Making Pandas Fly (live from London)
EuroPython 2020

Ian Ozsvald
@IanOzsvald – ianozsvald.com
Introductions

• Interim Chief Data Scientist

• 19+ years experience

• Team coaching & public courses
  – I’m sharing from my Higher Performance Python course
Thank the organisers!

• All volunteers – go say thank you in #lobby
• They’ve put in a huge amount of volunteered work for us!
Today’s goal

• Pandas
  – Saving RAM to fit in more data
  – Calculating faster by dropping to Numpy

• Advice for “being highly performant”

• Has Covid 19 affected UK Company Registrations?
Strings are expensive and slow

display(df.CompanyCategory.value_counts()[:10])

Private Limited Company  4294231
PRI/LTD BY GUAR/NSC (Private, limited by guarantee, no share capital)  100365
Limited Liability Partnership  51750
PRI/LBG/NSC (Private, Limited by guarantee, no share capital, use of 'Limited' exemption)  40534
Community Interest Company  20189
Other company type  8492
Public Limited Company  5935
Private Unlimited Company  4106
Registered Society  151
Industrial and Provident Society  130
Name: CompanyCategory, dtype: int64

f"{df['CompanyCategory'].memory_usage(deep=True, index=False):,} bytes"

'369,713,766 bytes'
Categoricals are cheap and fast!

```python
df['CompanyCategory_cat'] = df.CompanyCategory.astype('category')
df['CompanyCategory_cat'].value_counts()[:10]
```

<table>
<thead>
<tr>
<th>Company Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Limited Company</td>
<td>4294231</td>
</tr>
<tr>
<td>PRI/LTD BY GUAR/NSC (Private, limited by guarantee, no share capital)</td>
<td>100365</td>
</tr>
<tr>
<td>Limited Liability Partnership</td>
<td>51750</td>
</tr>
<tr>
<td>PRI/LBG/NSC (Private, Limited by guarantee, no share capital, use of 'Limited' exemption)</td>
<td>40534</td>
</tr>
<tr>
<td>Community Interest Company</td>
<td>20189</td>
</tr>
<tr>
<td>Other company type</td>
<td>8492</td>
</tr>
<tr>
<td>Public Limited Company</td>
<td>5935</td>
</tr>
<tr>
<td>Private Unlimited Company</td>
<td>4106</td>
</tr>
<tr>
<td>Registered Society</td>
<td>151</td>
</tr>
<tr>
<td>Industrial and Provident Society</td>
<td>130</td>
</tr>
</tbody>
</table>

Name: CompanyCategory_cat, dtype: int64

```python
f"{df['CompanyCategory_cat'].memory_usage(deep=True, index=False):,} bytes"
```

4,528,328 bytes

Circa 1% of previous memory cost

By [ian]@ianozsvald[.com]
Categoricals
".cat" accessor

```
df['CompanyCategory_cat'].cat.categories

Index(['Community Interest Company',
      'European Public Limited-Liability Company (SE)',
      'Industrial and Provident Society',
      'Investment Company with Variable Capital(Umbrella)',
      'Limited Liability Partnership', 'Limited Partnership',
      'Old Public Company', 'Other Company Type', 'Other c',
      'PRI/LBG/NSC (Private, Limited by guarantee, no shar',
      'PRI/LTD BY GUAR/NSC (Private, limited by guarantee,
      'PRIV LTD SECT. 30 (Private limited company, section
      'Private Limited Company', 'Private Unlimited',
      'Private Unlimited Company', 'Public Limited Company
      'Registered Society'],
dtype='object')
```

```
df['CompanyCategory_cat'].cat.codes

<table>
<thead>
<tr>
<th>CompanyNumber</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>08209948</td>
<td>12</td>
</tr>
<tr>
<td>11399177</td>
<td>12</td>
</tr>
<tr>
<td>11743365</td>
<td>12</td>
</tr>
<tr>
<td>12402527</td>
<td>12</td>
</tr>
<tr>
<td>12234705</td>
<td>12</td>
</tr>
</tbody>
</table>
```

By [ian]@ianozsvald[.com]
Ian Ozsváld
Categoricals – over 10x speed up (on this data)!

```python
%timeit df['CompanyCategory'].value_counts()
485 ms ± 52.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```python
%timeit df['CompanyCategory_cat'].value_counts() # HUGE SAVING
28.6 ms ± 3.91 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```
Categoricals – index queries faster!

df2_nocat = df.set_index('CompanyCategory')
df2_cat = df.set_index('CompanyCategory_cat')

%timeit (df2_nocat.index == 'Private Limited Company')
281 ms ± 3.35 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

%timeit (df2_cat.index == 'Private Limited Company') # HUGE SAVING!
569 µs ± 223 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Circa 500x speed-up!

By [ian]@ianozsvald[.com]
float64 is default and a bit expensive

Company ages for still-alive entities
Note >50 years is not unusual

```
now = datetime.datetime.now()
age_deltas = now - df.IncorporationDate
df['age_years'] = age_deltas.dt.days / 360
```
float32 “half-price” and a bit faster

ser64 = df.age_years
show_size(ser64)
ser32 = ser64.astype("float32")
show_size(ser32)
ser16 = ser64.astype("float16")  # NOTE hardware _interpreted_ so SLOW
show_size(ser16)

Type float64 Range 0.0528 - 190.8972, float64, 36,208,992 bytes
Type float32 Range 0.0528 - 190.8972, float32, 18,104,496 bytes
Type float16 Range 0.0528 - 190.8750, float16, 9,052,248 bytes
Make choices to save RAM

```python
df_original_ram = df[['CompanyCategory', 'age_years']]
df_original_ram.info(memory_usage="deep")
```

<table>
<thead>
<tr>
<th>Column</th>
<th>Dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>0 CompanyCategory</td>
<td>object</td>
</tr>
<tr>
<td>1 age_years</td>
<td>float64</td>
</tr>
</tbody>
</table>

Dtypes: float64(1), object(1)

Memory usage: **827.7 MB**

```python
df_smaller_ram = df[['CompanyCategory_cat', 'age_years_f32']]
df_smaller_ram.info(memory_usage="deep")
```

<table>
<thead>
<tr>
<th>Column</th>
<th>Dtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>0 CompanyCategory_cat</td>
<td>category</td>
</tr>
<tr>
<td>1 age_years_f32</td>
<td>float32</td>
</tr>
</tbody>
</table>

Dtypes: category(1), float32(1)

Memory usage: **462.2 MB**

Including the index (previously we ignored it) we still save circa 50% RAM so you can fit in more rows of data.
“**dtype_diet**” gives you advice

```python
res = dtype_diet.report_on_dataframe(df_sample)  drop(columns=['col'])
res['saved_mb'] = ((res.current_nbytes - res.nbytes) / 1_000_000).astype('int')
res['shrunken_to'] = (res.nbytes / res.current_nbytes * 100).astype('int')
res.style.format({'nbytes': '{:,}','current_nbytes': '{:,}','saved_mb': '{}MB','shrunken_to': '{}%'})
```

Smallest non-breaking conversion per column:

<table>
<thead>
<tr>
<th>column</th>
<th>dtype</th>
<th>nbr_different</th>
<th>nbytes</th>
<th>current_nbytes</th>
<th>saved_mb</th>
<th>shrunken_to</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompanyName</td>
<td>category</td>
<td>0</td>
<td>552,546,573</td>
<td>366,681,836</td>
<td>-185MB</td>
<td>150%</td>
</tr>
<tr>
<td>CompanyCategory</td>
<td>category</td>
<td>0</td>
<td>4,528,328</td>
<td>369,713,766</td>
<td>365MB</td>
<td>1%</td>
</tr>
<tr>
<td>Accounts.AccountRefDay</td>
<td>float16</td>
<td>0</td>
<td>9,052,248</td>
<td>36,208,992</td>
<td>27MB</td>
<td>25%</td>
</tr>
<tr>
<td>Accounts.AccountRefDay</td>
<td>float32</td>
<td>0</td>
<td>18,104,496</td>
<td>36,208,992</td>
<td>18MB</td>
<td>50%</td>
</tr>
</tbody>
</table>
Drop to NumPy if you know you can

```python
timeit
df['age_years'].sum()
```

19.1 ms ± 1.58 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

```python
timeit
df['age_years'].values.sum()
# 10X SAVING DROPPING - IF YOU _KNOW_ YOU CAN (see slides for some details)
```

2.44 ms ± 68.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Caveat – Pandas mean is not np mean, the fair comparison is to np nanmean which is slower – see my blog or PyDataAmsterdam 2020 talk for details.
NumPy vs Pandas overhead (ser.sum())

25 files, 83 functions
Very few NumPy calls!

Thanks!

By [ian]@ianozsvald[.com]

Ian Ozsváld
Overhead...

By [ian]@ianozsvald[.com]
Overhead with `ser.values.sum()`

18 files, 51 functions
Many fewer Pandas calls (but still a lot!)
Is Pandas unnecessarily slow – NO!

In [15]: pd.options.compute.use_bottleneck=False

In [16]: %timeit ser.mean()
54.4 ms ± 228 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [17]: pd.options.compute.use_bottleneck=True

In [18]: %timeit ser.mean()
16.7 ms ± 30.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [19]: %timeit ser.values.mean()
5.05 ms ± 75.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

https://github.com/pandas-dev/pandas/issues/34773 - the truth is a bit complicated!

By [ian]@ianozsvald[.com]
Being highly performant

• Install optional (but great!) Pandas dependencies
  – bottleneck
  – numexpr

• Investigate https://github.com/ianozsvald/dtype_diet

• Investigate my ipython_memory_usage (PyPI/Conda)
Pure Python is “slow” and expressive

Deliberately poor function – pretend this is clever but slow!

```python
def won_huge_project(age):
    """Older companies have a higher chance of winning a low-probability big project""
    flag = False
    for n in range(int(age)):
        if random.uniform(0, 1) > 0.9:
            flag = True
    return flag
```

```
%timeit df['age_years'].apply(won_huge_project)
13.2 s ± 21.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
<table>
<thead>
<tr>
<th>IncorporationDate</th>
<th>age_years</th>
<th>won_huge_prj</th>
</tr>
</thead>
<tbody>
<tr>
<td>08148215</td>
<td>2012-07-18</td>
<td>True</td>
</tr>
<tr>
<td>10596616</td>
<td>2017-02-02</td>
<td>False</td>
</tr>
</tbody>
</table>
```

```
def df_sample = df.sample(5)
ser = df_sample['age_years'].apply(won_huge_project) # example output
```
Compile to Numba judiciously

```python
from numba import njit

@njit()
def won_huge_project_numba(age):
    """Older companies have a higher chance
    of winning a low-probability big project""
    flag = False
    for n in range(int(age)):
        if random.uniform(0, 1) > 0.9:
            flag=True
    return flag

%timeit df['age_years'].apply(won_huge_project_numba)
1.5 s ± 2.05 ms per loop (mean ± std. dev. of 7 runs, 1 loop, best of 3)
```

Near 10x speed-up!
Parallelise with Dask for multi-core

```python
import dask.dataframe as dd
cols = ['age_years']
# NOTE the reset_index on text column!
ddf = dd.from_pandas(df[cols].reset_index(drop=True),
    npartitions=8, sort=False)
t1 = time.time()
result = ddf.apply(won_huge_project, axis=1,
    raw=True, meta=(None, 'bool'))
    .compute(scheduler='processes')
print(f"Took {time.time() - t1}\")
```

- Make plain-Python code multi-core
- Note I had to drop text index column due to speed-hit
- Data copy cost can overwhelm any benefits so (always) profile & time
**Being highly performant**

- Mistakes slow us down (PAY ATTENTION!)
  - Try **nullable** Int64 & boolean, forthcoming Float64
  - Write tests (unit & end-to-end)
  - Lots more material & my newsletter on my blog
    IanOzsvald.com
  - Time saving docs: 📜 ianozsvald / notes_to_self

By [ian]@ianozsvald[.com]
Vaex / Modin

- Memory mapped & lazy computation
  - New string dtype (RAM efficient)

- Modin sits on Pandas, new “algebra” for dfs
  - Drop in replacement, easy to try

See talks on my blog: “Flying Pandas” and “Making Pandas Fly” – virtual talks this weekend on faster data processing with Pandas, Modin, Dask and Vaex
Summary

• Make it right then make it fast
• Think about **being** performant
• See blog for my classes
• I’d love a postcard if you learned something new!
Covid 19’s effect on UK Economy?

14 day rolling mean of UK company registrations (Note extant companies only, dissolved are censored)

Sharp decline in corporate registration after Lockdown - then apparent surge (perhaps just backed-up paperwork?). Will the recovery “last”? All open data, you can do similar things!