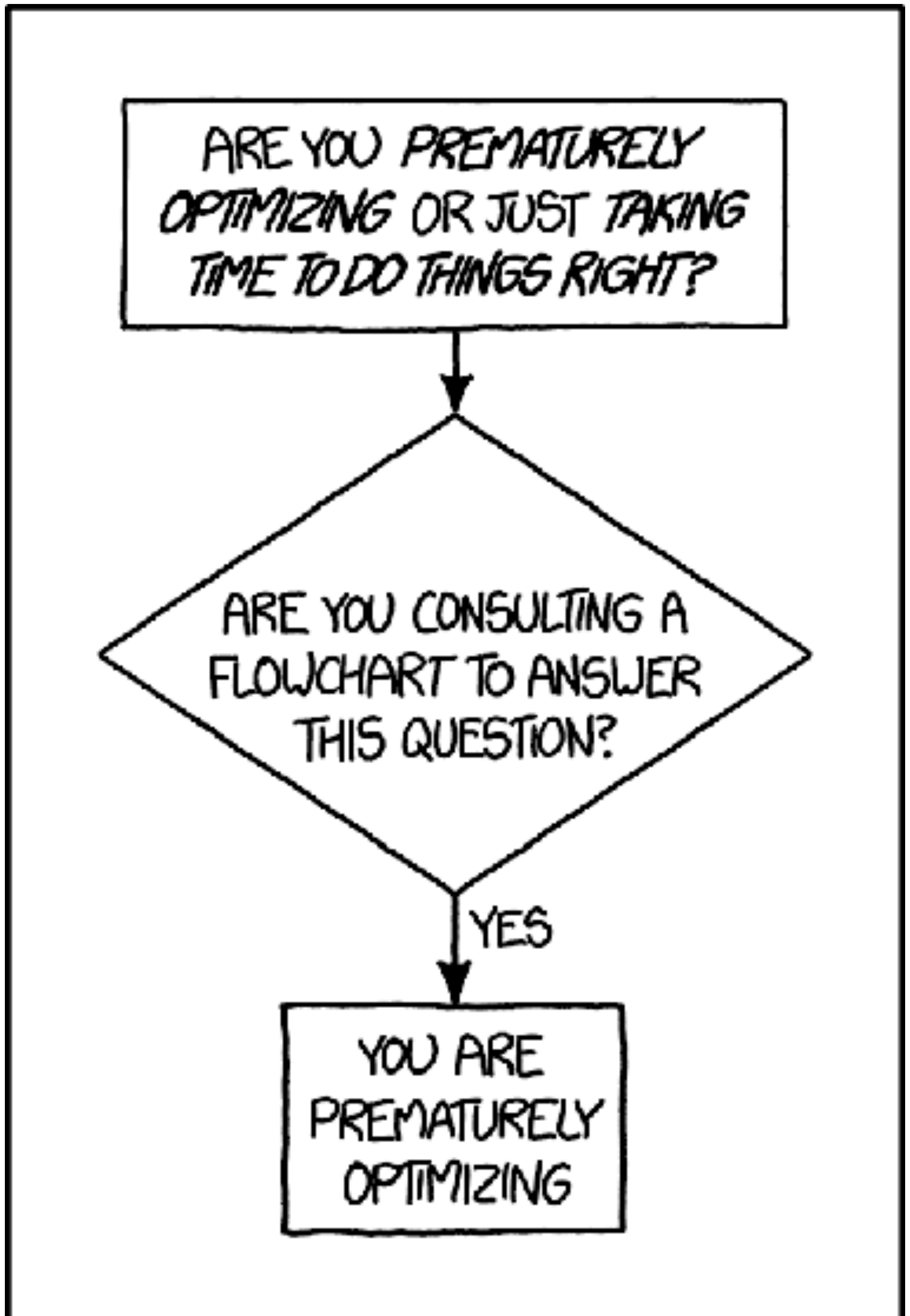
bit.ly/2Jci3BS

The zen of Pandas optimization

- Avoid loops
- If you must loop, use apply, not iteration functions
- If you must apply, use Cython to make it faster
- Vectorization is usually better than scalar operations
- Vector operations on NumPy arrays are more efficient than on native Pandas series

bit.ly/2Jci3BS

A word of warning…



Source: https://xkcd.com/1691/

"Premature optimization is the root of all evil"

--Donald Knuth

References

- http://cython.readthedocs.io/en/latest/
- http://cython.org/
- http://pandas.pydata.org/pandas-docs/stable/
- http://www.nongnu.org/avr-libc/user-manual/group__avr__math.html
- https://docs.python.org/2/library/profile.html
- https://docs.scipy.org/doc/numpy/user/whatisnumpy.html
- https://ipython.org/notebook.html
- https://penandpants.com/2014/09/05/performance-of-pandas-series-vs-numpy-arrays/
- https://www.datascience.com/blog/straightening-loops-how-to-vectorize-data-aggregation-with-pandas-and-numpy/