Photo by Hunter Harritt on Unsplash

Strategies for working with data as it grows

Conf42: Python 2022

With @gvanrossum in #pycon2018

Marco Carranza
Technical Co-Founder Teamcore

@mccrnz
www.teamcore.net

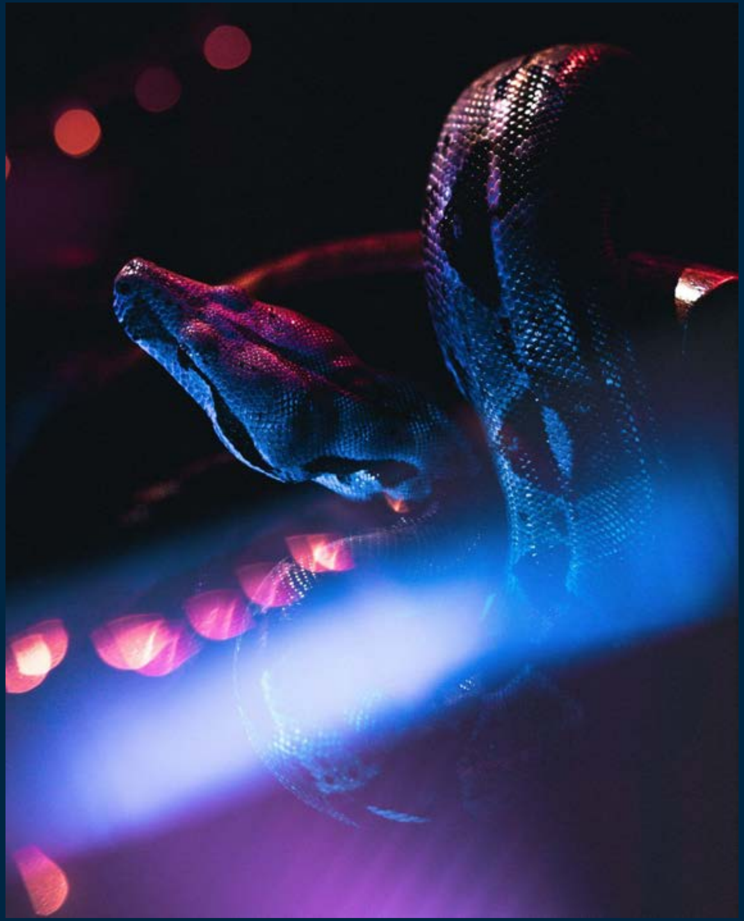


Photo by Nick Rickert on Unsplash

Agenda

Introduction

Pandas Tricks for memory control

Vertical scaling with Jupyter + Cloud

Processing larger datasets with Vaex

Speed Up Pandas with Modin

All-in with Pyspark

Introduction

Photo by Ussama Azam on Unsplash

Data is getting bigger and bigger, making it almost impossible to processed it in desktop machines.

A lot of new technologies (Hadoop, Spark, Presto, Dask, etc.)

Multiple challenges that requires combining multiple technologies and building Data Pipelines.

Pandas Tricks for memory control

Trick #1 - Sparse data structures

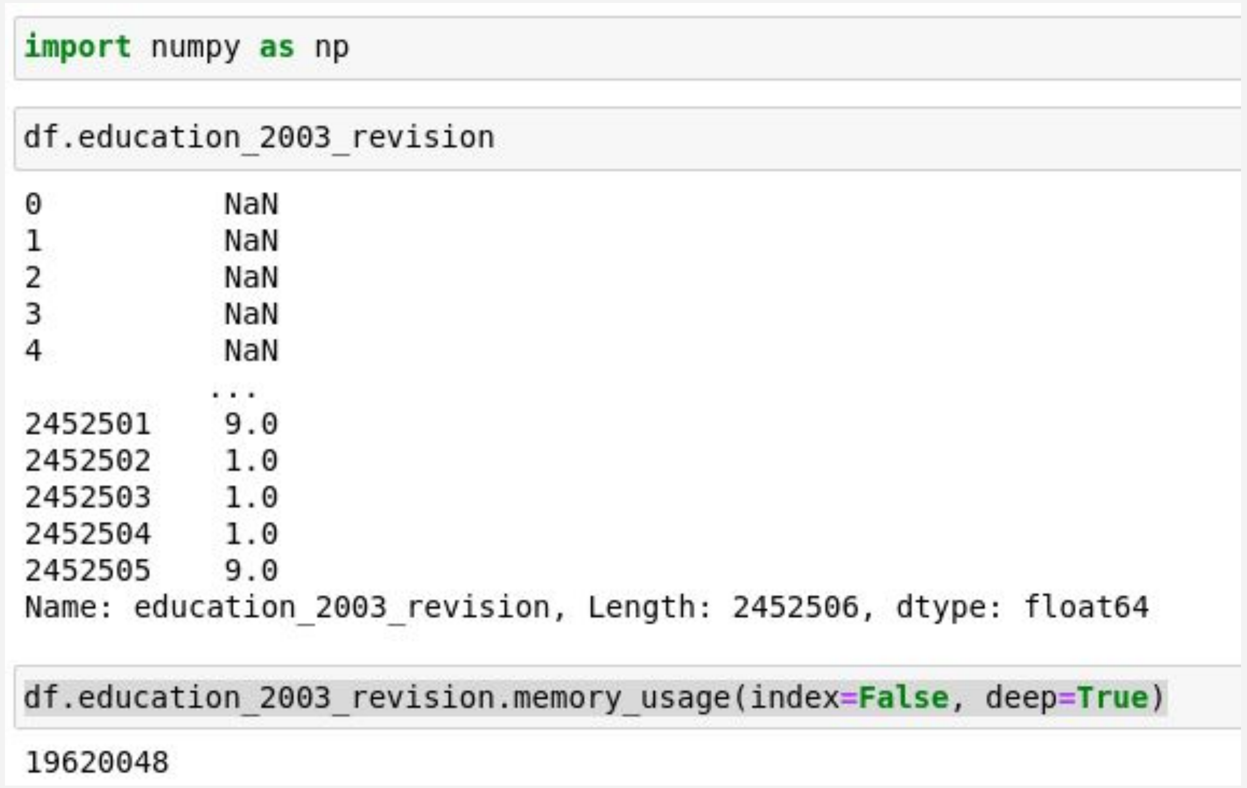
```
          requests.get(url)
    # checking response.status_code (if you ge
    if response.status_code != 200:
        print(f"Status: {response.status_code
    else:
        print(f"Status: {response.status_code

    # using BeautifulSoup to parse the response
    soup = BeautifulSoup(response.content, "htm

    # finding Post images in the soup
    images = soup.find_all("img", attrs={"alt":

    # downloading images
```

Photo by Artturi Jalli on Unsplash

Sometimes datasets comes with many empty values, usually represented as NaN values.

Using a sparse column representation could help us save some memory.

Sparse objects uses much less memory on disk (pickled) and in the Python interpreter.



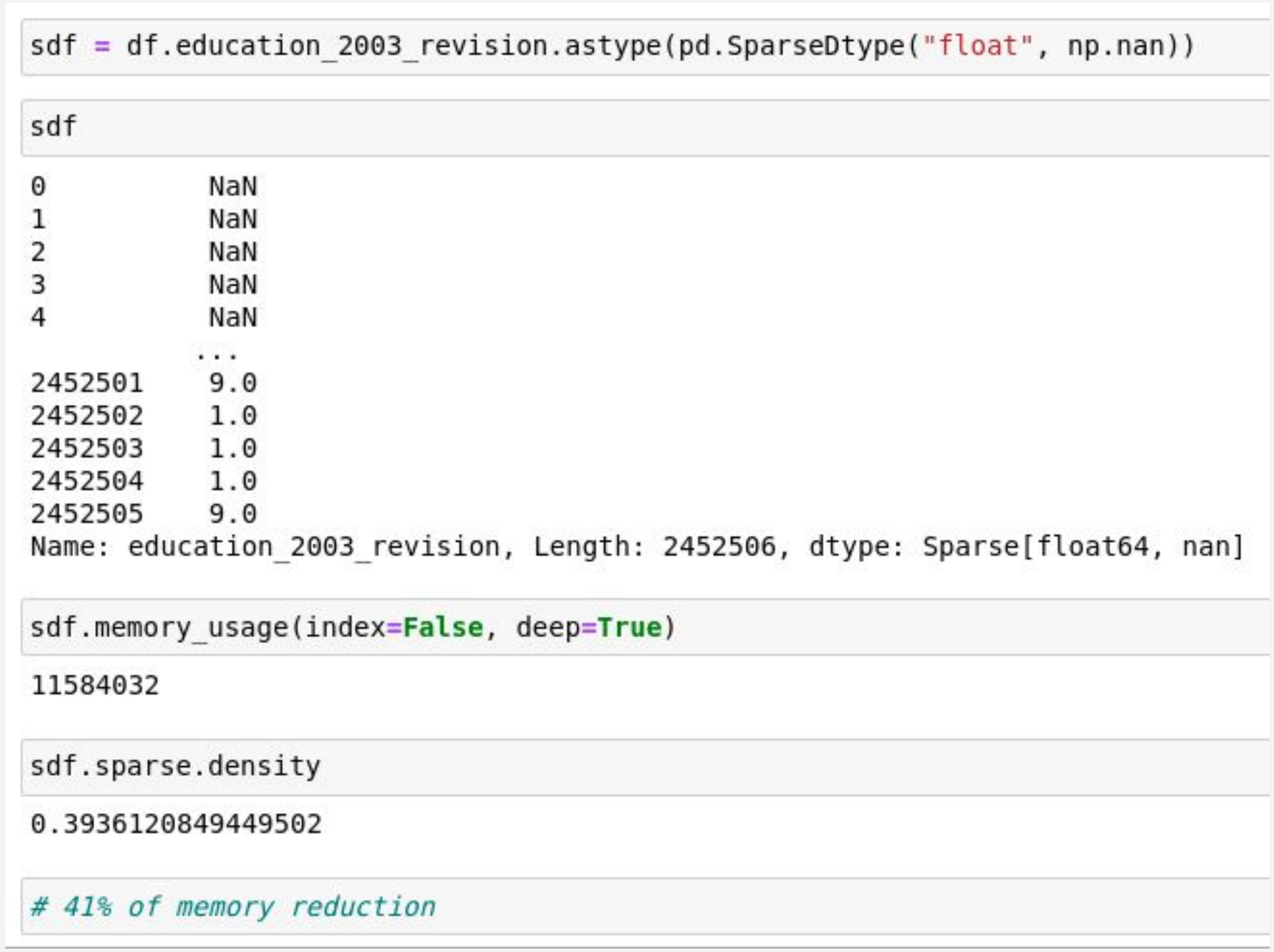
```
import numpy as np
```

```
df.education_2003_revision
```

```
0          NaN
1          NaN
2          NaN
3          NaN
4          NaN
          ...
2452501    9.0
2452502    1.0
2452503    1.0
2452504    1.0
2452505    9.0
Name: education_2003_revision, Length: 2452506, dtype: float64
```

```
df.education_2003_revision.memory_usage(index=False, deep=True)
```

```
19620048
```



```
sdf = df.education_2003_revision.astype(pd.SparseDtype("float", np.nan))
```

```
sdf
```

```
0          NaN
1          NaN
2          NaN
3          NaN
4          NaN
          ...
2452501    9.0
2452502    1.0
2452503    1.0
2452504    1.0
2452505    9.0
Name: education_2003_revision, Length: 2452506, dtype: Sparse[float64, nan]
```

```
sdf.memory_usage(index=False, deep=True)
```

```
11584032
```

```
sdf.sparse.density
```

```
0.3936120849449502
```

```
# 41% of memory reduction
```

Trick #2 - Sampling

Photo by Ilya Pavlov on Unsplash

Sampling data is very useful when you are working with a large dataset.

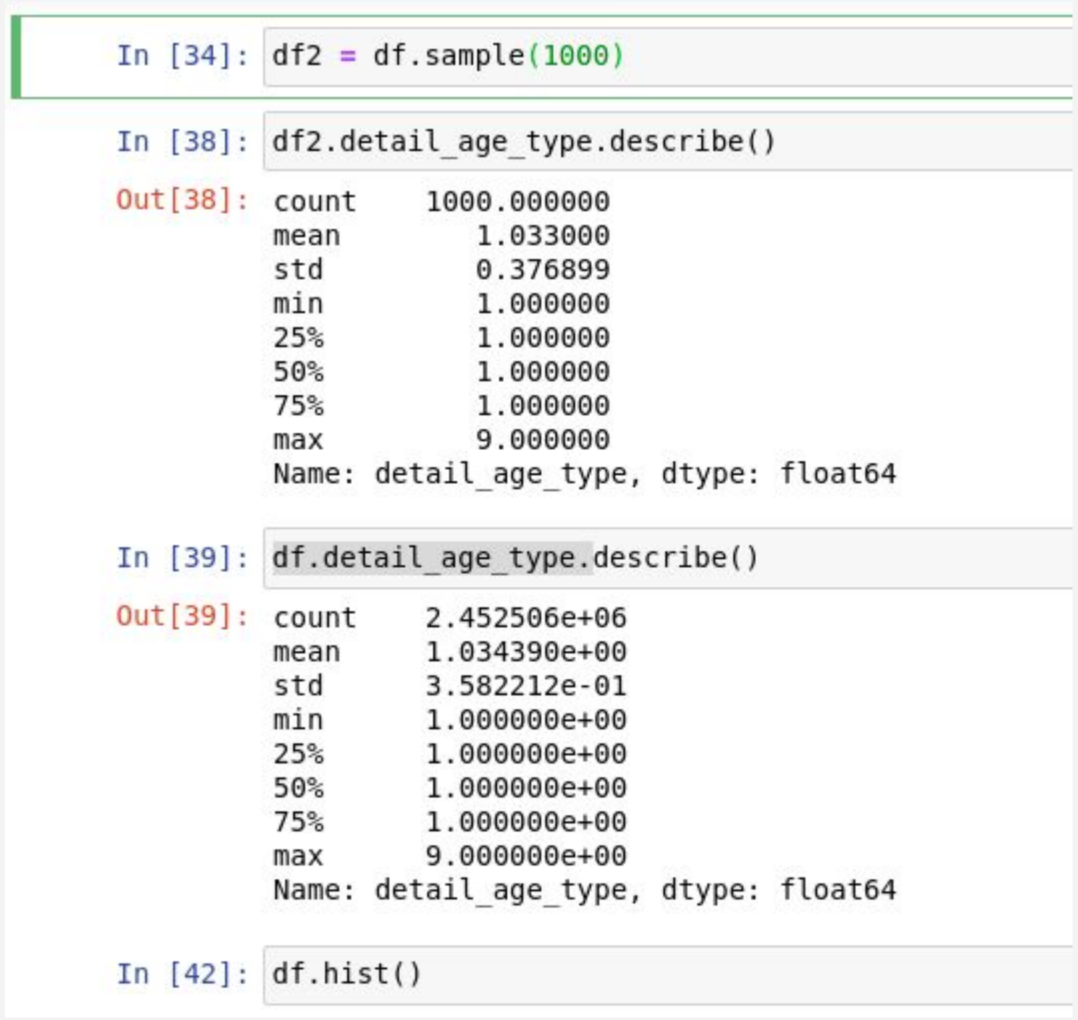
Sample the data representatively can help you work with a much smaller dataset,

In most of the cases the analysis will run faster without sacrificing the quality of the results.

pandas.DataFrame.sample

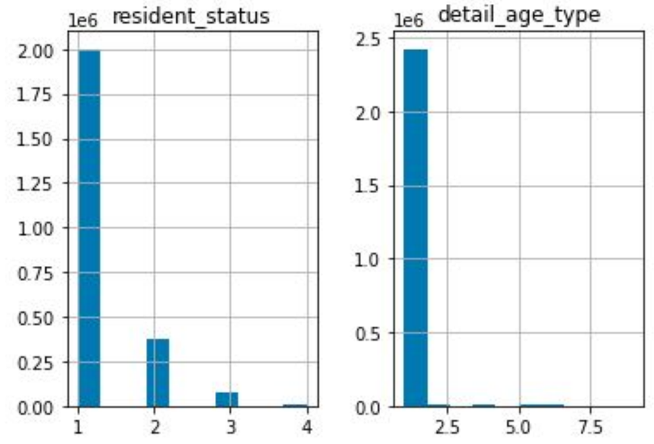
https://pandas.pydata.org/pandas-docs/stable /reference/api/pandas.DataFrame.sample.html

DataFrame.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None, ignore_index=False)

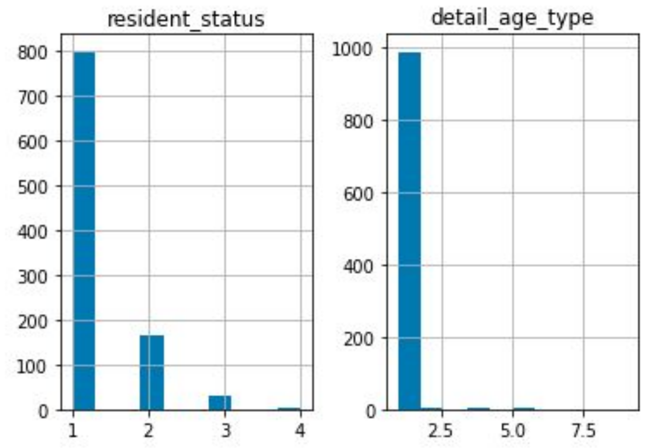


```
In [34]: df2 = df.sample(1000)

In [38]: df2.detail_age_type.describe()

Out[38]: count    1000.000000
         mean        1.033000
         std         0.376899
         min         1.000000
         25%         1.000000
         50%         1.000000
         75%         1.000000
         max         9.000000
         Name: detail_age_type, dtype: float64

In [39]: df.detail_age_type.describe()

Out[39]: count    2.452506e+06
         mean     1.034390e+00
         std      3.582212e-01
         min      1.000000e+00
         25%      1.000000e+00
         50%      1.000000e+00
         75%      1.000000e+00
         max      9.000000e+00
         Name: detail_age_type, dtype: float64

In [42]: df.hist()
```

```
In [42]: df.hist()

Out[42]: array([[<AxesSubplot:title={'center':'resident_status'}>,
                <AxesSubplot:title={'center':'detail_age_type'}>]], dtype=object)
```



```
In [43]: df2.hist()

Out[43]: array([[<AxesSubplot:title={'center':'resident_status'}>,
                <AxesSubplot:title={'center':'detail_age_type'}>]], dtype=object)
```



Trick #3 - Load only the columns that you need

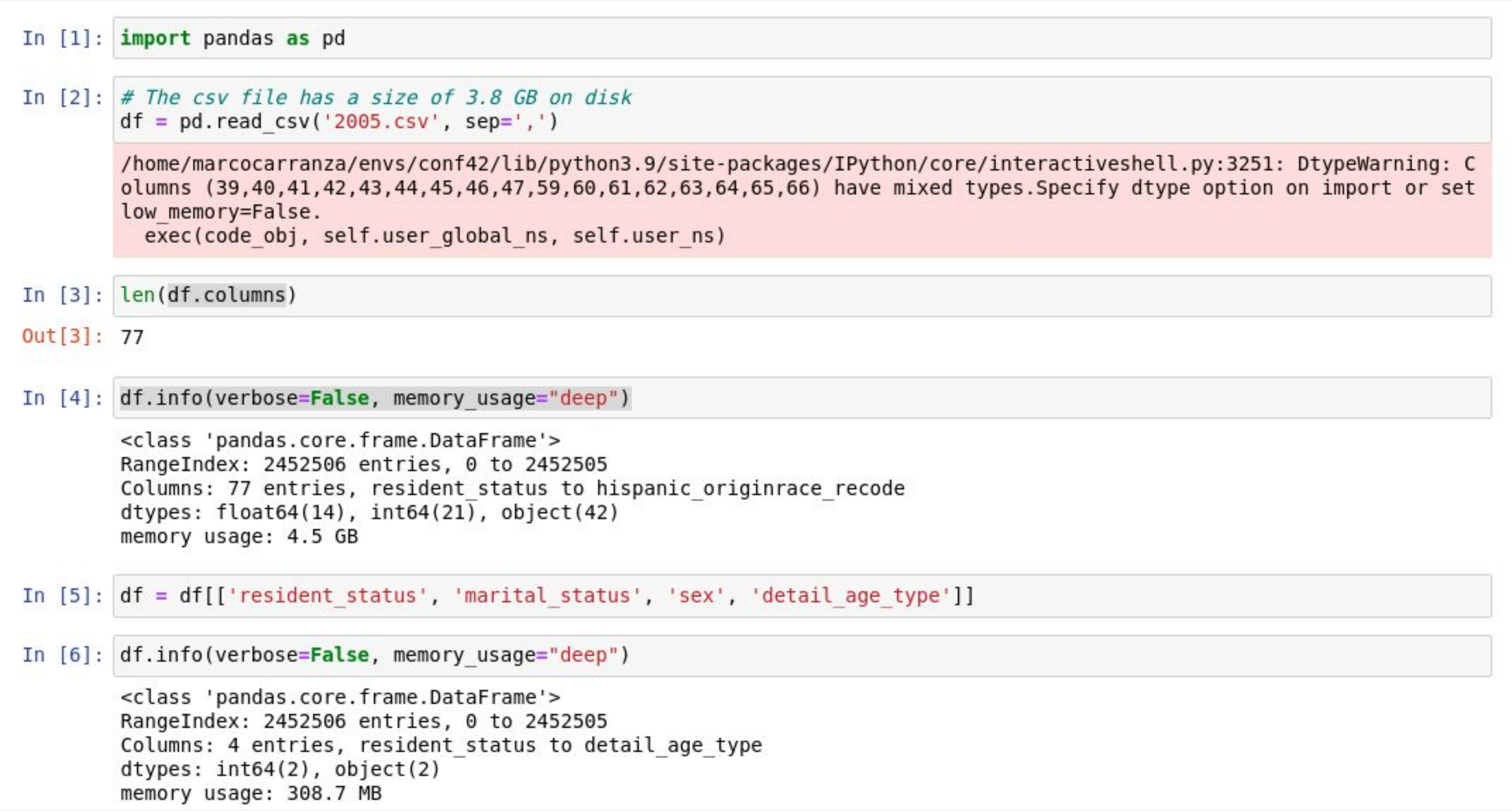
```
attachEvent("onreadystatechang
oolean Number String Function
_={};function F(e){var t=_[e]
t[1])===!1&&e.stopOnFalse){r=!
?o=u.length:r&&(s=t,c(r))}retu
nction(){return u=[],this},dis
re:function(){return p.fireWit
ending",r={state:function(){re
romise)?e.promise().done(n.res
dd(function(){n=s},t[1^e][2].d
=0,n=h.call(arguments),r=n.len
(r),l=Array(r);r>t;t++)n[t]&&b
/><table></table><a href='/a'>
TagName("input")[0],r.style.c
test(r.getAttribute("style")),
```

Photo by Markus Spiske on Unsplash

Some data sources include too many columns.

If you're not going to use all the columns, there's no need to load them

Less columns = Less memory



```
In [1]: import pandas as pd

In [2]: # The csv file has a size of 3.8 GB on disk
        df = pd.read_csv('2005.csv', sep=',')

        /home/marcocarranza/envs/conf42/lib/python3.9/site-packages/IPython/core/interactiveshell.py:3251: DtypeWarning: C
        olumns (39,40,41,42,43,44,45,46,47,59,60,61,62,63,64,65,66) have mixed types.Specify dtype option on import or set
        low_memory=False.
          exec(code_obj, self.user_global_ns, self.user_ns)

In [3]: len(df.columns)

Out[3]: 77

In [4]: df.info(verbose=False, memory_usage="deep")

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 2452506 entries, 0 to 2452505
        Columns: 77 entries, resident_status to hispanic_originrace_recode
        dtypes: float64(14), int64(21), object(42)
        memory usage: 4.5 GB

In [5]: df = df[['resident_status', 'marital_status', 'sex', 'detail_age_type']]

In [6]: df.info(verbose=False, memory_usage="deep")

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 2452506 entries, 0 to 2452505
        Columns: 4 entries, resident_status to detail_age_type
        dtypes: int64(2), object(2)
        memory usage: 308.7 MB
```

Trick #4 - Change numerical columns with smaller dtypes

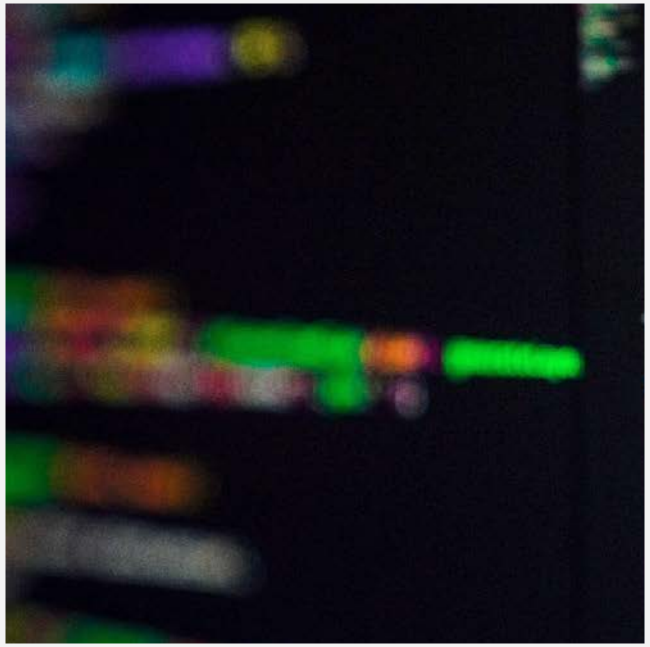
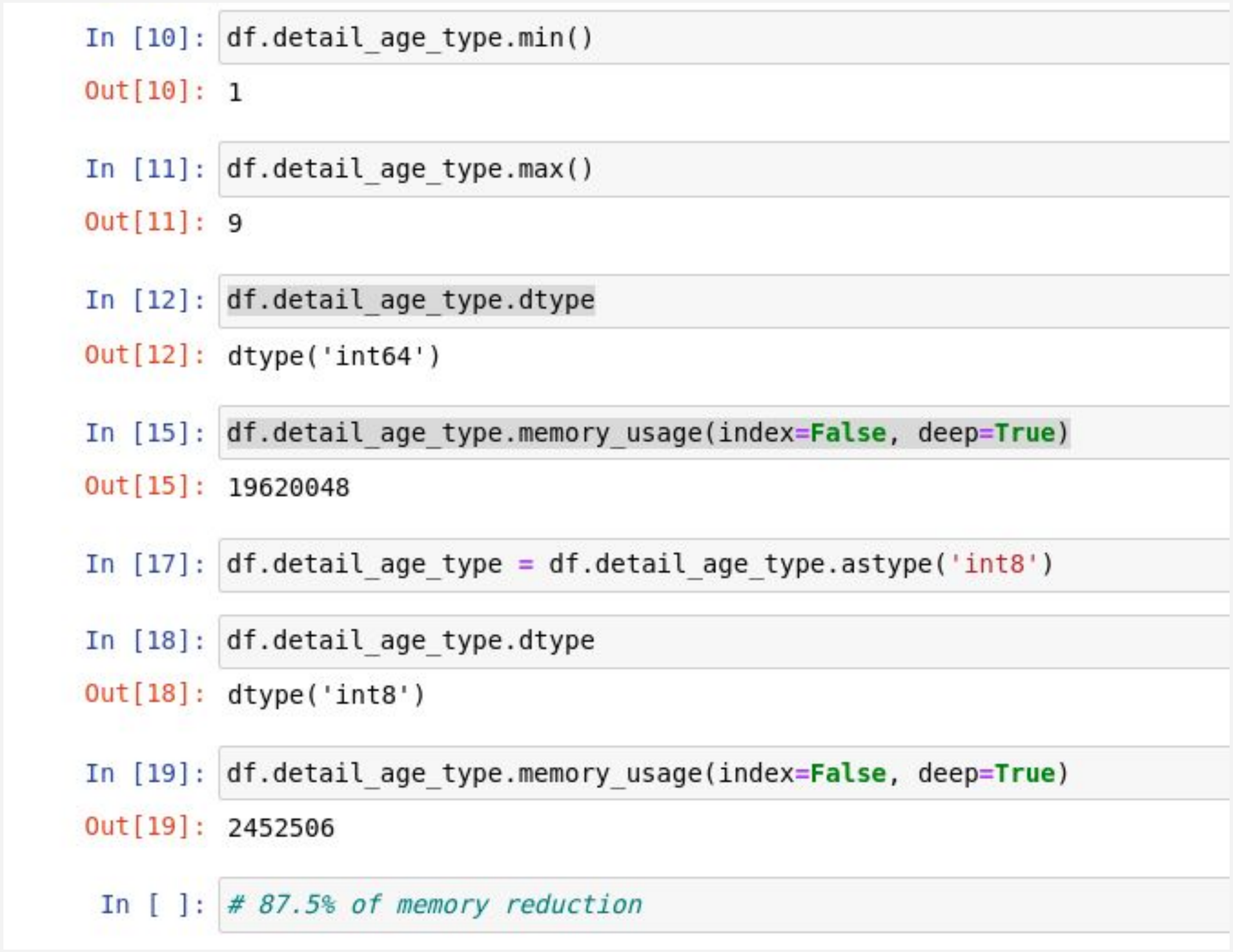


Photo by Artturi Jalli on Unsplash

Numerical types can store different range of numbers.

int8 can store integers from -128 to 127.
int16 can store integers from -32768 to 32767.
int64 can store integers from -9223372036854775808 to 9223372036854775807.

Pandas always try to guess the dtype



```
In [10]: df.detail_age_type.min()

Out[10]: 1

In [11]: df.detail_age_type.max()

Out[11]: 9

In [12]: df.detail_age_type.dtype

Out[12]: dtype('int64')

In [15]: df.detail_age_type.memory_usage(index=False, deep=True)

Out[15]: 19620048

In [17]: df.detail_age_type = df.detail_age_type.astype('int8')

In [18]: df.detail_age_type.dtype

Out[18]: dtype('int8')

In [19]: df.detail_age_type.memory_usage(index=False, deep=True)

Out[19]: 2452506

In [ ]: # 87.5% of memory reduction
```

Trick #5 - Use Categorical dtypes

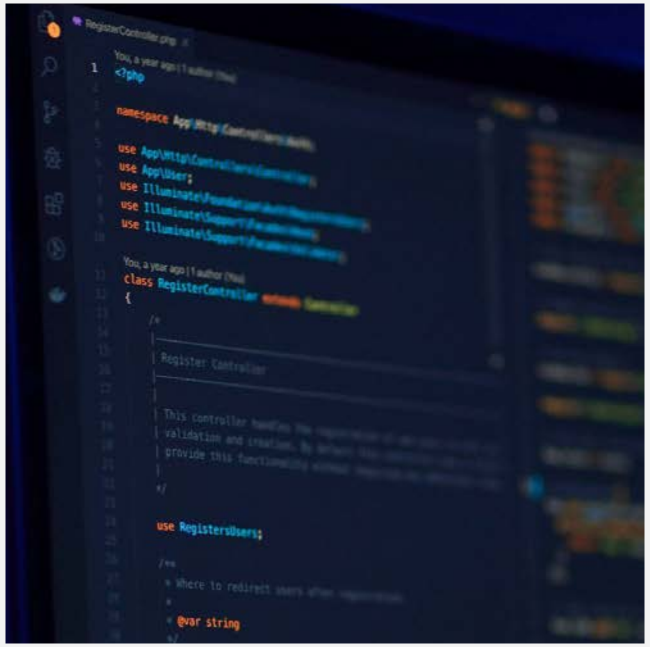
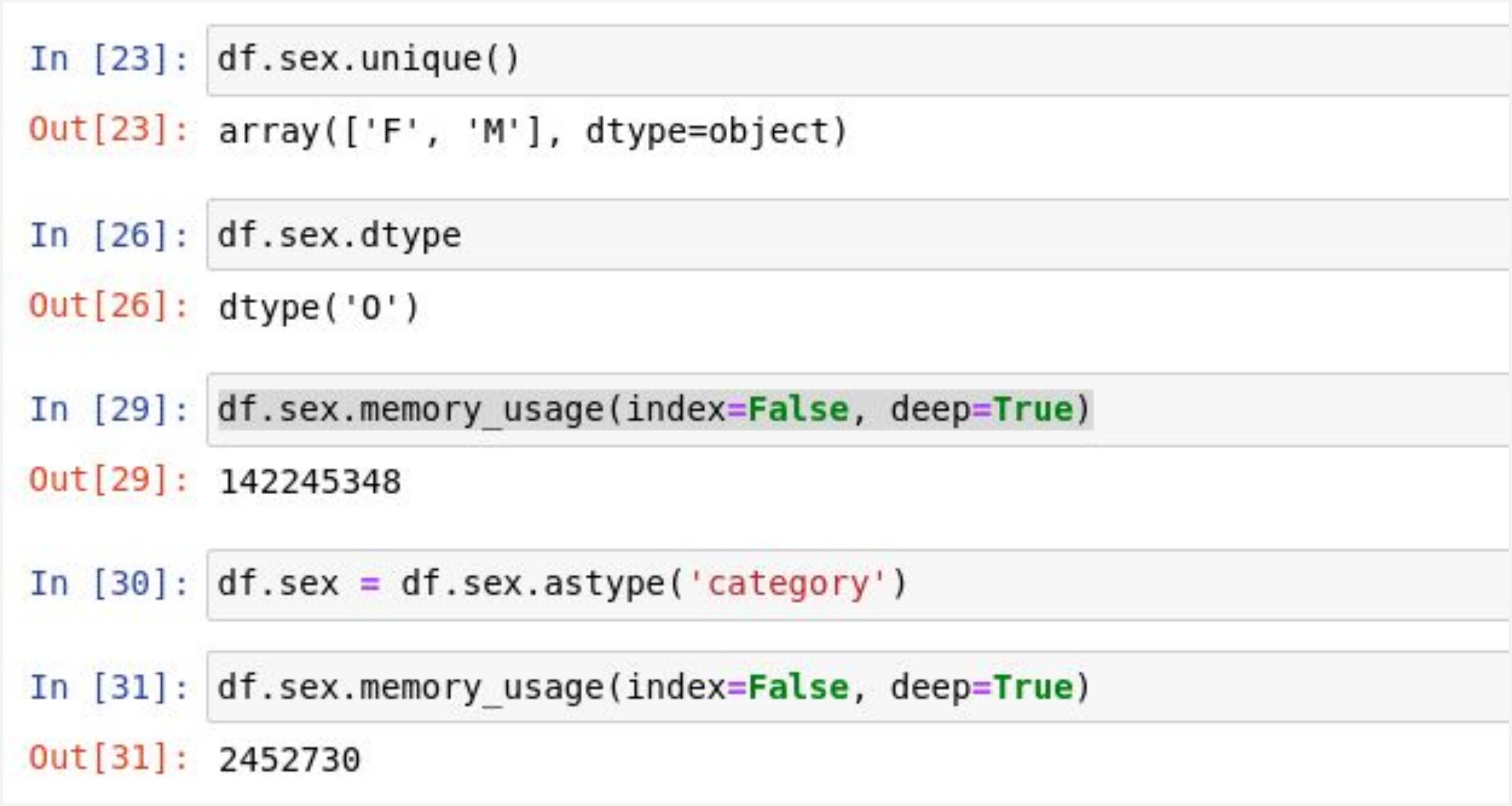


Photo by Mohammad Rahmani on Unsplash

In some cases is possible to shrink non-numerical data and reduce the memory footprint.

Pandas has a custom categorical Dtype for this cases.

https://pandas.pydata.org/pandas-docs/stable/user_guide/categorical.html



```
In [23]: df.sex.unique()
Out[23]: array(['F', 'M'], dtype=object)

In [26]: df.sex.dtype
Out[26]: dtype('O')

In [29]: df.sex.memory_usage(index=False, deep=True)
Out[29]: 142245348

In [30]: df.sex = df.sex.astype('category')

In [31]: df.sex.memory_usage(index=False, deep=True)
Out[31]: 2452730
```

98% of memory reduccion

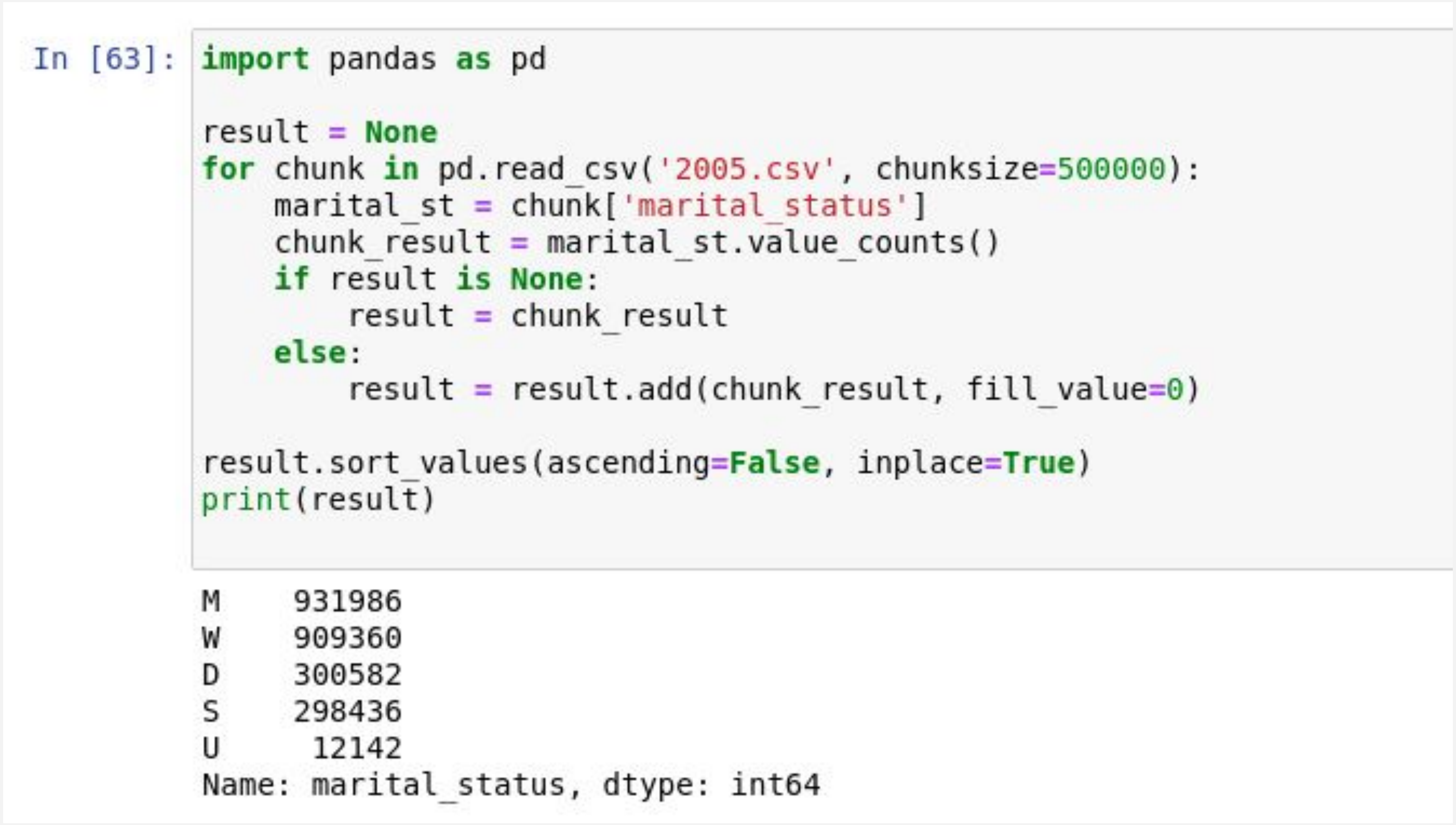
Trick #6- Reading data by chunks

Photo by Joan Gamell on Unsplash

You can load only part of the file into memory at any given time by loading and then processing the data in chunks.

This will prevent your code crashing if there's not enough memory.

Reading files by chunks helps process large files that will not fit into memory.



```
In [63]: import pandas as pd

         result = None
         for chunk in pd.read_csv('2005.csv', chunksize=500000):
             marital_st = chunk['marital_status']
             chunk_result = marital_st.value_counts()
             if result is None:
                 result = chunk_result
             else:
                 result = result.add(chunk_result, fill_value=0)

         result.sort_values(ascending=False, inplace=True)
         print(result)
```

```
M    931986
W    909360
D    300582
S    298436
U     12142
Name: marital_status, dtype: int64
```

Vertical Scaling with jupyter and the cloud

Vertical scaling vs Horizontal Scaling

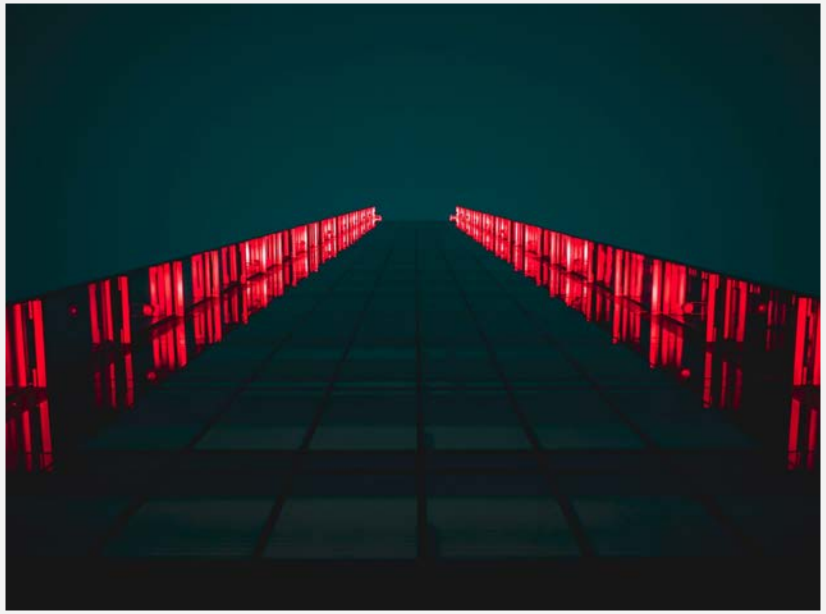


Photo by Tanner Boriack on Unsplash

Vertical scaling is the ability to increase the capacity of existing hardware or software by adding resources. (CPU, Memory, etc.)

Horizontal scaling involves adding machines in the pool of existing resources.

Jupyter + Cloud

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.

Very easy to run code on the cloud.

Machines of multiple sizes (+1TB ram)

Jupyter + Cloud

PRO

No code changes needed.

Easy, if using cloud tools. (*Binder, Kaggle Kernels, Google Colab, Azure Notebooks, CoCalc, Datalore, etc.*)

Good for testing, data cleaning and visualization.

You pay only for what you use (*If you don't forget to turn off your VM!*)

CONS

Expensive in the long run. Not optimized.

Does no escales very well.

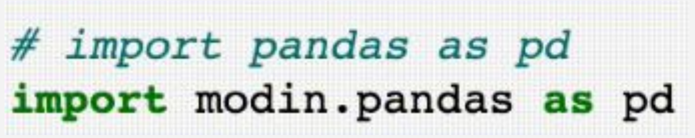
Not production ready

Speed Up Pandas with Modin

Modin

Scale your pandas workflow by changing a single line of code.

Multiprocess Dataframe library with an identical API to pandas that allows users to speed up their Pandas workflows



```
# import pandas as pd
import modin.pandas as pd
```

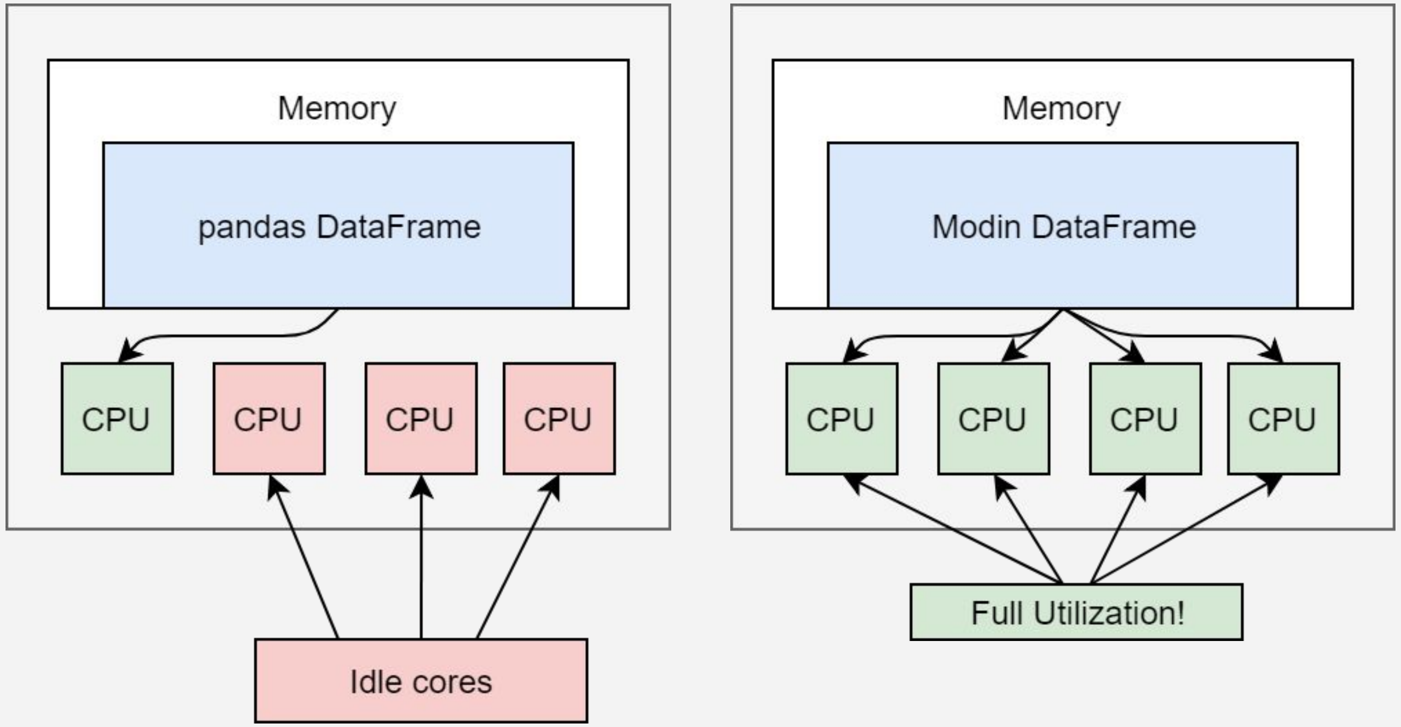
Accelerates Pandas queries by 4x on an 8-core machine, only requires to change a single line of code.

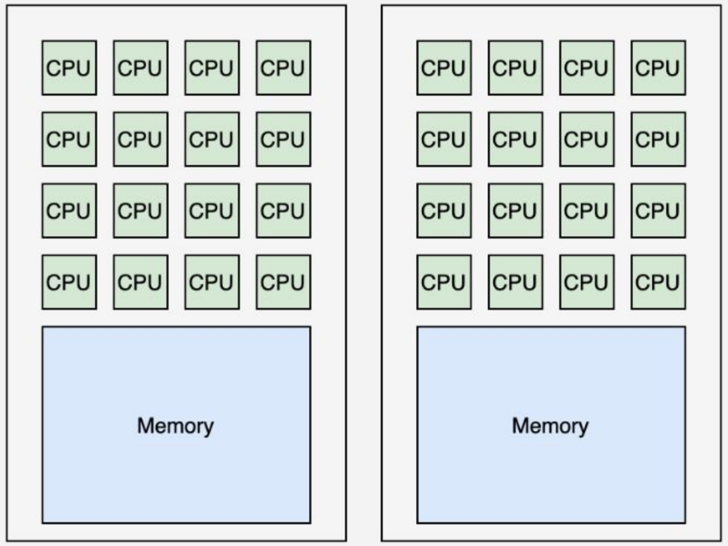
pip install modin

Modin vs. Pandas

Pandas implementation is inherently single-threaded. This means that only one of your CPU cores can be utilized at any given time.

Modin's implementation enables you to use all of the cores on your machine, or all of the cores in an entire cluster.





Modin

PRO

Unlocks all the CPU power

Only one import is needed, so no changes in the code are needed.

Really fast when reading data.

Compute engines available to distribute the calculations on a cluster with Dask or Ray.

CONS

Extra effort depending of the compute engine setup (Dask / Ray) + clusters

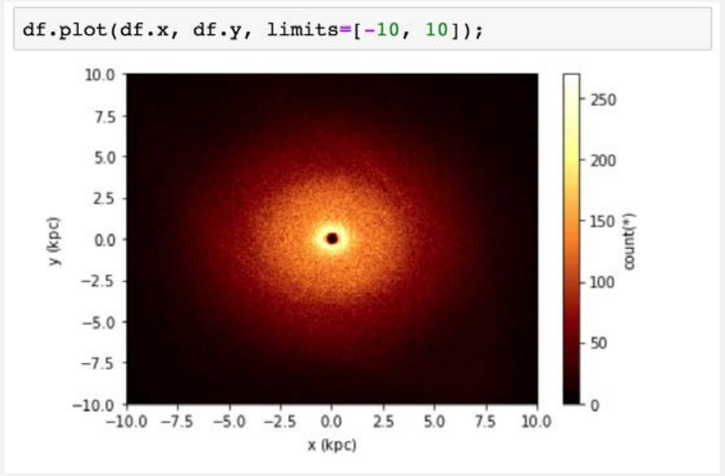
Distributed systems are complex

Requires a lot of memory as Pandas

Processing large datasets with Vaex

Vaex

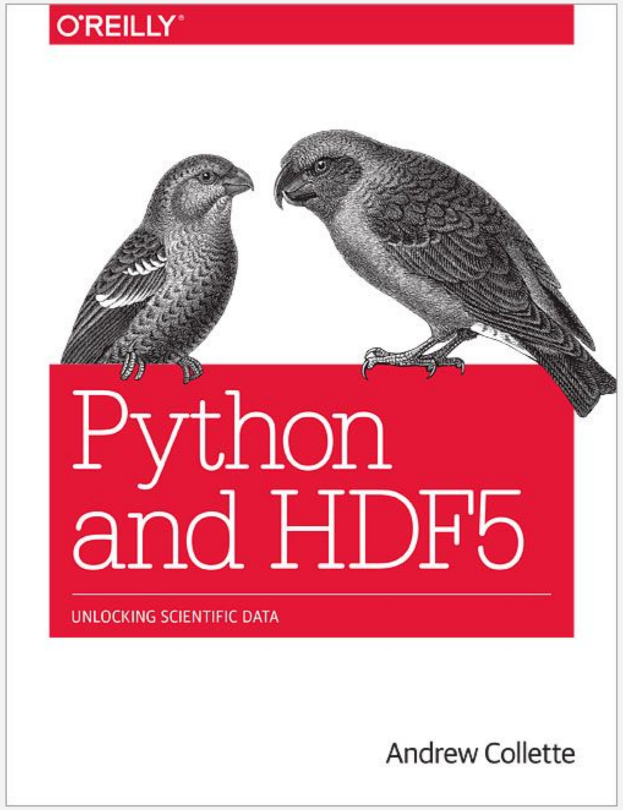
```
df.plot(df.x, df.y, limits=[-10, 10]);
```



Vaex is a Python library, with a similar syntax to Pandas, that help us work with large data-sets in machines with limited resources were the only limitation is the size of the hard drive.

Vaex provides memory-mapping, so it will never touch or copy the data to memory unless is explicitly requested.

Vaex + HDF5

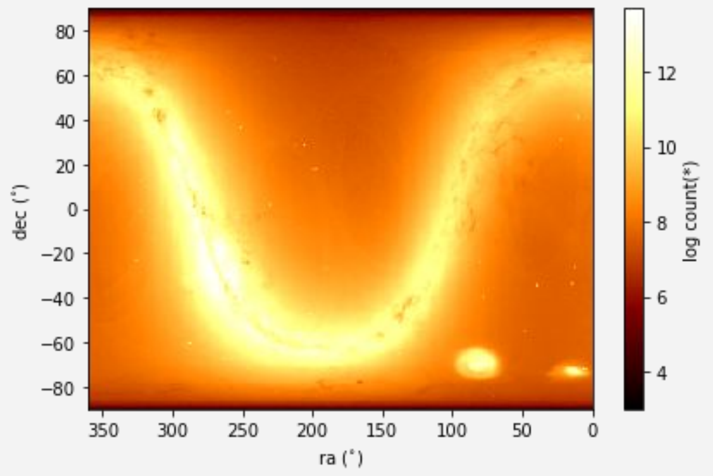


Is a multipurpose hierarchical container format capable of storing large numerical datasets with their metadata. The specification is open and the tools are open source.

Convert large CSV files to HDF5 on the fly (Memory mapping)

```
df = vaex.from_csv('./my_data/my_big_file.csv',
                   convert=True,
                   chunk_size=5_000_000)
```

Vaex API



Provides a dataframe Server so calculations and/or aggregations could run on a different computer than where the (aggregated) data is needed.

Python API (websockets) and REST API available

Vaex

PRO

Helps control memory usage with memory mapping (Amazing samples)

Computes on the fly (Lazy / Virtual columns)

Easy to build visualizations with datasets larger than memory

Machine learning algorithms available through vaex.ml package.

Can export data to a Pandas Dataframe

CONS

Need modification in the code, syntax similar to Pandas

Not as mature as Pandas, but improving every day.

Tricky to work with exported HDf5 from pandas

All-in with Pyspark

Pyspark

When you need to work with a very large-scale data, its mandatory to distribute both the data and computations to a cluster. This can not be achieved with Pandas.

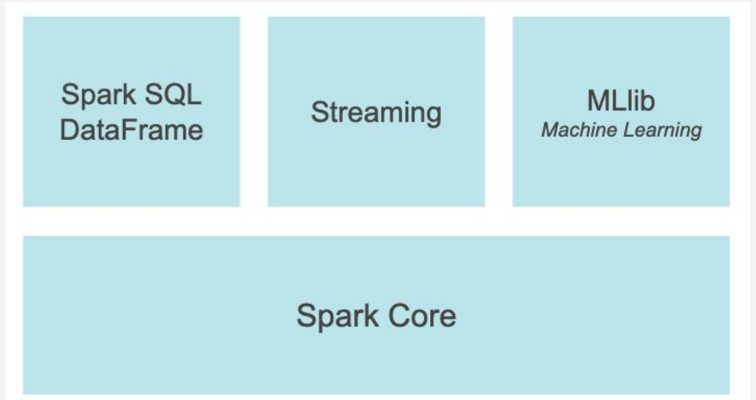
Spark is an analytics engine used for large-scale data processing. It lets you spread both data and computations over clusters to achieve a substantial performance increase.

Pyspark

PySpark is a Python API for Spark. It combines the simplicity of Python with the high performance of Spark.

Also provides the PySpark shell for interactively analyzing your data in a distributed environment.

PySpark supports most of Spark's features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core.



Pyspark

PRO

Great speed with large dataset.

Very rich and mature ecosystem. With a lot of libraries for Machine Learning, feature extraction and, transformations

Run on Hadoop alongside other tools in the Hadoop ecosystem

CONS

Need modification in the code, syntax is different to Pandas

Bad performance with small datasets, Pandas could be faster.

In Spark MLlib there are fewer algorithms present.

Spark requires huge RAM to process in memory, so is not very cost effective.

Final Notes

Multiple options to scale your workloads

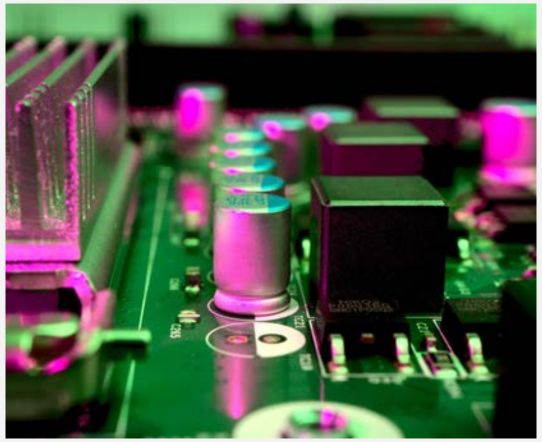


Photo by Michael Dziedzic on Unsplash

The easiest is to vertical scale your resources with Jupyter and a Cloud Provider, but first don't forget to optimize your dataframe.

There are some powerful alternatives to work with large datasets like Vaex.

If you need to process a huge amount of data, you can use Modin with Ray or Dask to distributed your workload.

Or you can rewrite your Pandas logic to make it run over Sparks Dataframes, and take advantage of many cloud providers PaaS.

“Premature optimization is the root of all evil”

Donald Knuth
The Art of Computer Programming

Thank you very much for your attention!

Marco Carranza
@mccrnz
https://github.com/marcocarranza/conf42_data_strategies