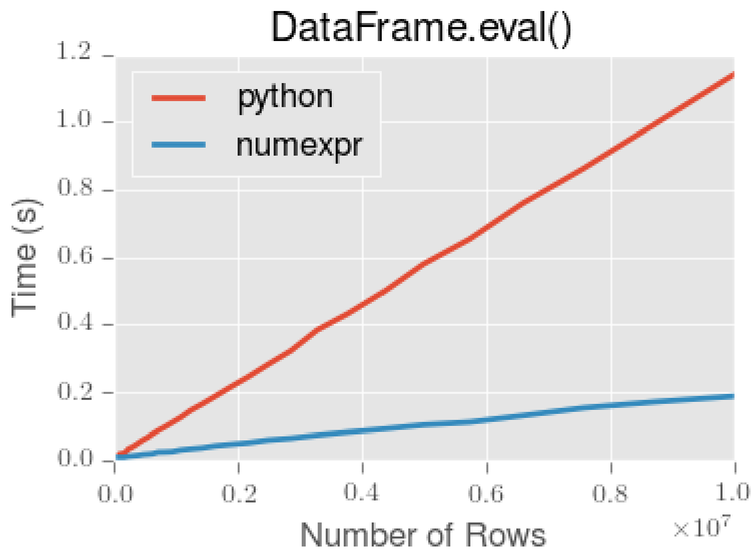
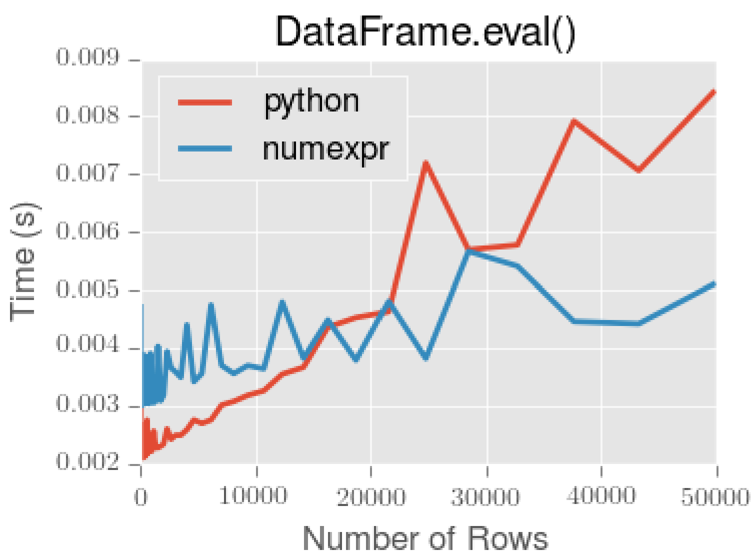


Note: Operations with smallish objects (around 15k-20k rows) are faster using plain Python:



This plot was created using a `DataFrame` with 3 columns each containing floating point values generated using `numpy.random.randn()`.

Technical minutia regarding expression evaluation

Expressions that would result in an object dtype or involve datetime operations (because of `NaT`) must be evaluated in Python space. The main reason for this behavior is to maintain backwards compatibility with versions of NumPy < 1.7. In those versions of NumPy a call to `ndarray.astype(str)` will truncate any strings that are more than 60 characters in length. Second, we can't pass `object` arrays to `numexpr` thus string comparisons must be evaluated in Python space.

The upshot is that this *only* applies to object-dtype expressions. So, if you have an expression–for example

```
In [64]: df = pd.DataFrame({'strings': np.repeat(list('cba'), 3),
   ....:                    'nums': np.repeat(range(3), 3)})
   ....:

In [65]: df
Out[65]:
  strings  nums
0       c     0
1       c     0
2       c     0
3       b     1
4       b     1
5       b     1
6       a     2
7       a     2
8       a     2

In [66]: df.query('strings == "a" and nums == 1')
Out[66]:
Empty DataFrame
Columns: [strings, nums]
Index: []
```

the numeric part of the comparison (`nums == 1`) will be evaluated by `numexpr`.

In general, *`DataFrame.query()`*/*`pandas.eval()`* will evaluate the subexpressions that *can* be evaluated by `numexpr` and those that must be evaluated in Python space transparently to the user. This is done by inferring the result type of an expression from its arguments and operators.

2.19 Scaling to large datasets

Pandas provides data structures for in-memory analytics, which makes using pandas to analyze datasets that are larger than memory datasets somewhat tricky. Even datasets that are a sizable fraction of memory become unwieldy, as some pandas operations need to make intermediate copies.

This document provides a few recommendations for scaling your analysis to larger datasets. It's a complement to *Enhancing performance*, which focuses on speeding up analysis for datasets that fit in memory.

But first, it's worth considering *not using pandas*. Pandas isn't the right tool for all situations. If you're working with very large datasets and a tool like PostgreSQL fits your needs, then you should probably be using that. Assuming you want or need the expressiveness and power of pandas, let's carry on.

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

In [2]: import numpy as np
```

2.19.1 Load less data

Suppose our raw dataset on disk has many columns:

```
                     id_0    name_0       x_0       y_0  id_1   name_1       x_1  ...␣
↪ name_8       x_8       y_8  id_9   name_9       x_9       y_9
timestamp                                                                         ...
2000-01-01 00:00:00  1015   Michael -0.399453  0.095427   994    Frank -0.176842  ...␣
↪    Dan -0.315310  0.713892  1025   Victor -0.135779  0.346801
2000-01-01 00:01:00   969  Patricia  0.650773 -0.874275  1003    Laura  0.459153  ...␣
↪ Ursula  0.913244 -0.630308  1047    Wendy -0.886285  0.035852
2000-01-01 00:02:00  1016    Victor -0.721465 -0.584710  1046  Michael  0.524994  ...␣
↪    Ray -0.656593  0.692568  1064   Yvonne  0.070426  0.432047
2000-01-01 00:03:00   939     Alice -0.746004 -0.908008   996   Ingrid -0.414523  ...␣
↪  Jerry -0.958994  0.608210   978    Wendy  0.855949 -0.648988
2000-01-01 00:04:00  1017       Dan  0.919451 -0.803504  1048    Jerry -0.569235  ...␣
↪  Frank -0.577022 -0.409088   994      Bob -0.270132  0.335176
...                   ...       ...       ...       ...   ...      ...       ...  ...␣
↪    ...       ...       ...   ...      ...       ...       ...
2000-12-30 23:56:00   999       Tim  0.162578  0.512817   973    Kevin -0.403352  ...␣
↪    Tim -0.380415  0.008097  1041  Charlie  0.191477 -0.599519
2000-12-30 23:57:00   970     Laura -0.433586 -0.600289   958   Oliver -0.966577  ...␣
↪  Zelda  0.971274  0.402032  1038   Ursula  0.574016 -0.930992
2000-12-30 23:58:00  1065     Edith  0.232211 -0.454540   971      Tim  0.158484  ...␣
↪  Alice -0.222079 -0.919274  1022      Dan  0.031345 -0.657755
2000-12-30 23:59:00  1019    Ingrid  0.322208 -0.615974   981   Hannah  0.607517  ...␣
↪  Sarah -0.424440 -0.117274   990   George -0.375530  0.563312
2000-12-31 00:00:00   937    Ursula -0.906523  0.943178  1018    Alice -0.564513  ...␣
↪  Jerry  0.236837  0.807650   985   Oliver  0.777642  0.783392

[525601 rows x 40 columns]
```

To load the columns we want, we have two options. Option 1 loads in all the data and then filters to what we need.

```
In [3]: columns = ['id_0', 'name_0', 'x_0', 'y_0']

In [4]: pd.read_parquet("timeseries_wide.parquet")[columns]
Out[4]:
                     id_0    name_0       x_0       y_0
timestamp
2000-01-01 00:00:00  1015   Michael -0.399453  0.095427
2000-01-01 00:01:00   969  Patricia  0.650773 -0.874275
2000-01-01 00:02:00  1016    Victor -0.721465 -0.584710
2000-01-01 00:03:00   939     Alice -0.746004 -0.908008
2000-01-01 00:04:00  1017       Dan  0.919451 -0.803504
...                   ...       ...       ...       ...
2000-12-30 23:56:00   999       Tim  0.162578  0.512817
2000-12-30 23:57:00   970     Laura -0.433586 -0.600289
2000-12-30 23:58:00  1065     Edith  0.232211 -0.454540
2000-12-30 23:59:00  1019    Ingrid  0.322208 -0.615974
2000-12-31 00:00:00   937    Ursula -0.906523  0.943178

[525601 rows x 4 columns]
```

Option 2 only loads the columns we request.

```
In [5]: pd.read_parquet("timeseries_wide.parquet", columns=columns)
Out[5]:
```

(continues on next page)

(continued from previous page)

```
                     id_0    name_0       x_0       y_0
timestamp
2000-01-01 00:00:00  1015   Michael -0.399453  0.095427
2000-01-01 00:01:00   969  Patricia  0.650773 -0.874275
2000-01-01 00:02:00  1016    Victor -0.721465 -0.584710
2000-01-01 00:03:00   939     Alice -0.746004 -0.908008
2000-01-01 00:04:00  1017       Dan  0.919451 -0.803504
...                   ...       ...       ...       ...
2000-12-30 23:56:00   999       Tim  0.162578  0.512817
2000-12-30 23:57:00   970     Laura -0.433586 -0.600289
2000-12-30 23:58:00  1065     Edith  0.232211 -0.454540
2000-12-30 23:59:00  1019    Ingrid  0.322208 -0.615974
2000-12-31 00:00:00   937    Ursula -0.906523  0.943178

[525601 rows x 4 columns]
```

If we were to measure the memory usage of the two calls, we'd see that specifying `columns` uses about 1/10th the memory in this case.

With `pandas.read_csv()`, you can specify `usecols` to limit the columns read into memory. Not all file formats that can be read by pandas provide an option to read a subset of columns.

2.19.2 Use efficient datatypes

The default pandas data types are not the most memory efficient. This is especially true for text data columns with relatively few unique values (commonly referred to as "low-cardinality" data). By using more efficient data types, you can store larger datasets in memory.

```
In [6]: ts = pd.read_parquet("timeseries.parquet")

In [7]: ts
Out[7]:
                       id      name         x         y
timestamp
2000-01-01 00:00:00  1029   Michael  0.278837  0.247932
2000-01-01 00:00:30  1010  Patricia  0.077144  0.490260
2000-01-01 00:01:00  1001    Victor  0.214525  0.258635
2000-01-01 00:01:30  1018     Alice -0.646866  0.822104
2000-01-01 00:02:00   991       Dan  0.902389  0.466665
...                   ...       ...       ...       ...
2000-12-30 23:58:00   992     Sarah  0.721155  0.944118
2000-12-30 23:58:30  1007    Ursula  0.409277  0.133227
2000-12-30 23:59:00  1009    Hannah -0.452802  0.184318
2000-12-30 23:59:30   978     Kevin -0.904728 -0.179146
2000-12-31 00:00:00   973    Ingrid -0.370763 -0.794667

[1051201 rows x 4 columns]
```

Now, let's inspect the data types and memory usage to see where we should focus our attention.

```
In [8]: ts.dtypes
Out[8]:
id          int64
name       object
x         float64
```

(continues on next page)

(continued from previous page)

```
y        float64
dtype: object
```

```
In [9]: ts.memory_usage(deep=True)  # memory usage in bytes
Out[9]:
Index     8409608
id        8409608
name     65537768
x         8409608
y         8409608
dtype: int64
```

The name column is taking up much more memory than any other. It has just a few unique values, so it's a good candidate for converting to a Categorical. With a Categorical, we store each unique name once and use space-efficient integers to know which specific name is used in each row.

```
In [10]: ts2 = ts.copy()

In [11]: ts2['name'] = ts2['name'].astype('category')

In [12]: ts2.memory_usage(deep=True)
Out[12]:
Index    8409608
id       8409608
name     1054102
x        8409608
y        8409608
dtype: int64
```

We can go a bit further and downcast the numeric columns to their smallest types using *pandas.to_numeric()*.

```
In [13]: ts2['id'] = pd.to_numeric(ts2['id'], downcast='unsigned')

In [14]: ts2[['x', 'y']] = ts2[['x', 'y']].apply(pd.to_numeric, downcast='float')

In [15]: ts2.dtypes
Out[15]:
id         uint16
name     category
x         float32
y         float32
dtype: object
```

```
In [16]: ts2.memory_usage(deep=True)
Out[16]:
Index    8409608
id       2102402
name     1054102
x        4204804
y        4204804
dtype: int64
```

```
In [17]: reduction = (ts2.memory_usage(deep=True).sum()
   ....:              / ts.memory_usage(deep=True).sum())
   ....:
```

(continues on next page)

(continued from previous page)

```
In [18]: print(f"{reduction:0.2f}")
0.20
```

In all, we've reduced the in-memory footprint of this dataset to 1/5 of its original size.

See *Categorical data* for more on `Categorical` and *dtypes* for an overview of all of pandas' dtypes.

2.19.3 Use chunking

Some workloads can be achieved with chunking: splitting a large problem like "convert this directory of CSVs to parquet" into a bunch of small problems ("convert this individual CSV file into a Parquet file. Now repeat that for each file in this directory."). As long as each chunk fits in memory, you can work with datasets that are much larger than memory.

Note: Chunking works well when the operation you're performing requires zero or minimal coordination between chunks. For more complicated workflows, you're better off *using another library*.

Suppose we have an even larger "logical dataset" on disk that's a directory of parquet files. Each file in the directory represents a different year of the entire dataset.

```
data
└── timeseries
    ├── ts-00.parquet
    ├── ts-01.parquet
    ├── ts-02.parquet
    ├── ts-03.parquet
    ├── ts-04.parquet
    ├── ts-05.parquet
    ├── ts-06.parquet
    ├── ts-07.parquet
    ├── ts-08.parquet
    ├── ts-09.parquet
    ├── ts-10.parquet
    └── ts-11.parquet
```

Now we'll implement an out-of-core `value_counts`. The peak memory usage of this workflow is the single largest chunk, plus a small series storing the unique value counts up to this point. As long as each individual file fits in memory, this will work for arbitrary-sized datasets.

```
In [19]: %%time
   ....: files = pathlib.Path("data/timeseries/").glob("ts*.parquet")
   ....: counts = pd.Series(dtype=int)
   ....: for path in files:
   ....:     df = pd.read_parquet(path)
   ....:     counts = counts.add(df['name'].value_counts(), fill_value=0)
   ....: counts.astype(int)
   ....:
CPU times: user 2.62 s, sys: 188 ms, total: 2.81 s
Wall time: 2.46 s
Out[19]:
Alice      229802
Bob        229211
```

(continues on next page)

(continued from previous page)

```
Charlie    229303
Dan        230621
Edith      230349
             ...
Victor     230502
Wendy      230038
Xavier     229553
Yvonne     228766
Zelda      229909
Length: 26, dtype: int64
```

Some readers, like `pandas.read_csv()`, offer parameters to control the `chunksize` when reading a single file.

Manually chunking is an OK option for workflows that don't require too sophisticated of operations. Some operations, like `groupby`, are much harder to do chunkwise. In these cases, you may be better switching to a different library that implements these out-of-core algorithms for you.

2.19.4 Use other libraries

Pandas is just one library offering a DataFrame API. Because of its popularity, pandas' API has become something of a standard that other libraries implement. The pandas documentation maintains a list of libraries implementing a DataFrame API in our ecosystem page.

For example, Dask, a parallel computing library, has dask.dataframe, a pandas-like API for working with larger than memory datasets in parallel. Dask can use multiple threads or processes on a single machine, or a cluster of machines to process data in parallel.

We'll import `dask.dataframe` and notice that the API feels similar to pandas. We can use Dask's `read_parquet` function, but provide a globstring of files to read in.

```
In [20]: import dask.dataframe as dd

In [21]: ddf = dd.read_parquet("data/timeseries/ts*.parquet", engine="pyarrow")

In [22]: ddf
Out[22]:
Dask DataFrame Structure:
                   id    name        x        y
npartitions=12
                int64  object  float64  float64
                  ...     ...      ...      ...
...               ...     ...      ...      ...
                  ...     ...      ...      ...
                  ...     ...      ...      ...
Dask Name: read-parquet, 12 tasks
```

Inspecting the `ddf` object, we see a few things

- There are familiar attributes like `.columns` and `.dtypes`
- There are familiar methods like `.groupby`, `.sum`, etc.
- There are new attributes like `.npartitions` and `.divisions`

The partitions and divisions are how Dask parallizes computation. A **Dask** DataFrame is made up of many **Pandas** DataFrames. A single method call on a Dask DataFrame ends up making many pandas method calls, and Dask knows how to coordinate everything to get the result.

```
In [23]: ddf.columns
Out[23]: Index(['id', 'name', 'x', 'y'], dtype='object')

In [24]: ddf.dtypes
Out[24]:
id          int64
name       object
x         float64
y         float64
dtype: object

In [25]: ddf.npartitions
Out[25]: 12
```

One major difference: the `dask.dataframe` API is *lazy*. If you look at the repr above, you'll notice that the values aren't actually printed out; just the column names and dtypes. That's because Dask hasn't actually read the data yet. Rather than executing immediately, doing operations build up a **task graph**.

```
In [26]: ddf
Out[26]:
Dask DataFrame Structure:
                   id    name        x        y
npartitions=12
                int64  object  float64  float64
                  ...     ...      ...      ...
...               ...     ...      ...      ...
                  ...     ...      ...      ...
                  ...     ...      ...      ...
Dask Name: read-parquet, 12 tasks

In [27]: ddf['name']
Out[27]:
Dask Series Structure:
npartitions=12
    object
       ...
     ...
       ...
       ...
Name: name, dtype: object
Dask Name: getitem, 24 tasks

In [28]: ddf['name'].value_counts()
Out[28]:
Dask Series Structure:
npartitions=1
    int64
      ...
Name: name, dtype: int64
Dask Name: value-counts-agg, 39 tasks
```

Each of these calls is instant because the result isn't being computed yet. We're just building up a list of computation to do when someone needs the result. Dask knows that the return type of a `pandas.Series.value_counts` is a pandas Series with a certain dtype and a certain name. So the Dask version returns a Dask Series with the same dtype and the same name.

To get the actual result you can call `.compute()`.

```
In [29]: %time ddf['name'].value_counts().compute()
CPU times: user 2.34 s, sys: 147 ms, total: 2.49 s
Wall time: 2.22 s
Out[29]:
Laura      230906
Ingrid     230838
Kevin      230698
Dan        230621
Frank      230595
            ...
Ray        229603
Xavier     229553
Charlie    229303
Bob        229211
Yvonne     228766
Name: name, Length: 26, dtype: int64
```

At that point, you get back the same thing you'd get with pandas, in this case a concrete pandas Series with the count of each `name`.

Calling `.compute` causes the full task graph to be executed. This includes reading the data, selecting the columns, and doing the `value_counts`. The execution is done *in parallel* where possible, and Dask tries to keep the overall memory footprint small. You can work with datasets that are much larger than memory, as long as each partition (a regular pandas DataFrame) fits in memory.

By default, `dask.dataframe` operations use a threadpool to do operations in parallel. We can also connect to a cluster to distribute the work on many machines. In this case we'll connect to a local "cluster" made up of several processes on this single machine.

```
>>> from dask.distributed import Client, LocalCluster

>>> cluster = LocalCluster()
>>> client = Client(cluster)
>>> client
<Client: 'tcp://127.0.0.1:53349' processes=4 threads=8, memory=17.18 GB>
```

Once this `client` is created, all of Dask's computation will take place on the cluster (which is just processes in this case).

Dask implements the most used parts of the pandas API. For example, we can do a familiar groupby aggregation.

```
In [30]: %time ddf.groupby('name')[['x', 'y']].mean().compute().head()
CPU times: user 2.67 s, sys: 510 ms, total: 3.18 s
Wall time: 3.05 s
Out[30]:
                x         y
name
Alice    0.000086 -0.001170
Bob     -0.000843 -0.000799
Charlie  0.000564 -0.000038
Dan      0.000584  0.000818
Edith   -0.000116 -0.000044
```

The grouping and aggregation is done out-of-core and in parallel.

When Dask knows the `divisions` of a dataset, certain optimizations are possible. When reading parquet datasets written by dask, the divisions will be known automatically. In this case, since we created the parquet files manually, we need to supply the divisions manually.

```
In [31]: N = 12

In [32]: starts = [f'20{i:>02d}-01-01' for i in range(N)]

In [33]: ends = [f'20{i:>02d}-12-13' for i in range(N)]

In [34]: divisions = tuple(pd.to_datetime(starts)) + (pd.Timestamp(ends[-1]),)

In [35]: ddf.divisions = divisions

In [36]: ddf
Out[36]:
Dask DataFrame Structure:
                   id    name        x        y
npartitions=12
2000-01-01      int64  object  float64  float64
2001-01-01        ...     ...      ...      ...
...               ...     ...      ...      ...
2011-01-01        ...     ...      ...      ...
2011-12-13        ...     ...      ...      ...
Dask Name: read-parquet, 12 tasks
```

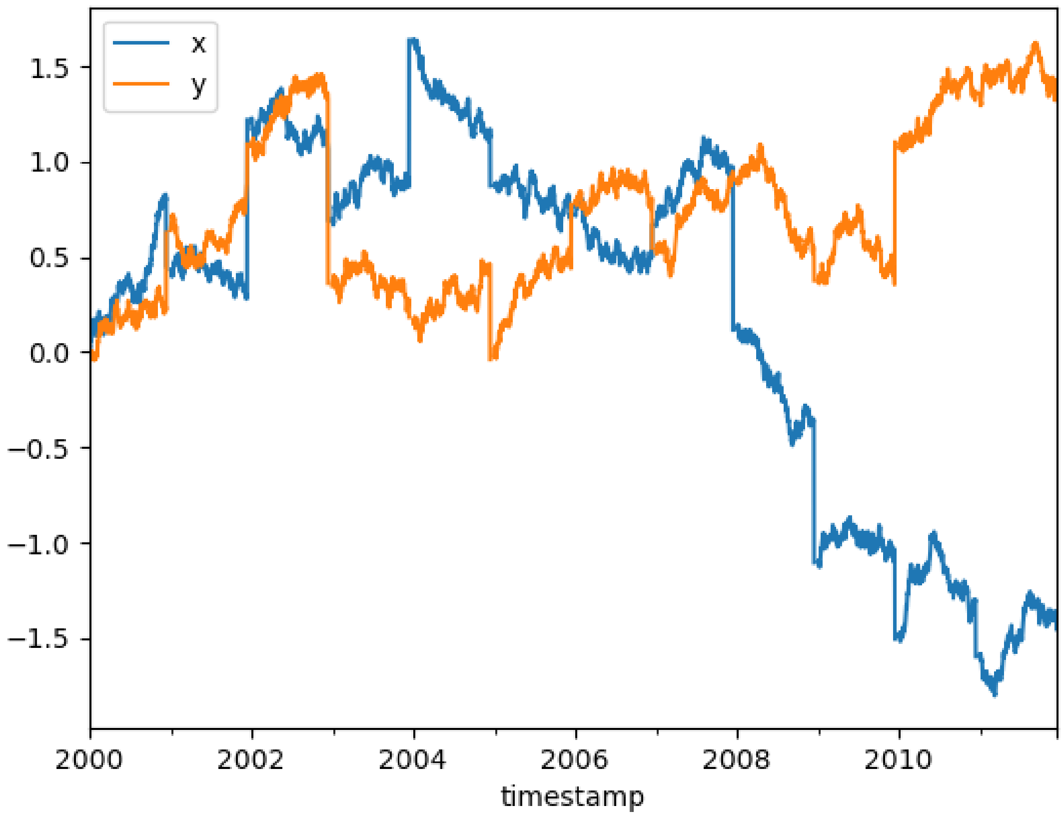
Now we can do things like fast random access with `.loc`.

```
In [37]: ddf.loc['2002-01-01 12:01':'2002-01-01 12:05'].compute()
Out[37]:
                       id    name         x         y
timestamp
2002-01-01 12:01:00   983   Laura  0.243985 -0.079392
2002-01-01 12:02:00  1001   Laura -0.523119 -0.226026
2002-01-01 12:03:00  1059  Oliver  0.612886  0.405680
2002-01-01 12:04:00   993   Kevin  0.451977  0.332947
2002-01-01 12:05:00  1014  Yvonne -0.948681  0.361748
```

Dask knows to just look in the 3rd partition for selecting values in *2002*. It doesn't need to look at any other data.

Many workflows involve a large amount of data and processing it in a way that reduces the size to something that fits in memory. In this case, we'll resample to daily frequency and take the mean. Once we've taken the mean, we know the results will fit in memory, so we can safely call `compute` without running out of memory. At that point it's just a regular pandas object.

```
In [38]: ddf[['x', 'y']].resample("1D").mean().cumsum().compute().plot()
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x7f53247d8890>
```



These Dask examples have all be done using multiple processes on a single machine. Dask can be deployed on a cluster to scale up to even larger datasets.

You see more dask examples at https://examples.dask.org.

2.20 Sparse data structures

Pandas provides data structures for efficiently storing sparse data. These are not necessarily sparse in the typical "mostly 0". Rather, you can view these objects as being "compressed" where any data matching a specific value (`NaN` / missing value, though any value can be chosen, including 0) is omitted. The compressed values are not actually stored in the array.

```
In [1]: arr = np.random.randn(10)

In [2]: arr[2:-2] = np.nan

In [3]: ts = pd.Series(pd.arrays.SparseArray(arr))

In [4]: ts
Out[4]:
0    0.469112
1   -0.282863
2         NaN
```

(continues on next page)

(continued from previous page)

```
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
dtype: Sparse[float64, nan]
```

Notice the dtype, `Sparse[float64, nan]`. The `nan` means that elements in the array that are `nan` aren't actually stored, only the non-`nan` elements are. Those non-`nan` elements have a `float64` dtype.

The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA `DataFrame`:

```
In [5]: df = pd.DataFrame(np.random.randn(10000, 4))

In [6]: df.iloc[:9998] = np.nan

In [7]: sdf = df.astype(pd.SparseDtype("float", np.nan))

In [8]: sdf.head()
Out[8]:
    0   1   2   3
0 NaN NaN NaN NaN
1 NaN NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN NaN NaN NaN
4 NaN NaN NaN NaN

In [9]: sdf.dtypes
Out[9]:
0    Sparse[float64, nan]
1    Sparse[float64, nan]
2    Sparse[float64, nan]
3    Sparse[float64, nan]
dtype: object

In [10]: sdf.sparse.density
Out[10]: 0.0002
```

As you can see, the density (% of values that have not been "compressed") is extremely low. This sparse object takes up much less memory on disk (pickled) and in the Python interpreter.

```
In [11]: 'dense : {:0.2f} bytes'.format(df.memory_usage().sum() / 1e3)
Out[11]: 'dense : 320.13 bytes'

In [12]: 'sparse: {:0.2f} bytes'.format(sdf.memory_usage().sum() / 1e3)
Out[12]: 'sparse: 0.22 bytes'
```

Functionally, their behavior should be nearly identical to their dense counterparts.

2.20.1 SparseArray

arrays.SparseArray is a *ExtensionArray* for storing an array of sparse values (see *dtypes* for more on extension arrays). It is a 1-dimensional ndarray-like object storing only values distinct from the fill_value:

```
In [13]: arr = np.random.randn(10)

In [14]: arr[2:5] = np.nan

In [15]: arr[7:8] = np.nan

In [16]: sparr = pd.arrays.SparseArray(arr)

In [17]: sparr
Out[17]:
[-1.9556635297215477, -1.6588664275960427, nan, nan, nan, 1.1589328886422277, 0.
↪14529711373305043, nan, 0.6060271905134522, 1.3342113401317768]
Fill: nan
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)
```

A sparse array can be converted to a regular (dense) ndarray with numpy.asarray()

```
In [18]: np.asarray(sparr)
Out[18]:
array([-1.9557, -1.6589,     nan,     nan,     nan,  1.1589,  0.1453,
          nan,  0.606 ,  1.3342])
```

2.20.2 SparseDtype

The SparseArray.dtype property stores two pieces of information

1. The dtype of the non-sparse values
2. The scalar fill value

```
In [19]: sparr.dtype
Out[19]: Sparse[float64, nan]
```

A *SparseDtype* may be constructed by passing each of these

```
In [20]: pd.SparseDtype(np.dtype('datetime64[ns]'))
Out[20]: Sparse[datetime64[ns], NaT]
```

The default fill value for a given NumPy dtype is the "missing" value for that dtype, though it may be overridden.

```
In [21]: pd.SparseDtype(np.dtype('datetime64[ns]'),
   ....:                fill_value=pd.Timestamp('2017-01-01'))
   ....:
Out[21]: Sparse[datetime64[ns], 2017-01-01 00:00:00]
```

Finally, the string alias 'Sparse[dtype]' may be used to specify a sparse dtype in many places

```
In [22]: pd.array([1, 0, 0, 2], dtype='Sparse[int]')
Out[22]:
[1, 0, 0, 2]
```

(continues on next page)

(continued from previous page)

```
Fill: 0
IntIndex
Indices: array([0, 3], dtype=int32)
```

2.20.3 Sparse accessor

New in version 0.24.0.

Pandas provides a `.sparse` accessor, similar to `.str` for string data, `.cat` for categorical data, and `.dt` for datetime-like data. This namespace provides attributes and methods that are specific to sparse data.

```
In [23]: s = pd.Series([0, 0, 1, 2], dtype="Sparse[int]")

In [24]: s.sparse.density
Out[24]: 0.5

In [25]: s.sparse.fill_value
Out[25]: 0
```

This accessor is available only on data with `SparseDtype`, and on the *Series* class itself for creating a Series with sparse data from a scipy COO matrix with.

New in version 0.25.0.

A `.sparse` accessor has been added for *DataFrame* as well. See *Sparse accessor* for more.

2.20.4 Sparse calculation

You can apply NumPy ufuncs to `SparseArray` and get a `SparseArray` as a result.

```
In [26]: arr = pd.arrays.SparseArray([1., np.nan, np.nan, -2., np.nan])

In [27]: np.abs(arr)
Out[27]: 
[1.0, nan, nan, 2.0, nan]
Fill: nan
IntIndex
Indices: array([0, 3], dtype=int32)
```

The *ufunc* is also applied to `fill_value`. This is needed to get the correct dense result.

```
In [28]: arr = pd.arrays.SparseArray([1., -1, -1, -2., -1], fill_value=-1)

In [29]: np.abs(arr)
Out[29]: 
[1.0, 1, 1, 2.0, 1]
Fill: 1
IntIndex
Indices: array([0, 3], dtype=int32)

In [30]: np.abs(arr).to_dense()
Out[30]: array([1., 1., 1., 2., 1.])
```

2.20.5 Migrating

Note: SparseSeries and SparseDataFrame were removed in pandas 1.0.0. This migration guide is present to aid in migrating from previous versions.

In older versions of pandas, the SparseSeries and SparseDataFrame classes (documented below) were the preferred way to work with sparse data. With the advent of extension arrays, these subclasses are no longer needed. Their purpose is better served by using a regular Series or DataFrame with sparse values instead.

Note: There's no performance or memory penalty to using a Series or DataFrame with sparse values, rather than a SparseSeries or SparseDataFrame.

This section provides some guidance on migrating your code to the new style. As a reminder, you can use the python warnings module to control warnings. But we recommend modifying your code, rather than ignoring the warning.

Construction

From an array-like, use the regular *Series* or *DataFrame* constructors with SparseArray values.

```
# Previous way
>>> pd.SparseDataFrame({"A": [0, 1]})
```

```
# New way
In [31]: pd.DataFrame({"A": pd.arrays.SparseArray([0, 1])})
Out[31]:
   A
0  0
1  1
```

From a SciPy sparse matrix, use *DataFrame.sparse.from_spmatrix()*,

```
# Previous way
>>> from scipy import sparse
>>> mat = sparse.eye(3)
>>> df = pd.SparseDataFrame(mat, columns=['A', 'B', 'C'])
```

```
# New way
In [32]: from scipy import sparse

In [33]: mat = sparse.eye(3)

In [34]: df = pd.DataFrame.sparse.from_spmatrix(mat, columns=['A', 'B', 'C'])

In [35]: df.dtypes
Out[35]:
A    Sparse[float64, 0.0]
B    Sparse[float64, 0.0]
C    Sparse[float64, 0.0]
dtype: object
```

Conversion

From sparse to dense, use the .sparse accessors

```
In [36]: df.sparse.to_dense()
Out[36]:
     A    B    C
0  1.0  0.0  0.0
1  0.0  1.0  0.0
2  0.0  0.0  1.0

In [37]: df.sparse.to_coo()
Out[37]:
<3x3 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in COOrdinate format>
```

From dense to sparse, use `DataFrame.astype()` with a `SparseDtype`.

```
In [38]: dense = pd.DataFrame({"A": [1, 0, 0, 1]})

In [39]: dtype = pd.SparseDtype(int, fill_value=0)

In [40]: dense.astype(dtype)
Out[40]:
   A
0  1
1  0
2  0
3  1
```

Sparse Properties

Sparse-specific properties, like `density`, are available on the `.sparse` accessor.

```
In [41]: df.sparse.density
Out[41]: 0.3333333333333333
```

General differences

In a `SparseDataFrame`, *all* columns were sparse. A `DataFrame` can have a mixture of sparse and dense columns. As a consequence, assigning new columns to a `DataFrame` with sparse values will not automatically convert the input to be sparse.

```
# Previous Way
>>> df = pd.SparseDataFrame({"A": [0, 1]})
>>> df['B'] = [0, 0]  # implicitly becomes Sparse
>>> df['B'].dtype
Sparse[int64, nan]
```

Instead, you'll need to ensure that the values being assigned are sparse

```
In [42]: df = pd.DataFrame({"A": pd.arrays.SparseArray([0, 1])})

In [43]: df['B'] = [0, 0]  # remains dense

In [44]: df['B'].dtype
Out[44]: dtype('int64')

In [45]: df['B'] = pd.arrays.SparseArray([0, 0])

In [46]: df['B'].dtype
Out[46]: Sparse[int64, 0]
```

The `SparseDataFrame.default_kind` and `SparseDataFrame.default_fill_value` attributes have no replacement.

2.20.6 Interaction with scipy.sparse

Use *DataFrame.sparse.from_spmatrix()* to create a `DataFrame` with sparse values from a sparse matrix.

New in version 0.25.0.

```
In [47]: from scipy.sparse import csr_matrix

In [48]: arr = np.random.random(size=(1000, 5))

In [49]: arr[arr < .9] = 0

In [50]: sp_arr = csr_matrix(arr)

In [51]: sp_arr
Out[51]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 517 stored elements in Compressed Sparse Row format>

In [52]: sdf = pd.DataFrame.sparse.from_spmatrix(sp_arr)

In [53]: sdf.head()
Out[53]:
          0    1    2         3    4
0  0.956380  0.0  0.0  0.000000  0.0
1  0.000000  0.0  0.0  0.000000  0.0
2  0.000000  0.0  0.0  0.000000  0.0
3  0.000000  0.0  0.0  0.000000  0.0
4  0.999552  0.0  0.0  0.956153  0.0

In [54]: sdf.dtypes
Out[54]:
0    Sparse[float64, 0.0]
1    Sparse[float64, 0.0]
2    Sparse[float64, 0.0]
3    Sparse[float64, 0.0]
4    Sparse[float64, 0.0]
dtype: object
```

All sparse formats are supported, but matrices that are not in `COOrdinate` format will be converted, copying data as needed. To convert back to sparse SciPy matrix in COO format, you can use the *DataFrame.sparse.to_coo()* method:

```
In [55]: sdf.sparse.to_coo()
Out[55]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
        with 517 stored elements in COOrdinate format>
```

meth:*Series.sparse.to_coo* is implemented for transforming a `Series` with sparse values indexed by a *MultiIndex* to a `scipy.sparse.coo_matrix`.

The method requires a `MultiIndex` with two or more levels.

```
In [56]: s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])

In [57]: s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
   ....:                                      (1, 2, 'a', 1),
   ....:                                      (1, 1, 'b', 0),
   ....:                                      (1, 1, 'b', 1),
   ....:                                      (2, 1, 'b', 0),
   ....:                                      (2, 1, 'b', 1)],
   ....:                                     names=['A', 'B', 'C', 'D'])
   ....:

In [58]: s
Out[58]:
A  B  C  D
1  2  a  0    3.0
         1    NaN
   1  b  0    1.0
         1    3.0
2  1  b  0    NaN
         1    NaN
dtype: float64

In [59]: ss = s.astype('Sparse')

In [60]: ss
Out[60]:
A  B  C  D
1  2  a  0    3.0
         1    NaN
   1  b  0    1.0
         1    3.0
2  1  b  0    NaN
         1    NaN
dtype: Sparse[float64, nan]
```

In the example below, we transform the `Series` to a sparse representation of a 2-d array by specifying that the first and second `MultiIndex` levels define labels for the rows and the third and fourth levels define labels for the columns. We also specify that the column and row labels should be sorted in the final sparse representation.

```
In [61]: A, rows, columns = ss.sparse.to_coo(row_levels=['A', 'B'],
   ....:                                     column_levels=['C', 'D'],
   ....:                                     sort_labels=True)
   ....:

In [62]: A
Out[62]:
<3x4 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in COOrdinate format>

In [63]: A.todense()
Out[63]:
matrix([[0., 0., 1., 3.],
        [3., 0., 0., 0.],
        [0., 0., 0., 0.]])

In [64]: rows
Out[64]: [(1, 1), (1, 2), (2, 1)]
```

(continues on next page)

(continued from previous page)

```
In [65]: columns
Out[65]: [('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

Specifying different row and column labels (and not sorting them) yields a different sparse matrix:

```
In [66]: A, rows, columns = ss.sparse.to_coo(row_levels=['A', 'B', 'C'],
   ....:                                     column_levels=['D'],
   ....:                                     sort_labels=False)
   ....:

In [67]: A
Out[67]: 
<3x2 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in COOrdinate format>

In [68]: A.todense()
Out[68]: 
matrix([[3., 0.],
        [1., 3.],
        [0., 0.]])

In [69]: rows
Out[69]: [(1, 2, 'a'), (1, 1, 'b'), (2, 1, 'b')]

In [70]: columns
Out[70]: [0, 1]
```

A convenience method *Series.sparse.from_coo()* is implemented for creating a `Series` with sparse values from a `scipy.sparse.coo_matrix`.

```
In [71]: from scipy import sparse

In [72]: A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
   ....:                       shape=(3, 4))
   ....:

In [73]: A
Out[73]: 
<3x4 sparse matrix of type '<class 'numpy.float64'>'
	with 3 stored elements in COOrdinate format>

In [74]: A.todense()
Out[74]: 
matrix([[0., 0., 1., 2.],
        [3., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

The default behaviour (with `dense_index=False`) simply returns a `Series` containing only the non-null entries.

```
In [75]: ss = pd.Series.sparse.from_coo(A)

In [76]: ss
Out[76]: 
0  2    1.0
   3    2.0
```

(continues on next page)

(continued from previous page)

```
1  0    3.0
dtype: Sparse[float64, nan]
```

Specifying `dense_index=True` will result in an index that is the Cartesian product of the row and columns coordinates of the matrix. Note that this will consume a significant amount of memory (relative to `dense_index=False`) if the sparse matrix is large (and sparse) enough.

```
In [77]: ss_dense = pd.Series.sparse.from_coo(A, dense_index=True)

In [78]: ss_dense
Out[78]: 
0  0    NaN
   1    NaN
   2    1.0
   3    2.0
1  0    3.0
   1    NaN
   2    NaN
   3    NaN
2  0    NaN
   1    NaN
   2    NaN
   3    NaN
dtype: Sparse[float64, nan]
```

2.21 Frequently Asked Questions (FAQ)

2.21.1 DataFrame memory usage

The memory usage of a `DataFrame` (including the index) is shown when calling the *info()*. A configuration option, `display.memory_usage` (see *the list of options*), specifies if the `DataFrame`'s memory usage will be displayed when invoking the `df.info()` method.

For example, the memory usage of the `DataFrame` below is shown when calling *info()*:

```
In [1]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
   ...:           'complex128', 'object', 'bool']
   ...: 

In [2]: n = 5000

In [3]: data = {t: np.random.randint(100, size=n).astype(t) for t in dtypes}

In [4]: df = pd.DataFrame(data)

In [5]: df['categorical'] = df['object'].astype('category')

In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
```

(continues on next page)

(continued from previous page)

```
 0   int64            5000 non-null   int64
 1   float64          5000 non-null   float64
 2   datetime64[ns]   5000 non-null   datetime64[ns]
 3   timedelta64[ns]  5000 non-null   timedelta64[ns]
 4   complex128       5000 non-null   complex128
 5   object           5000 non-null   object
 6   bool             5000 non-null   bool
 7   categorical      5000 non-null   category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↪object(1), timedelta64[ns](1)
memory usage: 289.1+ KB
```

The + symbol indicates that the true memory usage could be higher, because pandas does not count the memory used by values in columns with `dtype=object`.

Passing `memory_usage='deep'` will enable a more accurate memory usage report, accounting for the full usage of the contained objects. This is optional as it can be expensive to do this deeper introspection.

```
In [7]: df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   int64            5000 non-null   int64
 1   float64          5000 non-null   float64
 2   datetime64[ns]   5000 non-null   datetime64[ns]
 3   timedelta64[ns]  5000 non-null   timedelta64[ns]
 4   complex128       5000 non-null   complex128
 5   object           5000 non-null   object
 6   bool             5000 non-null   bool
 7   categorical      5000 non-null   category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↪object(1), timedelta64[ns](1)
memory usage: 425.6 KB
```

By default the display option is set to `True` but can be explicitly overridden by passing the `memory_usage` argument when invoking `df.info()`.

The memory usage of each column can be found by calling the *`memory_usage()`* method. This returns a `Series` with an index represented by column names and memory usage of each column shown in bytes. For the `DataFrame` above, the memory usage of each column and the total memory usage can be found with the `memory_usage` method:

```
In [8]: df.memory_usage()
Out[8]:
Index                128
int64              40000
float64            40000
datetime64[ns]     40000
timedelta64[ns]    40000
complex128         80000
object             40000
bool                5000
categorical        10920
dtype: int64

# total memory usage of dataframe
```

(continues on next page)

(continued from previous page)

```
In [9]: df.memory_usage().sum()
Out[9]: 296048
```

By default the memory usage of the DataFrame's index is shown in the returned Series, the memory usage of the index can be suppressed by passing the index=False argument:

```
In [10]: df.memory_usage(index=False)
Out[10]:
int64              40000
float64            40000
datetime64[ns]     40000
timedelta64[ns]    40000
complex128         80000
object             40000
bool                5000
categorical        10920
dtype: int64
```

The memory usage displayed by the *info()* method utilizes the *memory_usage()* method to determine the memory usage of a DataFrame while also formatting the output in human-readable units (base-2 representation; i.e. 1KB = 1024 bytes).

See also *Categorical Memory Usage*.

2.21.2 Using if/truth statements with pandas

pandas follows the NumPy convention of raising an error when you try to convert something to a bool. This happens in an if-statement or when using the boolean operations: and, or, and not. It is not clear what the result of the following code should be:

```
>>> if pd.Series([False, True, False]):
...     pass
```

Should it be True because it's not zero-length, or False because there are False values? It is unclear, so instead, pandas raises a ValueError:

```
>>> if pd.Series([False, True, False]):
...     print("I was true")
Traceback
    ...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

You need to explicitly choose what you want to do with the DataFrame, e.g. use *any()*, *all()* or *empty()*. Alternatively, you might want to compare if the pandas object is None:

```
>>> if pd.Series([False, True, False]) is not None:
...     print("I was not None")
I was not None
```

Below is how to check if any of the values are True:

```
>>> if pd.Series([False, True, False]).any():
...     print("I am any")
I am any
```

To evaluate single-element pandas objects in a boolean context, use the method *bool()*:

```
In [11]: pd.Series([True]).bool()
Out[11]: True

In [12]: pd.Series([False]).bool()
Out[12]: False

In [13]: pd.DataFrame([[True]]).bool()
Out[13]: True

In [14]: pd.DataFrame([[False]]).bool()
Out[14]: False
```

Bitwise boolean

Bitwise boolean operators like == and != return a boolean `Series`, which is almost always what you want anyways.

```
>>> s = pd.Series(range(5))
>>> s == 4
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

See *boolean comparisons* for more examples.

Using the `in` operator

Using the Python `in` operator on a `Series` tests for membership in the index, not membership among the values.

```
In [15]: s = pd.Series(range(5), index=list('abcde'))

In [16]: 2 in s
Out[16]: False

In [17]: 'b' in s
Out[17]: True
```

If this behavior is surprising, keep in mind that using `in` on a Python dictionary tests keys, not values, and `Series` are dict-like. To test for membership in the values, use the method `isin()`:

```
In [18]: s.isin([2])
Out[18]:
a    False
b    False
c     True
d    False
e    False
dtype: bool

In [19]: s.isin([2]).any()
Out[19]: True
```

For `DataFrames`, likewise, `in` applies to the column axis, testing for membership in the list of column names.

2.21.3 NaN, Integer NA values and NA type promotions

Choice of NA representation

For lack of NA (missing) support from the ground up in NumPy and Python in general, we were given the difficult choice between either:

- A *masked array* solution: an array of data and an array of boolean values indicating whether a value is there or is missing.
- Using a special sentinel value, bit pattern, or set of sentinel values to denote NA across the dtypes.

For many reasons we chose the latter. After years of production use it has proven, at least in my opinion, to be the best decision given the state of affairs in NumPy and Python in general. The special value NaN (Not-A-Number) is used everywhere as the NA value, and there are API functions isna and notna which can be used across the dtypes to detect NA values.

However, it comes with it a couple of trade-offs which I most certainly have not ignored.

Support for integer NA

In the absence of high performance NA support being built into NumPy from the ground up, the primary casualty is the ability to represent NAs in integer arrays. For example:

```
In [20]: s = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))

In [21]: s
Out[21]:
a    1
b    2
c    3
d    4
e    5
dtype: int64

In [22]: s.dtype
Out[22]: dtype('int64')

In [23]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])

In [24]: s2
Out[24]:
a    1.0
b    2.0
c    3.0
f    NaN
u    NaN
dtype: float64

In [25]: s2.dtype
Out[25]: dtype('float64')
```

This trade-off is made largely for memory and performance reasons, and also so that the resulting Series continues to be "numeric".

If you need to represent integers with possibly missing values, use one of the nullable-integer extension dtypes provided by pandas

- *Int8Dtype*
- *Int16Dtype*
- *Int32Dtype*
- *Int64Dtype*

```
In [26]: s_int = pd.Series([1, 2, 3, 4, 5], index=list('abcde'),
   ....:                   dtype=pd.Int64Dtype())
   ....:

In [27]: s_int
Out[27]:
a    1
b    2
c    3
d    4
e    5
dtype: Int64

In [28]: s_int.dtype
Out[28]: Int64Dtype()

In [29]: s2_int = s_int.reindex(['a', 'b', 'c', 'f', 'u'])

In [30]: s2_int
Out[30]:
a       1
b       2
c       3
f    <NA>
u    <NA>
dtype: Int64

In [31]: s2_int.dtype
Out[31]: Int64Dtype()
```

See *Nullable integer data type* for more.

`NA` type promotions

When introducing NAs into an existing `Series` or `DataFrame` via *`reindex()`* or some other means, boolean and integer types will be promoted to a different dtype in order to store the NAs. The promotions are summarized in this table:

Typeclass	Promotion dtype for storing NAs
`floating`	no change
`object`	no change
`integer`	cast to `float64`
`boolean`	cast to `object`

While this may seem like a heavy trade-off, I have found very few cases where this is an issue in practice i.e. storing values greater than 2**53. Some explanation for the motivation is in the next section.