vaex Documentation

Release 2.4.0

Maarten A. Breddels

Dec 02, 2019

Contents

Warning: It is recommended not to install directly into your operating system's Python using sudo since it may break your system. Instead, you should install Anaconda, which is a Python distribution that makes installing Python packages much easier or use virtualenv or venv.

CHAPTER 1

Short version

- **Anaconda users**: `conda install -c conda-forge vaex`
- **Regular Python users using virtualenv**: `pip install vaex`
- **Regular Python users (not recommended)**: `pip install --user vaex`
- **System install (not recommended)**: `sudo pip install vaex`

CHAPTER 2

Longer version

If you don't want all packages installed, do not install the vaex package. The vaex package is a meta packages that depends on all other vaex packages so it will instal them all, but if you don't need astronomy related parts (`vaex-astro`), or don't care about distributed (`vaex-distributed`), you can leave out those packages. Copy paste the following lines and remove what you do not need:

- **Regular Python users**: `pip install vaex-core vaex-viz vaex-jupyter vaex-arrow vaex-server vaex-ui vaex-hdf5 vaex-astro vaex-distributed`
- **Anaconda users**: `conda install -c conda-forge vaex-core vaex-viz vaex-jupyter vaex-arrow vaex-server vaex-ui vaex-hdf5 vaex-astro vaex-distributed`

When installing `vaex-ui` it does not install PyQt4, PyQt5 or PySide, you have to choose yourself and installing may be tricky. If running pip install PyQt5 fails, you may want to try your favourite package manager (brew, macports) to install it instead. You can check if you have one of these packages by running:

- `python -c "import PyQt4"`
- `python -c "import PyQt5"`
- `python -c "import PySide"`

CHAPTER 3

For developers

If you want to work on vaex for a Pull Request from the source, use the following recipe:

- `git clone --recursive https://github.com/vaexio/vaex` # make sure you get the submodules
- `cd vaex`
- make sure the dev versions of pcre are installed (e.g. `conda install -c conda-forge pcre`)
- install using:
- `pip install -e .` (again, use (ana)conda or virtualenv/venv)
- If you want to do a PR
- `git remote rename origin upstream`
- (now fork on github)
- `git remote add origin https://github.com/yourusername/vaex/`
- … edit code … (or do this after the next step)
- `git checkout -b feature_X`
- `git commit -a -m "new: some feature X"`
- `git push origin feature_X`
- `git checkout master`
- Get your code in sync with upstream
- `git checkout master`
- `git fetch upstream`
- `git merge upstream/master`

CHAPTER 4

Vaex tutorial

4.1 DataFrame

Central to vaex is the DataFrame (similar, but more efficient than a pandas dataframe), and we often use the variables `df` to represent it. A DataFrame is an efficient representation for large tabular data, and has:

- A bunch of columns, say `x`, `y` and `z`
- Backed by a numpy array, e.g. `df.data.x` (but you shouldn't work with this directly)
- Wrapped by an expression system, e.g. `df.x`, `df['x']` or `df.col.x` is an expression
- Columns/expression can perform lazy computations, e.g. `df.x * np.sin(df.y)` does nothing, until the result is needed
- A set of virtual columns, columns that are backed by a (lazy) computation, e.g. `df['r'] = df.x/df.y`
- A set of selection, that can be used to explore the dataset, e.g. `df.select(df.x < 0)`
- Filtered DataFrames, that does not copy the data, `df_negative = df[df.x < 0]`

Lets start with an example dataset, included in vaex

[1]:
```
import vaex
df = vaex.example()
df  # begin the last statement it will print out the tabular data
```

[1]:
```
#         x              y              z              vx             vy             vz          ␣
↪    E                  L                   Lz                  FeH
0         -0.777470767  2.10626292     1.93743467     53.276722      288.386047     -95.
↪2649078  -121238.171875   831.0799560546875   -336.426513671875    -2.
↪309227609164518
1         3.77427316     2.23387194     3.76209331     252.810791     -69.9498444   -56.
↪3121033  -100819.9140625  1435.1839599609375  -828.7567749023438   -1.
↪788735491591229
2         1.3757627      -6.3283844     2.63250017     96.276474      226.440201    -34.
↪7527161  -100559.9609375  1039.2989501953125  920.802490234375     -0.
↪7618109022478798
```

(continues on next page)

(continued from previous page)

```
3        -7.06737804   1.31737781    -6.10543537   204.968842   -205.679016  -58.
↪9777031  -70174.8515625   2441.724853515625   1183.5899658203125   -1.
↪5208778422936413
4         0.243441463   -0.822781682  -0.206593871  -311.742371  -238.41217   186.
↪824127   -144138.75       374.8164367675781   -314.5353088378906   -2.
↪655341358427361
...       ...           ...           ...           ...          ...          ...     ␣
↪    ...              ...               ...                   ...
329,995  3.76883793    4.66251659    -4.42904139   107.432999   -2.13771296  17.
↪5130272   -119687.3203125  746.8833618164062   -508.96484375        -1.
↪6499842518381402
329,996  9.17409325    -8.87091351   -8.61707687   32.0         108.089264   179.
↪060638   -68933.8046875   2395.633056640625   1275.490234375       -1.
↪4336036247720836
329,997  -1.14041007   -8.4957695    2.25749826    8.46711349   -38.2765236  -127.
↪541473  -112580.359375   1182.436279296875   115.58557891845703   -1.
↪9306227597361942
329,998  -14.2985935   -5.51750422   -8.65472317   110.221558   -31.3925591  86.
↪2726822   -74862.90625     1324.5926513671875  1057.017333984375    -1.
↪225019818838568
329,999  10.5450506    -8.86106777   -4.65835428   -2.10541415  -27.6108856  3.
↪80799961   -95361.765625    351.0955505371094   -309.81439208984375  -2.
↪5689636894079477
```

4.1.1 Columns

The above preview shows this dataset contains $> 300,000$ rows, and columns named x,y,z (positions), vx, vy, vz (velocities), E (energy), L (angular momentum). Printing out a column, shows it is not a numpy array, but an expression

[2]:
```
df.x  # df.col.x or df['x'] are equivalent, but may be preferred because it is more␣
↪tab completion friend or programmatics friendly respectively
```

[2]:
```
<vaex.expression.Expression(expressions='x')> instance at 0x117ac0438 values=[-0.
↪777470767, 3.77427316, 1.3757627, -7.06737804, 0.243441463 ... (total 330000␣
↪values) ... 3.76883793, 9.17409325, -1.14041007, -14.2985935, 10.5450506]
```

The underlying data is often accessible using `df.data.x`, but should not be used, since selections and filtering are not reflected in this. However sometimes it is useful to access the raw numpy array.

[3]:
```
df.data.x
```

[3]:
```
array([ -0.77747077,   3.77427316,   1.3757627 , ...,  -1.14041007,
       -14.2985935 ,  10.5450506 ])
```

A better way, if you need a numpy array (for instance for plotting, or passing to a different library) it to use evalulate, which will also work with virtual columns, selections and filtered DataFrames (more on that below).

[4]:
```
df.evaluate(df.x)
```

[4]:
```
array([ -0.77747077,   3.77427316,   1.3757627 , ...,  -1.14041007,
       -14.2985935 ,  10.5450506 ])
```

Most numpy function (ufuncs) can be performed on expressions, and will not result in a direct result, but in a new expression.

[5]:
```
import numpy as np
np.sqrt(df.x**2 + df.y**2 + df.z**2)
```

[5]:
```
<vaex.expression.Expression(expressions='sqrt((((x ** 2) + (y ** 2)) + (z ** 2)))')>␣
↪instance at 0x116244cf8 values=[2.9655450396553587, 5.77829281049018, 6.
↪99079603950256, 9.431842752707537, 0.8825613121347967 ... (total 330000 values) ...␣
↪7.453831761514681, 15.398412491068198, 8.864250273925633, 17.601047186042507, 14.
↪540181524970293]
```

4.1.2 Virtual functions

Sometimes it is convenient to store an expression as a column, or virtual column, a column that does not take up memory, but will be computed on the fly. A virtual column can be treated as a normal column.

[6]:
```
df['r'] = np.sqrt(df.x**2 + df.y**2 + df.z**2)
df[['x', 'y', 'z', 'r']]
```

[6]:
```
#        x            y            z            r
0        -0.777470767 2.10626292   1.93743467   2.9655450396553587
1        3.77427316   2.23387194   3.76209331   5.77829281049018
2        1.3757627    -6.3283844   2.63250017   6.99079603950256
3        -7.06737804  1.31737781   -6.10543537  9.431842752707537
4        0.243441463  -0.822781682 -0.206593871 0.8825613121347967
...      ...          ...          ...          ...
329,995  3.76883793   4.66251659   -4.42904139  7.453831761514681
329,996  9.17409325   -8.87091351  -8.61707687  15.398412491068198
329,997  -1.14041007  -8.4957695   2.25749826   8.864250273925633
329,998  -14.2985935  -5.51750422  -8.65472317  17.601047186042507
329,999  10.5450506   -8.86106777  -4.65835428  14.540181524970293
```

4.1.3 Selections and filtering

Vaex can be efficient when exploring subsets of the data, for instance to remove outlier or to inspect only a part of the data. Instead of making copies, internally vaex keeps track which rows is selected.

[7]:
```
df.select(df.x < 0)
df.evaluate(df.x, selection=True)
```

[7]:
```
array([ -0.77747077,  -7.06737804,  -5.17174435, ...,  -1.87310386,
        -1.14041007, -14.2985935 ])
```

Selections can be useful if you want to change what you select frequently, as in visualization, or when you want to compute statistics on several selections efficiently. Instead, you can also create a filtered dataset, and is similar in use to pandas, except that it does not copy the data.

[8]:
```
df_negative = df[df.x < 0]
df_negative[['x', 'y', 'z', 'r']]
```

[8]:
```
#        x            y            z            r
0        -0.777470767 2.10626292   1.93743467   2.9655450396553587
1        -7.06737804  1.31737781   -6.10543537  9.431842752707537
2        -5.17174435  7.82915306   1.82668829   9.559255586471544
3        -15.9538851  5.77125883   -9.02472305  19.21664654397474
4        -12.3994961  13.9181805   -5.43482304  19.416502090763164
...      ...          ...          ...          ...
```

(continues on next page)

(continued from previous page)

```
165,935  -9.88553238   -6.59253597   6.53742027    13.561826747838182
165,936  -2.38018084   4.73540306    0.141765863   5.301829922929686
165,937  -1.87310386   -0.503091216  -0.951977015  2.1605275001840565
165,938  -1.14041007   -8.4957695    2.25749826    8.864250273925633
165,939  -14.2985935   -5.51750422   -8.65472317   17.601047186042507
```

4.2 Statistics on N-d grids

A core feature of vaex, and used for visualization, is calculation of statistics on N dimensional gridf.

[9]:
```
df.count(), df.mean(df.x), df.mean(df.x, selection=True)
```

[9]:
```
(array(330000.), -0.06713149126400597, -5.211037972111967)
```

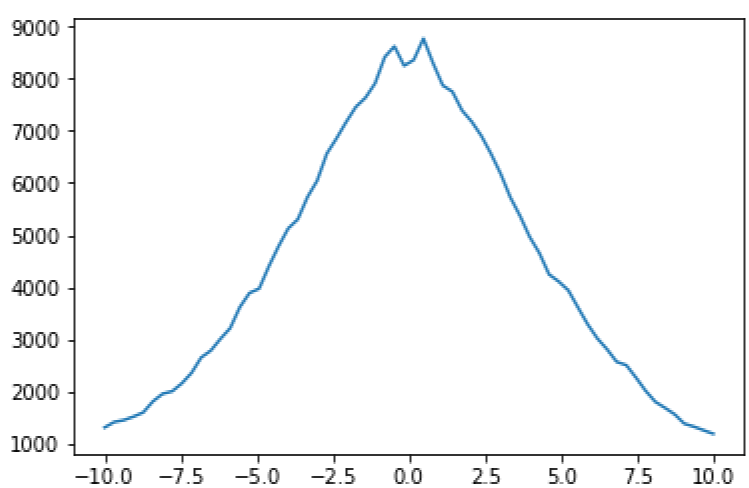
Similar to SQL's groupby, vaex uses the binby concept, which tells vaex that a statistic should be calculated on a regular grid (for performance reasons)

[10]:
```
xcounts = df.count(binby=df.x, limits=[-10, 10], shape=64)
xcounts
```

[10]:
```
array([1310., 1416., 1452., 1519., 1599., 1810., 1956., 2005., 2157.,
       2357., 2653., 2786., 3012., 3215., 3619., 3890., 3973., 4400.,
       4782., 5126., 5302., 5729., 6042., 6562., 6852., 7167., 7456.,
       7633., 7910., 8415., 8619., 8246., 8358., 8769., 8294., 7870.,
       7749., 7389., 7174., 6901., 6557., 6173., 5721., 5367., 4963.,
       4655., 4246., 4110., 3939., 3611., 3289., 3018., 2811., 2570.,
       2505., 2267., 2013., 1803., 1687., 1563., 1384., 1326., 1257.,
       1189.])
```

This results in a numpy array with the number counts in 64 bins distributed between x = -10, and x = 10. We can quickly visualize this using matplotlib.

[11]:
```
import matplotlib.pylab as plt
plt.plot(np.linspace(-10, 10, 64), xcounts)
plt.show()
```

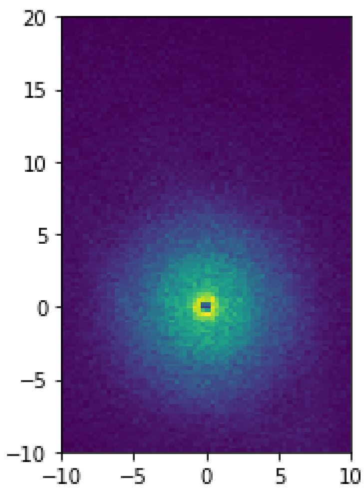


We can instead of doing 1d binning, do it in 2d as well (N-d actually), and visualize it using imshow.

```
[12]: xycounts = df.count(binby=[df.x, df.y], limits=[[-10, 10], [-10, 20]], shape=(64,␣
      ↪128))
      xycounts
```

```
[12]: array([[ 9.,  3.,  3., ...,  3.,  2.,  1.],
             [ 5.,  3.,  1., ...,  1.,  3.,  3.],
             [11.,  3.,  2., ...,  1.,  1.,  4.],
             ...,
             [12.,  6.,  8., ...,  0.,  1.,  0.],
             [ 7.,  6., 12., ...,  3.,  0.,  0.],
             [11., 10.,  7., ...,  1.,  1.,  1.]])
```

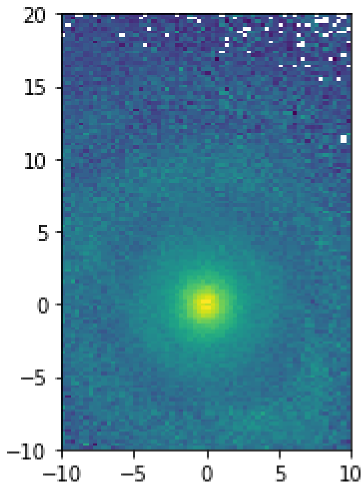
```
[13]: plt.imshow(xycounts.T, origin='lower', extent=[-10, 10, -10, 20])
      plt.show()
```




```
[14]: v = np.sqrt(df.vx**2 + df.vy**2 + df.vz**2)
      xy_mean_v = df.mean(v, binby=[df.x, df.y], limits=[[-10, 10], [-10, 20]], shape=(64,␣
      ↪128))
      xy_mean_v
```

```
[14]: array([[144.38495511, 183.45775869, 187.78325557, ..., 138.99392387,
              168.66141282, 142.55018784],
             [143.72427758, 152.14679337, 107.90949865, ..., 119.65318885,
               94.00098292, 104.35109636],
             [172.08240652, 137.47896886,  72.51331138, ..., 179.85933835,
               33.36968912, 111.81826254],
             ...,
             [186.56949934, 161.3747346 , 174.27411865, ...,          nan,
              105.96746091,          nan],
             [179.55997022, 137.48979882, 113.82121826, ..., 104.90205692,
                       nan,          nan],
             [151.94323763, 135.44083212,  84.81787495, ..., 175.79289144,
              129.63799565, 108.19069385]])
```

```
[15]: plt.imshow(xy_mean_v.T, origin='lower', extent=[-10, 10, -10, 20])
      plt.show()
```



Other statistics can be computed, such as:

- *DataFrame.count*
- *DataFrame.mean*
- *DataFrame.std*
- *DataFrame.var*
- *DataFrame.median_approx*
- *DataFrame.percentile_approx*
- *DataFrame.mode*
- *DataFrame.min*
- *DataFrame.max*
- *DataFrame.minmax*
- *DataFrame.mutual_information*
- *DataFrame.correlation*

Or see the full list at the *API docs*

4.3 Getting your data in

Before continuing, you may want to read in your own data. Ultimately, a vaex DataFrame just wraps a set of numpy arrays. If you can access your data as a set of numpy arrays, you can therefore make dataset using *from_arrays*

[17]:
```
import vaex
import numpy as np
x = np.arange(5)
y = x**2
df = vaex.from_arrays(x=x, y=y)
df
```

[17]:

#	x	y
0	0	0
1	1	1
2	2	4
3	3	9
4	4	16

Other quick ways to get your data in are:

- *from_arrow_table*: Arrow table support.
- *from_csv*: Comma separated files
- *from_ascii*: Space/tab separated files
- *from_pandas*: Converts a pandas DataFrame
- *from_astropy_table*: Converts a astropy table

Exporting, or converting a DataFrame to a different datastructure is also quite easy:

- *DataFrame.to_arrow_table*
- *DataFrame.to_dask_array*
- *DataFrame.to_pandas_df*
- *DataFrame.export*
- *DataFrame.export_hdf5*
- *DataFrame.export_arrow*
- *DataFrame.export_fits*

[ ]:

```
```

4.4 Plotting

4.4.1 1d and 2d

Most visualization can be done in 1 and 2d, and vaex wraps matplotlib to provide most use cases.

[18]:

```
import vaex
import numpy as np
df = vaex.example()
%matplotlib inline
```

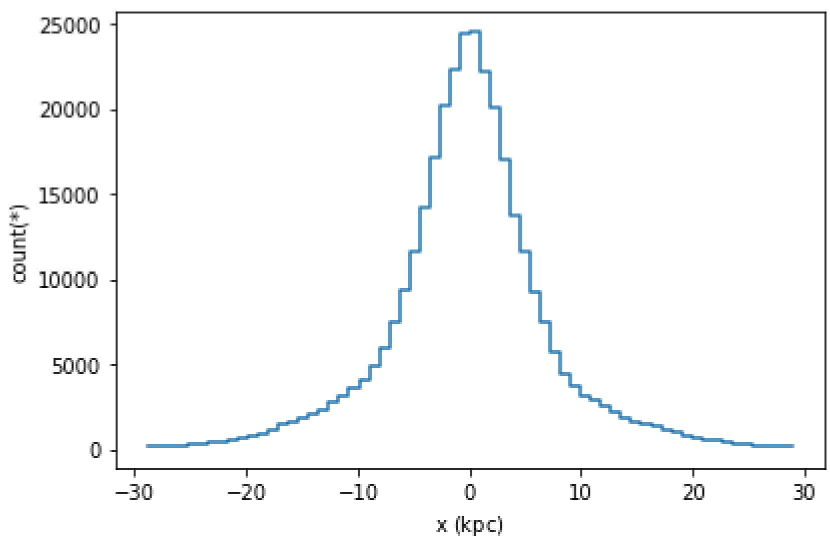
The simpelest visualization is a 1d plot using *DataFrame.plot1d*. When only given one arguments, it will show a histogram showing 99.8% of the data.

[19]:

```
df.plot1d(df.x)
```

[19]:

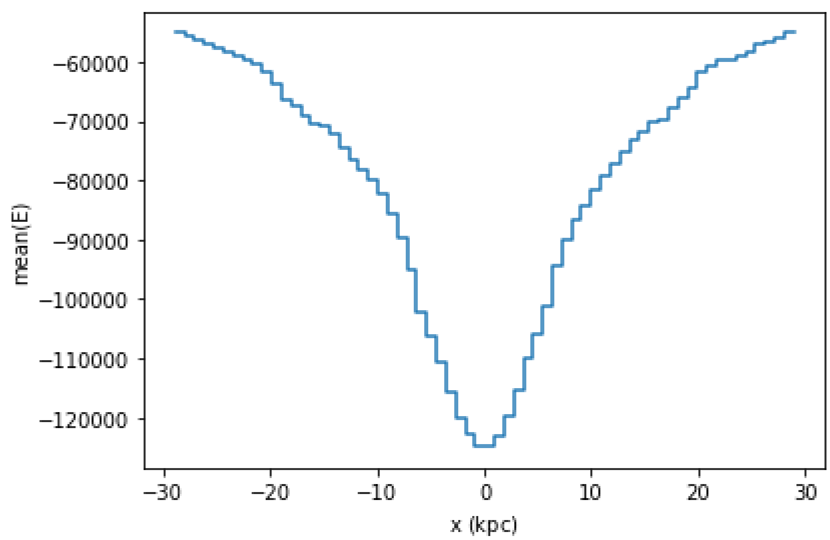
```
[<matplotlib.lines.Line2D at 0x11c3bb128>]
```



A slighly more complication visualization, is to not plot the counts, but a different statistic for that bin. In most cases, passing the `what='<statistic>(<expression>)` argument will do, where `<statistic>` is any of the statistics mentioned in the list above, or in the *API docs*

[20]:
```
df.plot1d(df.x, what='mean(E)')
```

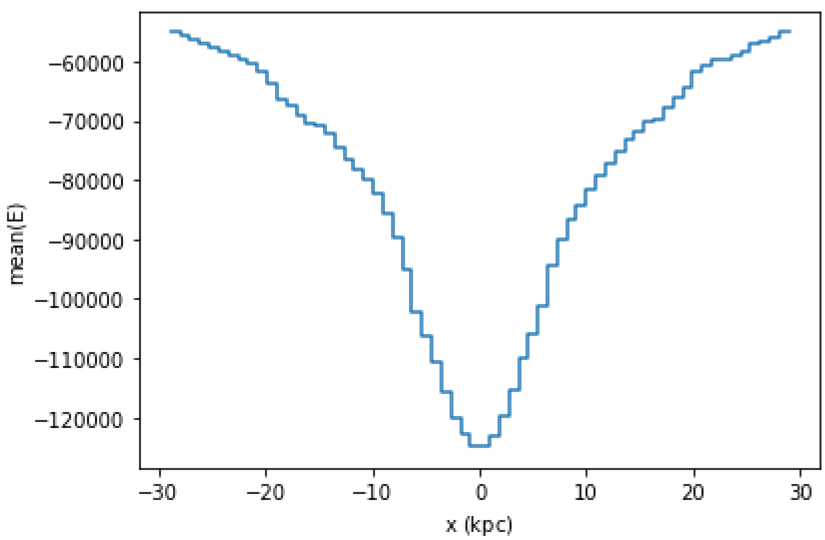
[20]:
```
[<matplotlib.lines.Line2D at 0x11c7d3898>]
```



An equivalent method, is to use the `vaex.stat.<statistic>` functions, e.g. *vaex.stat.mean*

[21]:
```
df.plot1d(df.x, what=vaex.stat.mean(df.E))
```

[21]:
```
[<matplotlib.lines.Line2D at 0x11e5df2b0>]
```



These objects are very similar to vaex' expression, in that they represent an underlying calculation, while normal arithmetic and numpy functions can be applied to it. However, these object represent a statistics computation, and not a column.

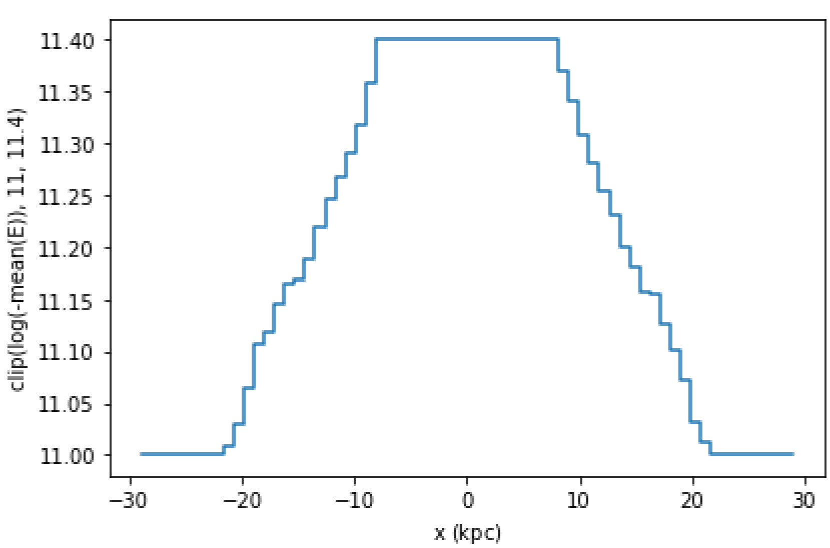
[22]:
```
np.log(vaex.stat.mean(df.x)/vaex.stat.std(df.x))
```

[22]:
```
log((mean(x) / std(x)))
```

These statistical objects can be passed to the what argument. The advantage being that the data will only have to be passed over once.

[23]:
```
df.plot1d(df.x, what=np.clip(np.log(-vaex.stat.mean(df.E)), 11, 11.4))
```

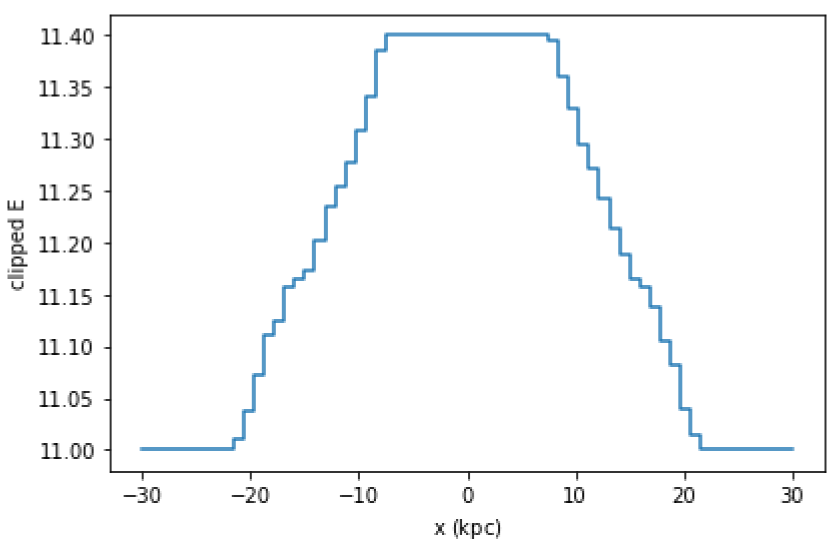
[23]:
```
[<matplotlib.lines.Line2D at 0x11e7381d0>]
```



A similar result can be obtained by calculating the statistic ourselves, and passing it to plot1d's grid argument. Care has to be taken that the limits used for calculating the statistics and the plot are the same, otherwise the x axis may not correspond to the real data.

[24]:
```
limits = [-30, 30]
shape  = 64
meanE  = df.mean(df.E, binby=df.x, limits=limits, shape=shape)
grid   = np.clip(np.log(-meanE), 11, 11.4)
df.plot1d(df.x, grid=grid, limits=limits, ylabel='clipped E')
```

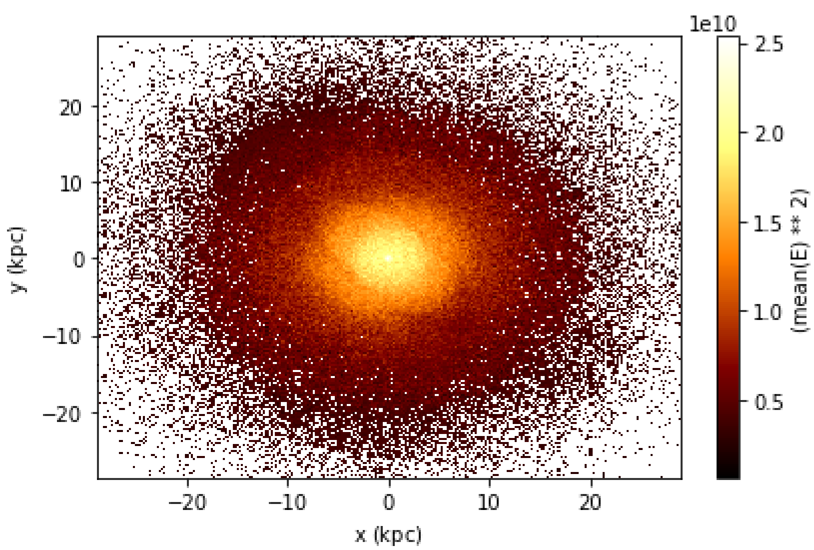
[24]:
```
[<matplotlib.lines.Line2D at 0x11c2dcac8>]
```



The same applies for 2d plotting.

[25]:
```
df.plot(df.x, df.y, what=vaex.stat.mean(df.E)**2)
```

[25]:
```
<matplotlib.image.AxesImage at 0x11e56c780>
```



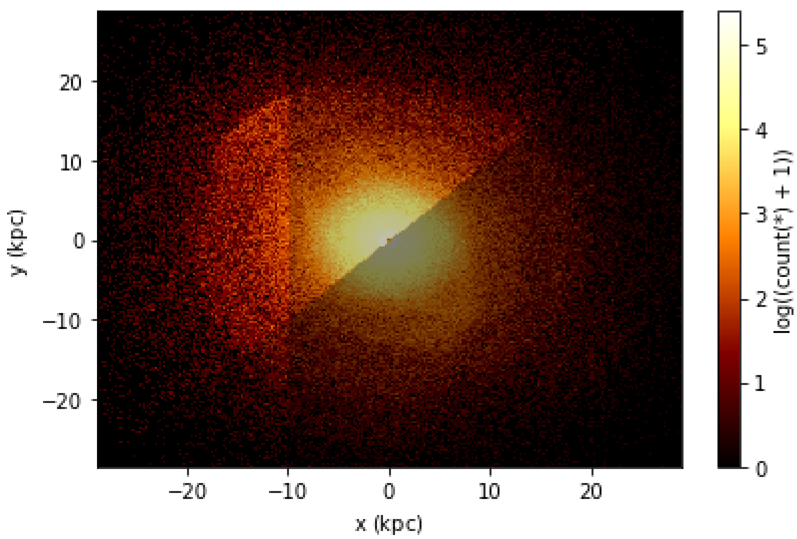
4.4.2 Selections for plotting

While filtering is useful for narrowing down a selection (e.g. `df_negative = df[df.x < 0]`) there are a few downsides to this. First, a practical issue is that when you filter 4 different ways, you will need to have 4 different objects, polluting your namespace. However, more importantly, when vaex executes a bunch of statistical computations, it will do that per DataFrame, meaning for 4 different DataFrames (although pointing to the same underlying data) it will do a total of 4 passes over the data. If instead, we have 4 (named) selections in our dataset, it can calculate statistics in one single pass over the data, which can speed up especially when you dataset is larger than your memory.

In the plot below, we show three selection, which by default are blended together, requiring just one pass over the data.

[26]:
```
df.plot(df.x, df.y, what=np.log(vaex.stat.count()+1),
        selection=[None, df.x < df.y, df.x < -10])
```

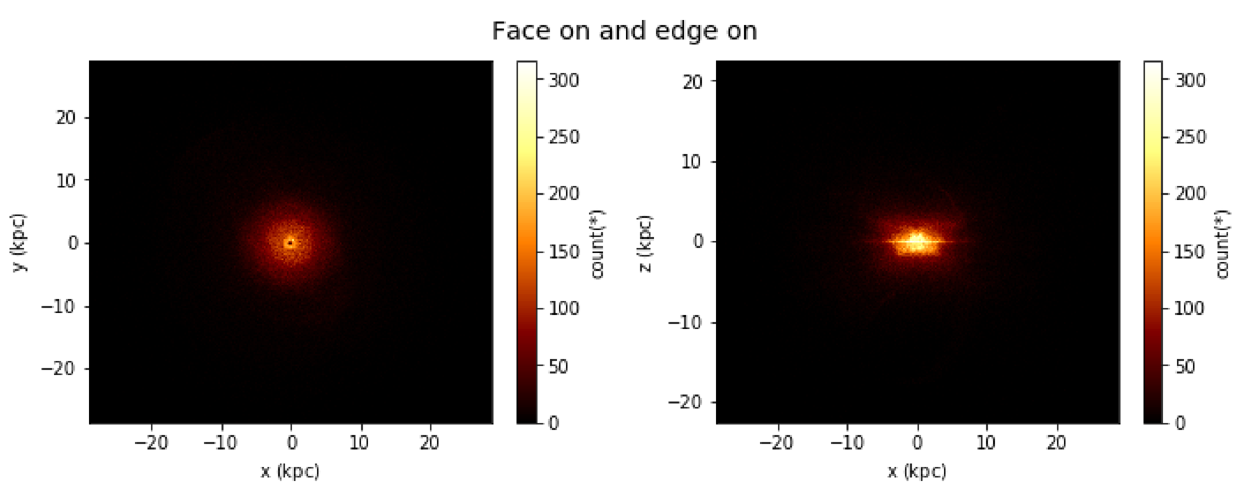
[26]:
```
<matplotlib.image.AxesImage at 0x11e7aab38>
```



4.4.3 Advanced Plotting

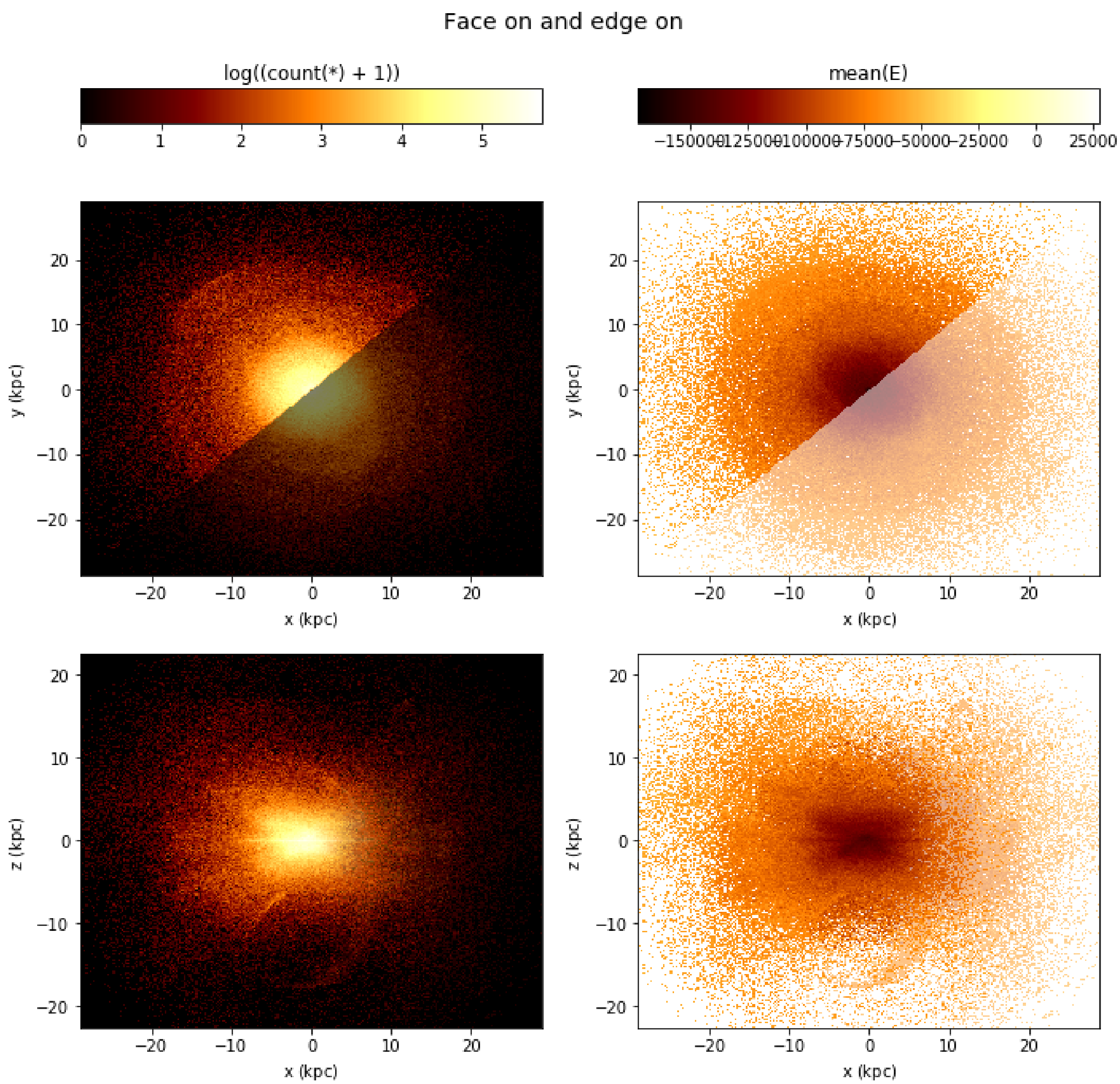
Lets say we would like to see two plots next to eachother, we can pass a list of expression pairs.

[27]:
```
df.plot([["x", "y"], ["x", "z"]],
        title="Face on and edge on", figsize=(10,4));
```



By default, if you have multiple plots, they are shows as columns, multiple selections are overplotted, and multiple 'whats' (statistics) are shows as rows.

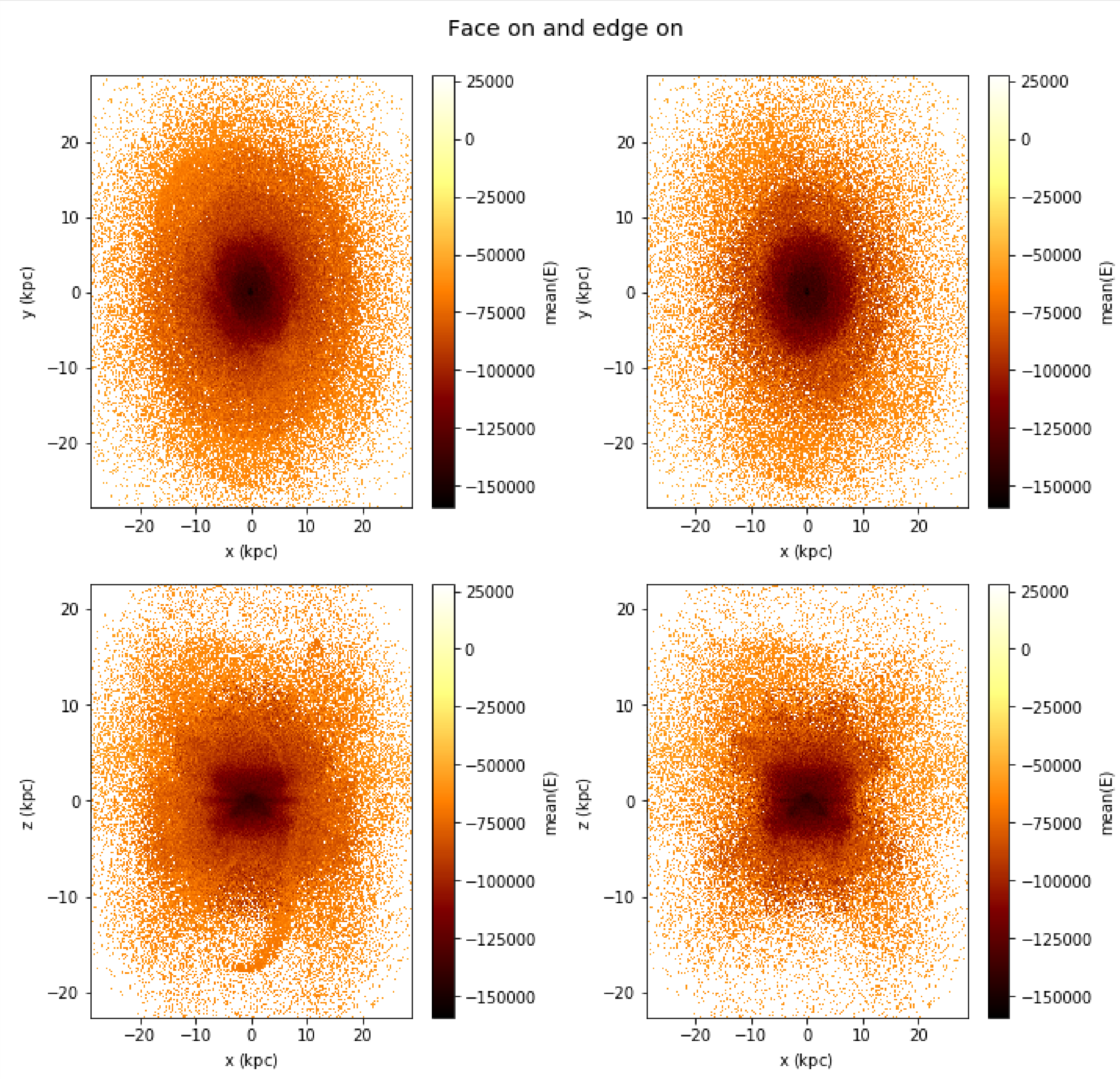
[28]:
```
df.plot([["x", "y"], ["x", "z"]],
        what=[np.log(vaex.stat.count()+1), vaex.stat.mean(df.E)],
        selection=[None, df.x < df.y],
        title="Face on and edge on", figsize=(10,10));
```



(Note that the selection has no effect in the bottom rows)

However, this behaviour can be changed using the `visual` argument.

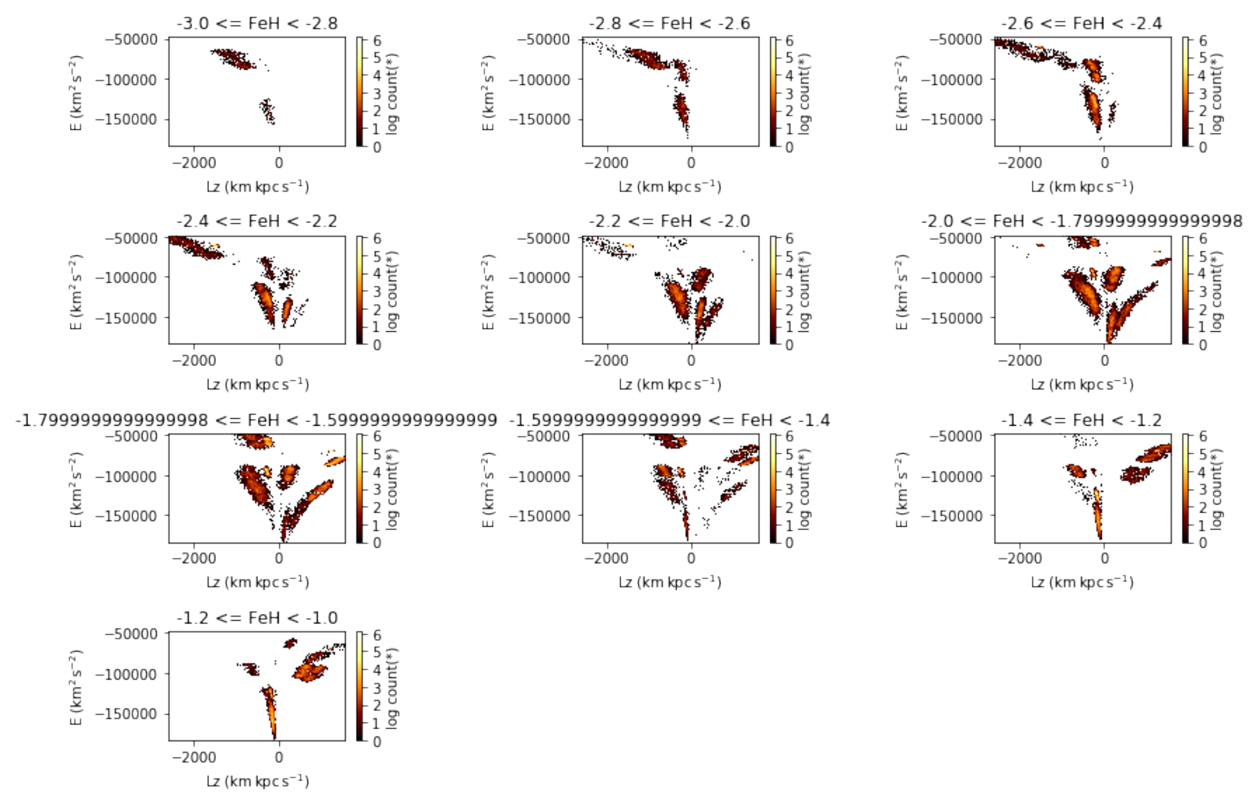
[29]:
```
df.plot([["x", "y"], ["x", "z"]],
        what=vaex.stat.mean(df.E),
        selection=[None, df.Lz < 0],
        visual=dict(column='selection'),
        title="Face on and edge on", figsize=(10,10));
```



4.4.4 Slices in a 3rd dimension

If a 3rd axis (z) is given, you can 'slice' through the data, displaying the z slices as rows. Note that here the rows are wrapped, which can be changed using the `wrap_columns` argument.

```
[30]: df.plot("Lz", "E", z="FeH:-3,-1,10", show=True, visual=dict(row="z"),
              figsize=(12,8), f="log", wrap_columns=3);
```

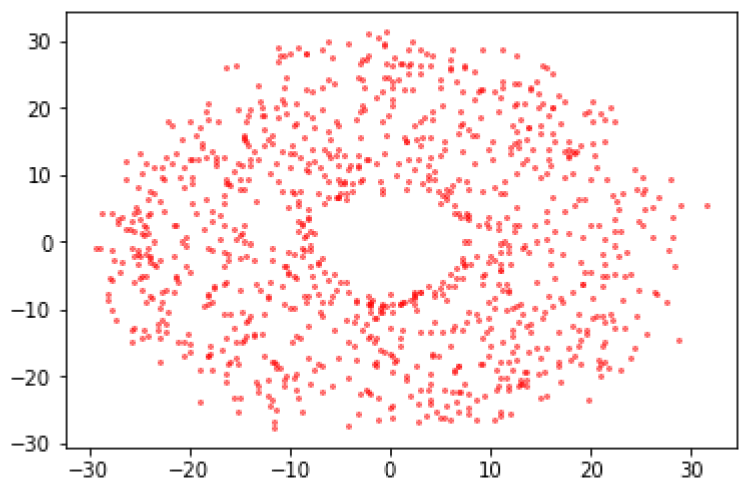


4.4.5 Smaller datasets / scatter plot

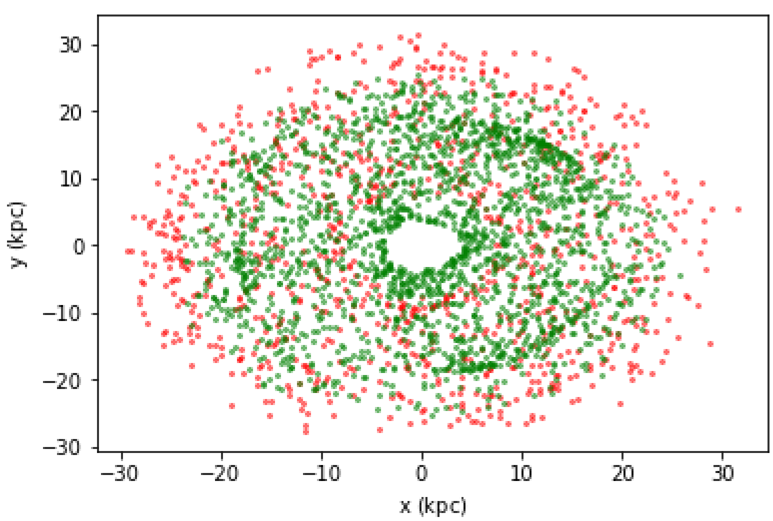
Although vaex focusses on large datasets, sometimes you end up with a fraction of the data (due to a selection) and you want to make a scatter plot. You could try the following approach:

[31]:
```
import vaex
df = vaex.example()
%matplotlib inline
```

[32]:
```
import matplotlib.pylab as plt
x = df.evaluate("x", selection=df.Lz < -2500)
y = df.evaluate("y", selection=df.Lz < -2500)
plt.scatter(x, y, c="red", alpha=0.5, s=4);
```



[33]:
```
df.scatter(df.x, df.y, selection=df.Lz < -2500, c="red", alpha=0.5, s=4)
df.scatter(df.x, df.y, selection=df.Lz > 1500, c="green", alpha=0.5, s=4);
```



4.4.6 In control

While vaex provides a wrapper for matplotlib, there are situations where you want to use the *DataFrame.plot* method, but want to be in control of the plot. Vaex simply uses the current figure and axes, so that it is easy to do.

[34]:
```
import numpy as np
```

[35]:
```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14,7))
plt.sca(ax1)
selection = df.Lz < -2500
x = df[selection].x.evaluate()#selection=selection)
y = df[selection].y.evaluate()#selection=selection)
df.plot(df.x, df.y)
plt.scatter(x, y)
```

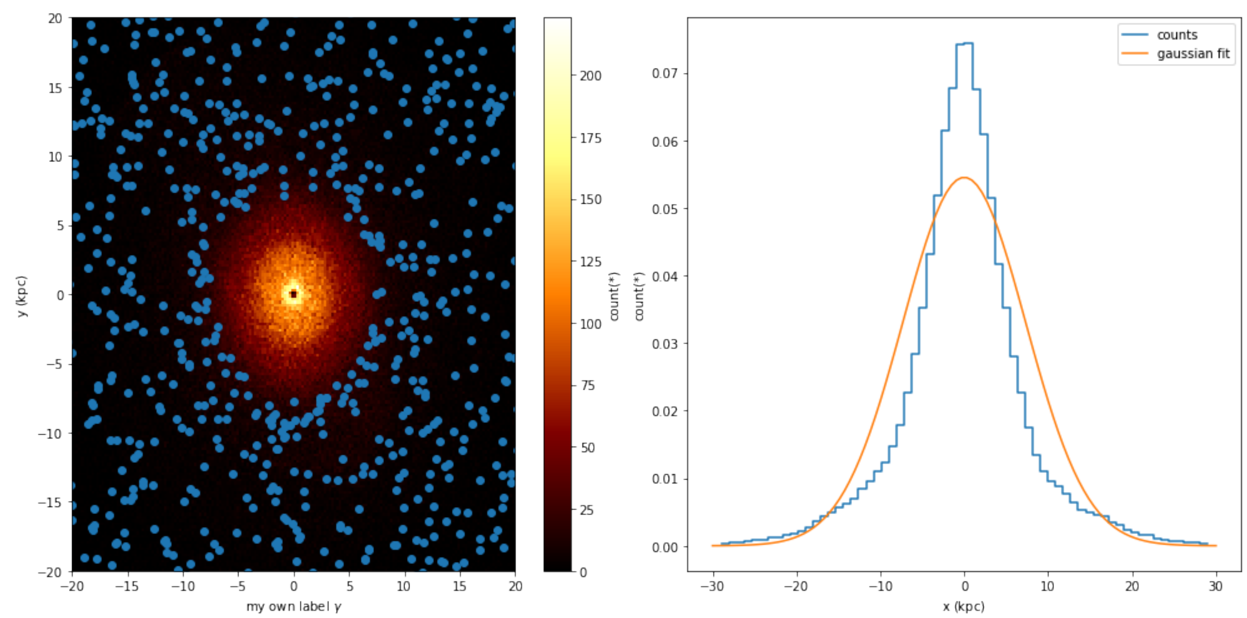
(continues on next page)

(continued from previous page)

```
plt.xlabel('my own label $\gamma$')
plt.xlim(-20, 20)
plt.ylim(-20, 20)

plt.sca(ax2)
df.plot1d(df.x, label='counts', n=True)
x = np.linspace(-30, 30, 100)
std = df.std(df.x.expression)
y = np.exp(-(x**2/std**2/2)) / np.sqrt(2*np.pi) / std
plt.plot(x, y, label='gaussian fit')
plt.legend()
```

[35]:
```
<matplotlib.legend.Legend at 0x11f963c18>
```



4.4.7 Healpix (Plotting)

Using healpix is made available by the vaex-healpix package using the healpy package. Vaex does not need special support for healpix, only for plotting, but some helper functions are introduced to make working with healpix easier. By diving the source_id by 34359738368 you get a healpix index level 12, and diving it further will take you to lower levels.

To understand healpix better, we will start from the beginning. If we want to make a density sky plot, we would like to pass healpy a 1d numpy array where each value represents the density at a location of the sphere, where the location is determined by the array size (the healpix level) and the offset (the location). Since the Gaia data includes the healpix index encoded in the `source_id`. By diving the source_id by 34359738368 you get a healpix index level 12, and diving it further will take you to lower levels.

[36]:
```
import vaex
import healpy as hp
%matplotlib inline
tgas = vaex.datasets.tgas.fetch()
```

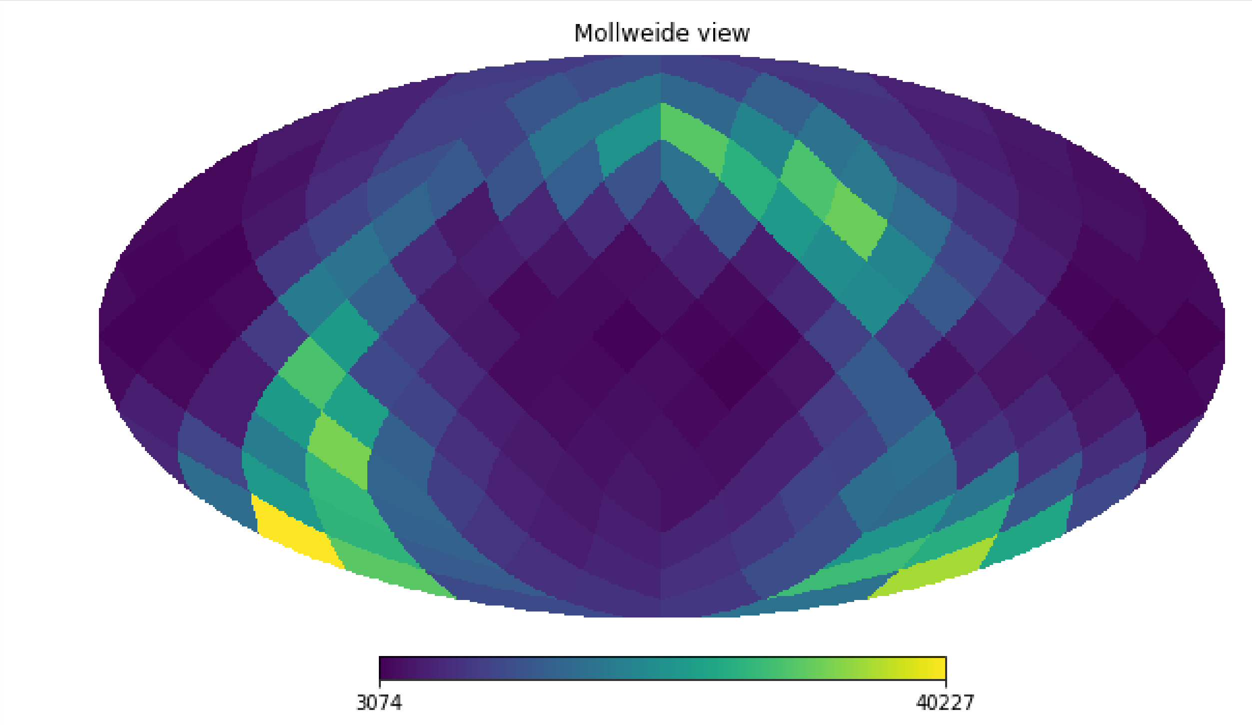
We will start showing how you could manually do statistics on healpix bins using vaex.count. We will do a really course healpix scheme (level 2).

[37]:
```
level = 2
factor = 34359738368 * (4**(12-level))
nmax = hp.nside2npix(2**level)
epsilon = 1e-16
counts = tgas.count(binby=tgas.source_id/factor, limits=[-epsilon, nmax-epsilon],␣
↪shape=nmax)
counts
```

[37]:
```
array([ 4021.,  6171.,  5318.,  7114.,  5755., 13420., 12711., 10193.,
        7782., 14187., 12578., 22038., 17313., 13064., 17298., 11887.,
        3859.,  3488.,  9036.,  5533.,  4007.,  3899.,  4884.,  5664.,
       10741.,  7678., 12092., 10182.,  6652.,  6793., 10117.,  9614.,
        3727.,  5849.,  4028.,  5505.,  8462., 10059.,  6581.,  8282.,
        4757.,  5116.,  4578.,  5452.,  6023.,  8340.,  6440.,  8623.,
        7308.,  6197., 21271., 23176., 12975., 17138., 26783., 30575.,
       31931., 29697., 17986., 16987., 19802., 15632., 14273., 10594.,
        4807.,  4551.,  4028.,  4357.,  4067.,  4206.,  3505.,  4137.,
        3311.,  3582.,  3586.,  4218.,  4529.,  4360.,  6767.,  7579.,
       14462., 24291., 10638., 11250., 29619.,  9678., 23322., 18205.,
        7625.,  9891.,  5423.,  5808., 14438., 17251.,  7833., 15226.,
        7123.,  3708.,  6135.,  4110.,  3587.,  3222.,  3074.,  3941.,
        3846.,  3402.,  3564.,  3425.,  4125.,  4026.,  3689.,  4084.,
       16617., 13577.,  6911.,  4837., 13553., 10074.,  9534., 20824.,
        4976.,  6707.,  5396.,  8366., 13494., 19766., 11012., 16130.,
        8521.,  8245.,  6871.,  5977.,  8789., 10016.,  6517.,  8019.,
        6122.,  5465.,  5414.,  4934.,  5788.,  6139.,  4310.,  4144.,
       11437., 30731., 13741., 27285., 40227., 16320., 23039., 10812.,
       14686., 27690., 15155., 32701., 18780.,  5895., 23348.,  6081.,
       17050., 28498., 35232., 26223., 22341., 15867., 17688.,  8580.,
       24895., 13027., 11223.,  7880.,  8386.,  6988.,  5815.,  4717.,
        9088.,  8283., 12059.,  9161.,  6952.,  4914.,  6652.,  4666.,
       12014., 10703., 16518., 10270.,  6724.,  4553.,  9282.,  4981.])
```

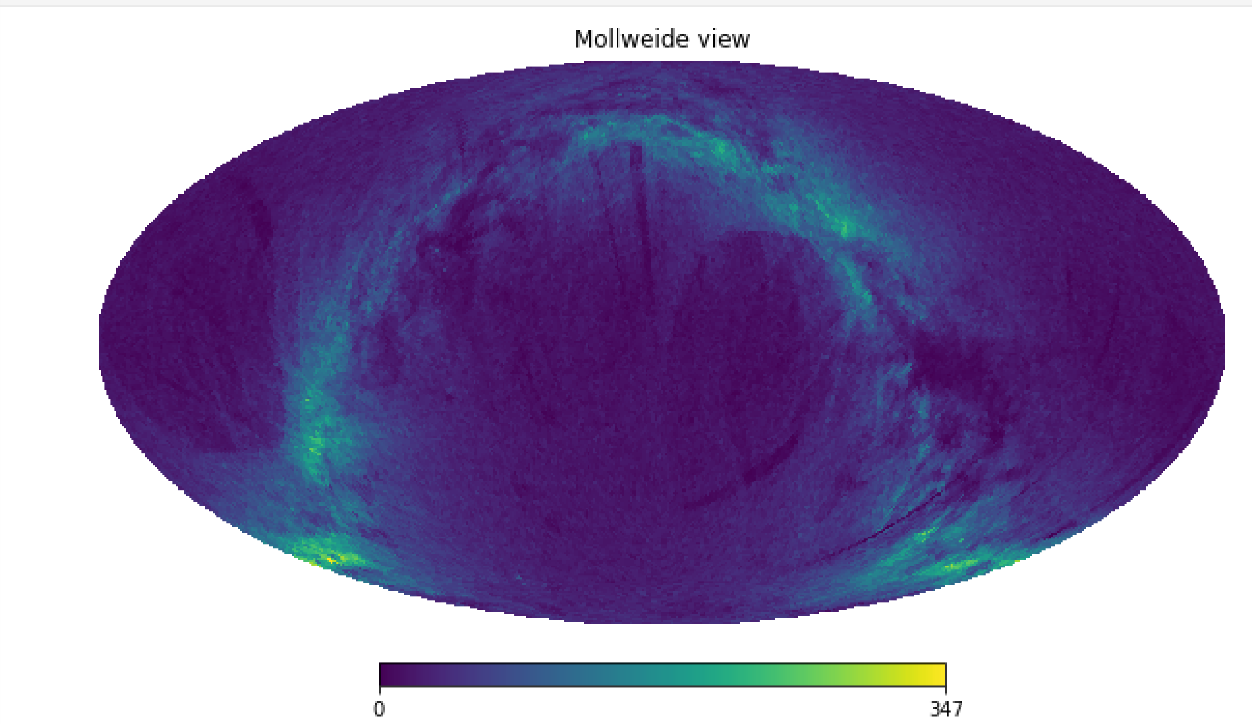
And using healpy's mollview we can visualize this.

[38]:
```
hp.mollview(counts, nest=True)
```



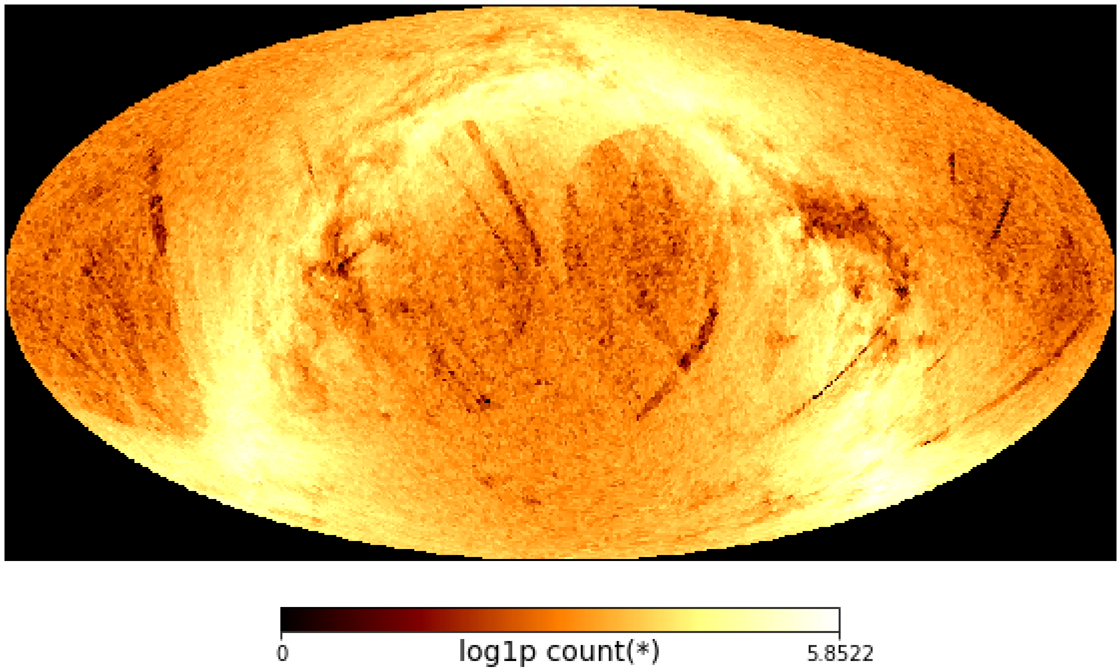
To simplify life, vaex includes *DataFrame.healpix_count* to take care of this.

[39]:
```
counts = tgas.healpix_count(healpix_level=6)
hp.mollview(counts, nest=True)
```



Or even simpler, use *DataFrame.healpix_plot*

[40]:
```
tgas.healpix_plot(f="log1p", healpix_level=6, figsize=(10,8),
                  healpix_output="ecliptic")
```



4.5 Propagation of uncertainties

In science we often deal with measurement uncertainties (sometimes refererred to as measurement errors). When transformations are made with quantities that have uncertainties associated with them, the uncertainties on these transformed quantities can be calculated automatically by vaex. Note that propagation of uncertainties requires derivatives and matrix multiplications of lengthy equations, which is not complex, but tedious. Vaex can automatically calculate all dependencies, derivatives and compute the full covariance matrix.

[41]:
```
import vaex
import pylab as plt
%matplotlib inline
tgas = vaex.datasets.tgas_1percent.fetch()
```

Even though the TGAS dataset already contains galactic sky coordiantes (l and b), we add them again as virtual columns such that the transformation between RA. and Dec. and the galactic sky coordinates is know.

[42]:
```
# convert parallas to distance
tgas.add_virtual_columns_distance_from_parallax(tgas.parallax)
# 'overwrite' the real columns 'l' and 'b' with virtual columns
tgas.add_virtual_columns_eq2gal('ra', 'dec', 'l', 'b')
# and combined with the galactic sky coordinates gives galactic cartesian coordinates
↪of the stars
tgas.add_virtual_columns_spherical_to_cartesian(tgas.l, tgas.b, tgas.distance, 'x', 'y
↪', 'z')
```

```
j2000
```

Since RA. and Dec. are in degrees, while ra_error and dec_error is in miliarcseconds, so we put them on the same scale

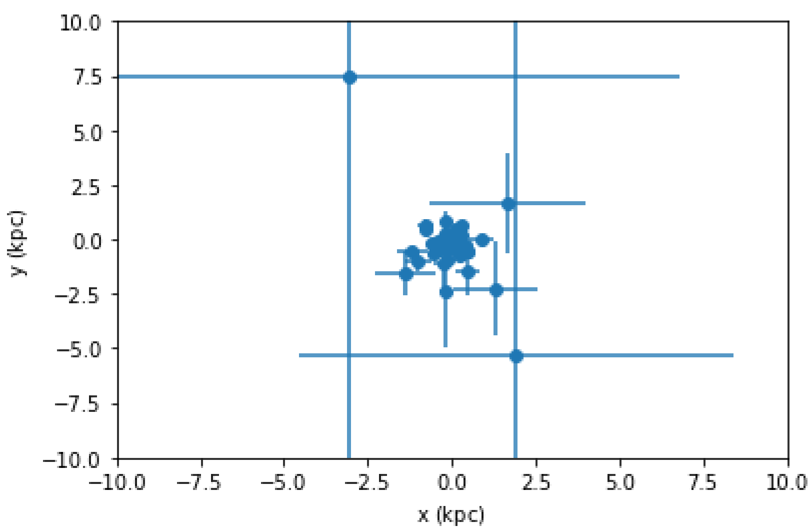
[43]:
```
tgas['ra_error'] = tgas.ra_error / 1000 / 3600
tgas['dec_error'] = tgas.dec_error / 1000 / 3600
```

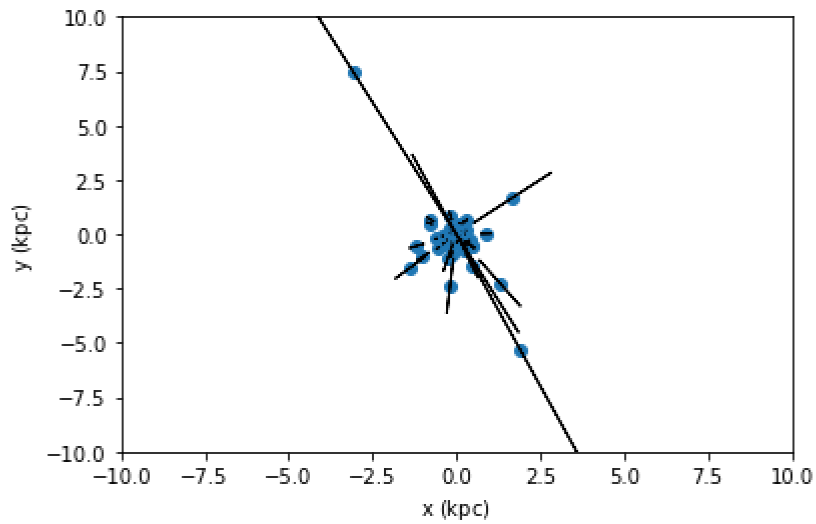
We now let vaex sort out what the covariance matrix is for the cartesian coordinates x, y, and z. And take 50 samples from the datasets for visualization.

[44]:
```
tgas.propagate_uncertainties([tgas.x, tgas.y, tgas.z])
tgas_50 = tgas.sample(50, random_state=42)
```

For this small dataset we visualize the uncertainties, with and without the covariance.

[45]:
```
tgas_50.scatter(tgas_50.x, tgas_50.y, xerr=tgas_50.x_uncertainty, yerr=tgas_50.y_
↪uncertainty)
plt.xlim(-10, 10)
plt.ylim(-10, 10)
plt.show()
tgas_50.scatter(tgas_50.x, tgas_50.y, xerr=tgas_50.x_uncertainty, yerr=tgas_50.y_
↪uncertainty, cov=tgas_50.y_x_covariance)
plt.xlim(-10, 10)
plt.ylim(-10, 10)
plt.show()
```





From the second plot, we see that showing error ellipses (so narrow that they appear as lines) instead of error bars reveal that the distance information dominates the uncertainty in this case.

4.6 Parallel computations

As mentioned in the sections on selections, vaex can do computations on a DataFrame in parallel. Often, this is taken care of, when for instance passing multiple selections, or multiple arguments to one of the statistical functions. However, sometimes it is difficult or impossible to express a computation in one expression, and we need to resort to doing so called 'delayed' computationed, similar as in joblib and dask.

[46]:
```
import vaex
df = vaex.example()
limits = [-10, 10]
delayed_count = df.count(df.E, binby=df.x, limits=limits,
                         shape=4, delay=True)
delayed_count
```

[46]:
```
<vaex.promise.Promise at 0x108245630>
```

Note that now the returned value is not a promise (TODO: a more Pythonic way would be to return a Future). This may be subject to change, and the best way to work with this is to use the *delayed* decorator. And call *DataFrame.execute* when the result is needed.

In addition to the above delayed computation, we schedule another computation, such that both the count and mean are execute in parallel such that we only do a single pass over the data. We schedule the execution of two extra functions using the `vaex.delayed` decorator, and run the whole pipeline using `df.execute()`.

[47]:
```
delayed_sum = df.sum(df.E, binby=df.x, limits=limits,
                     shape=4, delay=True)

@vaex.delayed
def calculate_mean(sums, counts):
    print('calculating mean')
    return sums/counts

print('before calling mean')
```

(continues on next page)

(continued from previous page)

```
# since calculate_mean is decorator with vaex.delayed
# this now also returns a 'delayed' object (a promise)
delayed_mean = calculate_mean(delayed_sum, delayed_count)

# if we'd like to perform operations on that, we can again
# use the same decorator
@vaex.delayed
def print_mean(means):
    print('means', means)
print_mean(delayed_mean)

print('before calling execute')
df.execute()

# Using the .get on the promise will also return the resut
# However, this will only work after execute, and may be
# subject to change
means = delayed_mean.get()
print('same means', means)
```

```
before calling mean
before calling execute
calculating mean
means [ -94415.16581227 -118856.63989386 -118919.86423543  -95000.5998913 ]
same means [ -94415.16581227 -118856.63989386 -118919.86423543  -95000.5998913 ]
```

4.7 Interactive widgets

Note: The interactive widgets require a running Python kernel, if you are viewing this documentation online you mean get a feeling for what the widgets can do, but computation will not be possible!

Using the `vaex-jupyter` package, we get access to interactive widgets.

[48]:
```
import vaex
import vaex.jupyter
import numpy as np
import pylab as plt
%matplotlib inline
df = vaex.example()
```

The simplest way to get a more interactive visualization (or even print out statistics) is to the the `vaex.jupyter.interactive_selection` decorator, which will execute the decorated function each time the selection is changed.

[50]:
```
df.select(df.x > 0)
@vaex.jupyter.interactive_selection(df)
def plot():
    print("Mean x for the selection is:", df.mean(df.x, selection=True))
    df.plot(df.x, df.y, what=np.log(vaex.stat.count()+1), selection=[None, True])
    plt.show()
```

```
Output()
```

After changing the selection programmatically, the visualization will update, as well as the print output.

[51]:
```
df.select(df.x > df.y)
```

However, to get truly interactive visualization, we need to use widgets, such as the bqplot library. Again, if we make a selection here, the above visualization will also update, so lets select a square region. One issue is that if you have installed ipywidget, bqplot, ipyvolume etc, it may not be enabled if you installed them from pip (from conda-forge will enabled it automagically). To enable it, run the next cell, and refresh the notebook if there were not enabled already. *(Note that these commands will execute in the environment where the notebook is running, not where the kernel is running)*

[52]:
```
import sys
!jupyter nbextension enable --sys-prefix --py widgetsnbextension
!jupyter nbextension enable --sys-prefix --py bqplot
!jupyter nbextension enable --sys-prefix --py ipyvolume
!jupyter nbextension enable --sys-prefix --py ipympl
!jupyter nbextension enable --sys-prefix --py ipyleaflet
```

```
Enabling notebook extension jupyter-js-widgets/extension...
      - Validating: OK
Enabling notebook extension bqplot/extension...
      - Validating: OK
Enabling notebook extension ipyvolume/extension...
      - Validating: OK
Enabling notebook extension jupyter-matplotlib/extension...
      - Validating: OK
Enabling notebook extension jupyter-leaflet/extension...
      - Validating: OK
```

[53]:
```
# the default backend is bqplot, but we pass it here explicity
df.plot_widget(df.x, df.y, f='log1p', backend='bqplot')
```

```
VBox(children=(HBox(children=(VBox(children=(VBox(children=(VBox(children=(HBox(children=(ToggleButto
```

4.8 Joining

Joining in vaex is similar to pandas, except the data will no be copied. Internally an index array is kept for each row on the left DataFrame, pointing to the right DataFrame, requiring about 8GB for a billion row 10^9 dataset. Lets start with 2 small DataFrames, `df1` and `df2`:

[56]:
```
a = np.array(['a', 'b', 'c'])
x = np.arange(1,4)
df1 = vaex.from_arrays(a=a, x=x)
df1
```

[56]:

#	a	x
0	a	1
1	b	2
2	c	3

```
[57]: b = np.array(['a', 'b', 'd'])
      y = x**2
      df2 = vaex.from_arrays(b=b, y=y)
      df2
```

[57]:

#	b	y
0	a	1
1	b	4
2	d	9

The default join, is a 'left' join, where all rows for the left DataFrame (`df1`) are kept, and matching rows of the right DataFrame (`df2`) are added. We see for for the columns b and y, some values are missing, as expected.

```
[58]: df1.join(df2, left_on='a', right_on='b')
```

[58]:

#	a	x	b	y
0	a	1	a	1
1	b	2	b	4
2	c	3	--	--

A 'right' join, is basically the same, but now the roles of the left and right label swapped, so now we have some values from columns x and a missing.

```
[59]: df1.join(df2, left_on='a', right_on='b', how='right')
```

[59]:

#	b	y	a	x
0	a	1	a	1
1	b	4	b	2
2	d	9	--	--

Other joins (inner and outer) aren't supported, feel free open an issue on github for this.

4.9 Just-In-Time compilation

Lets start with a function that converts from two angles, to an angular distance. The function assumes as input, 2 pairs on angular coordinates, in radians.

```
[60]: import vaex
      import numpy as np
      # From http://pythonhosted.org/pythran/MANUAL.html
      def arc_distance(theta_1, phi_1, theta_2, phi_2):
          """
          Calculates the pairwise arc distance
          between all points in vector a and b.
          """
          temp = (np.sin((theta_2-2-theta_1)/2)**2
                 + np.cos(theta_1)*np.cos(theta_2) * np.sin((phi_2-phi_1)/2)**2)
          distance_matrix = 2 * np.arctan2(np.sqrt(temp), np.sqrt(1-temp))
          return distance_matrix
```

Let us use the New York Taxi dataset of 2015, *as can be downloaded in hdf5 format*

```
[61]: nytaxi = vaex.open("/Users/maartenbreddels/datasets/nytaxi/nyc_taxi2015.hdf5")
      # lets use just 20% of the data, since we want to make sure it fits
      # into memory (so we don't measure just hdd/ssd speed)
      nytaxi.set_active_fraction(0.2)
```

Although the function above expected numpy arrays, vaex can pass in columns or expression, which will delay execution till needed, and add the resulting expression as a virtual column.

[62]:
```
nytaxi['arc_distance'] = arc_distance(nytaxi.pickup_longitude * np.pi/180,
                                      nytaxi.pickup_latitude * np.pi/180,
                                      nytaxi.dropoff_longitude * np.pi/180,
                                      nytaxi.dropoff_latitude * np.pi/180)
```

When we calculate the mean angular distance of a taxi trip, we encounter some invalid data, that will give warnings, which we can savely ignore for this demonstration.

[63]:
```
%%time
nytaxi.mean(nytaxi.arc_distance)
```

```
CPU times: user 8.61 s, sys: 3.79 s, total: 12.4 s
Wall time: 4.09 s
```

[63]:
```
1.9999877196036897
```

This computation uses quite some heavy mathematical operation, and since it's (internally) using numpy arrays, also uses quite some temporary arrays. We can optimize this calculation by doing a Just-In-Time compilation, based on numba or pythran. Choose whichever gives the best performance or is easiest to install.

[64]:
```
nytaxi['arc_distance_jit'] = nytaxi.arc_distance.jit_numba()
# nytaxi['arc_distance_jit'] = nytaxi.arc_distance.jit_pythran()
```

[65]:
```
%%time
nytaxi.mean(nytaxi.arc_distance_jit)
```

```
CPU times: user 3.43 s, sys: 25 ms, total: 3.46 s
Wall time: 609 ms
```

[65]:
```
1.9999877196037037
```

We can that we can get a significant speedup ($4x$) in this case.

4.10 String processing

String processing is similar to Pandas, except all operations are performed lazily, multithreaded, and faster (in C++). Check the *API docs* for more examples.

[3]:
```
import vaex
text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
df = vaex.from_arrays(text=text)
df
```

[3]:

#	text
0	Something
1	very pretty
2	is coming
3	our
4	way.

[4]:
```
df.text.str.upper()
```

[4]:
```
Expression = str_upper(text)
Length: 5 dtype: str (expression)
```

(continues on next page)

(continued from previous page)

```
--------------------------------
0    SOMETHING
1  VERY PRETTY
2    IS COMING
3          OUR
4         WAY.
```

[10]:
```
df.text.str.title().str.replace('et', 'ET')
```

[10]:
```
Expression = str_replace(str_title(text), 'et', 'ET')
Length: 5 dtype: str (expression)
--------------------------------
0    SomEThing
1  Very PrETty
2    Is Coming
3          Our
4         Way.
```

[12]:
```
df.text.str.contains('e')
```

[12]:
```
Expression = str_contains(text, 'e')
Length: 5 dtype: bool (expression)
---------------------------------
0   True
1   True
2  False
3  False
4  False
```

[13]:
```
df.text.str.count('e')
```

[13]:
```
Expression = str_count(text, 'e')
Length: 5 dtype: int64 (expression)
----------------------------------
0  1
1  2
2  0
3  0
4  0
```

4.11 Extending vaex

Vaex can be extended using several mechanisms.

4.11.1 Adding functions

Use the vaex.register_function dectorator API to add new functions.

[1]:
```
import vaex
import numpy as np
@vaex.register_function()
def add_one(ar):
    return ar+1
```

The function can be invoked using the `df.func` accessor, to return a new expression. Each argument that is an expresssion, will be replaced by a numpy array on evaluations in any vaex context.

[2]:
```
df = vaex.from_arrays(x=np.arange(4))
df.func.add_one(df.x)
```

[2]:
```
Expression = add_one(x)
Length: 4 dtype: int64 (expression)
-----------------------------------
0  1
1  2
2  3
3  4
```

By default (passing `on_expression=True`), the function is also available as a method on Expressions, where the expression itself is automatically set as the first argument (since this is a quite common use case).

[3]:
```
df.x.add_one()
```

[3]:
```
Expression = add_one(x)
Length: 4 dtype: int64 (expression)
-----------------------------------
0  1
1  2
2  3
3  4
```

In case the first argument is not an expression, pass `on_expression=True`, and use `df.func.<funcname>`, to build a new expression using the function:

[4]:
```
@vaex.register_function(on_expression=False)
def addmul(a, b, x, y):
    return a*x + b * y
```

[5]:
```
df = vaex.from_arrays(x=np.arange(4))
df['y'] = df.x**2
df.func.addmul(2, 3, df.x, df.y)
```

[5]:
```
Expression = addmul(2, 3, x, y)
Length: 4 dtype: int64 (expression)
-----------------------------------
0   0
1   5
2  16
3  33
```

These expressions can be added as virtual columns, as expected.

[6]:
```
df = vaex.from_arrays(x=np.arange(4))
df['y'] = df.x**2
df['z'] = df.func.addmul(2, 3, df.x, df.y)
df['w'] = df.x.add_one()
df
```

[6]:

#	x	y	z	w
0	0	0	0	1
1	1	1	5	2
2	2	4	16	3
3	3	9	33	4

4.11.2 Adding DataFrame accessors

To add methods that operate on dataframes, it makes sense to group them together in a single namespace.

[7]:
```
@vaex.register_dataframe_accessor('scale', override=True)
class ScalingOps(object):
    def __init__(self, df):
        self.df = df

    def mul(self, a):
        df = self.df.copy()
        for col in df.get_column_names(strings=False):
            if df[col].dtype:
                df[col] = df[col] * a
        return df

    def add(self, a):
        df = self.df.copy()
        for col in df.get_column_names(strings=False):
            if df[col].dtype:
                df[col] = df[col] + a
        return df
```

[8]:
```
df.scale.add(1)
```

[8]:

#	x	y	z	w
0	1	1	1	2
1	2	2	6	3
2	3	5	17	4
3	4	10	34	5

[9]:
```
df.scale.mul(2)
```

[9]:

#	x	y	z	w
0	0	0	0	2
1	2	2	10	4
2	4	8	32	6
3	6	18	66	8

[ ]:

CHAPTER 5

Examples

5.1 Arrow

Vaex supports Arrow. We will demonstrate vaex+arrow by giving a quick look at a large dataset that does not fit into memory. The NYC taxi dataset for the year 2015 contains about 150 million rows containing information about taxi trips in New York, and is about 23GB in size. You can download it here:

- https://docs.vaex.io/en/latest/datasets.html

In case you want to convert it to the arrow format, use the code below:

```
ds_hdf5 = vaex.open('/Users/maartenbreddels/datasets/nytaxi/nyc_taxi2015.hdf5')
# this may take a while to export
ds_hdf5.export('./nyc_taxi2015.arrow')
```

Also make sure you install vaex-arrow:

```
$ pip install vaex-arrow
```

[1]:
```
!ls -alh /Users/maartenbreddels/datasets/nytaxi/nyc_taxi2015.arrow
```

```
-rw-r--r--  1 maartenbreddels  staff    23G Oct 31 18:56 /Users/maartenbreddels/
↪datasets/nytaxi/nyc_taxi2015.arrow
```

[3]:
```
import vaex
```

5.1.1 Opens instantly

Opening the file goes instantly, since nothing is being copied to memory. The data is only memory mapped, a technique that will only read the data when needed.

[4]:
```
%time
df = vaex.open('/Users/maartenbreddels/datasets/nytaxi/nyc_taxi2015.arrow')
```

```
CPU times: user 3 µs, sys: 1 µs, total: 4 µs
Wall time: 6.91 µs
```

[5]:
```
df
```

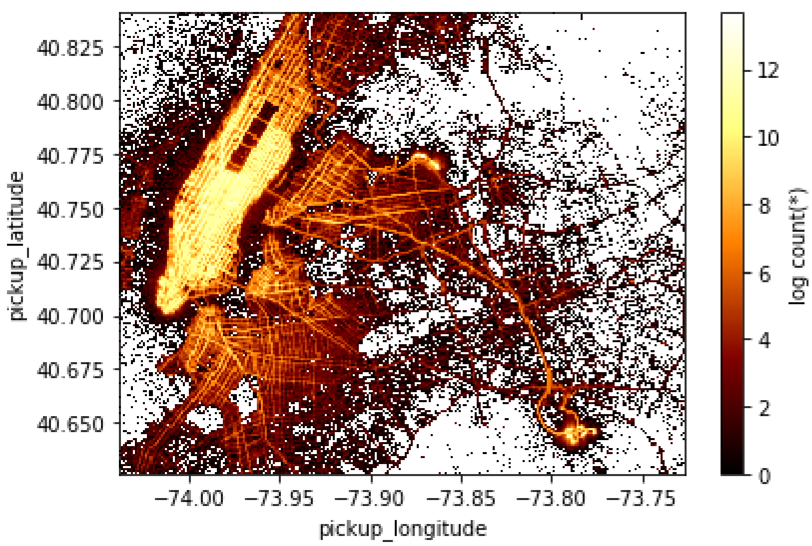
```
<IPython.core.display.HTML object>
```

[5]:
```
<vaex_arrow.dataset.DatasetArrow at 0x11d87e6a0>
```

5.1.2 Quick viz of 146 million rows

As can be seen, this dataset contains 146 million rows. Using plot, we can generate a quick overview what the data contains. The pickup locations nicely outline Manhattan.

[6]:
```
df.plot(df.pickup_longitude, df.pickup_latitude, f='log');
```



[7]:
```
df.total_amount.minmax()
```

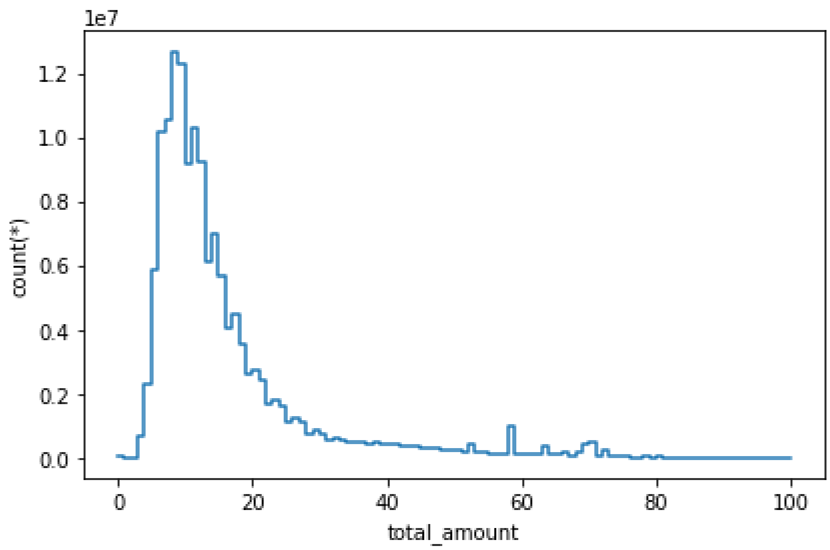
[7]:
```
array([-4.9630000e+02,  3.9506116e+06])
```

5.1.3 Data cleansing: outliers

As can be seen from the total_amount columns (how much people payed), this dataset contains outliers. From a quick 1d plot, we can see reasonable ways to filter the data

[8]:
```
df.plot1d(df.total_amount, shape=100, limits=[0, 100])
```

[8]:
```
[<matplotlib.lines.Line2D at 0x121d26320>]
```



[9]:
```
# filter the dataset
dff = df[(df.total_amount >= 0) & (df.total_amount < 100)]
```

5.1.4 Shallow copies

This filtered dataset did not copy any data (otherwise it would have costed us about ~23GB of RAM). Shallow copies of the data are made instead and a booleans mask tracks which rows should be used.

[10]:
```
dff['ratio'] = dff.tip_amount/dff.total_amount
```

5.1.5 Virtual column

The new column `ratio` does not do any computation yet, it only stored the expression and does not waste any memory. However, the new (virtual) column can be used in calculations as if it were a normal column.

[11]:
```
dff.ratio.mean()
```

```
<string>:1: RuntimeWarning: invalid value encountered in true_divide
```

[11]:
```
0.09601926650107262
```

5.1.6 Result

Our final result, the percentage of the tip, can be easily calcualted for this large dataset, it did not require any excessive amount of memory.

5.1.7 Interoperability

Since the data lives as Arrow arrays, we can pass them around to other libraries such as pandas, or even pass it to other processes.

[12]:
```
arrow_table = df.to_arrow_table()
arrow_table
```

[12]:
```
pyarrow.Table
VendorID: int64
dropoff_dayofweek: double
dropoff_hour: double
dropoff_latitude: double
dropoff_longitude: double
extra: double
fare_amount: double
improvement_surcharge: double
mta_tax: double
passenger_count: int64
payment_type: int64
pickup_dayofweek: double
pickup_hour: double
pickup_latitude: double
pickup_longitude: double
tip_amount: double
tolls_amount: double
total_amount: double
tpep_dropoff_datetime: timestamp[ns]
tpep_pickup_datetime: timestamp[ns]
trip_distance: double
```

[13]:
```
# Although you can 'convert' (pass the data) in to pandas,
# some memory will be wasted (at least an index will be created by pandas)
# here we just pass a subset of the data
df_pandas = df[:10000].to_pandas_df()
df_pandas
```

[13]:
```
      VendorID  dropoff_dayofweek  dropoff_hour  dropoff_latitude  \
0            2                3.0          19.0         40.750618
1            1                5.0          20.0         40.759109
2            1                5.0          20.0         40.824413
3            1                5.0          20.0         40.719986
4            1                5.0          20.0         40.742653
5            1                5.0          20.0         40.758194
6            1                5.0          20.0         40.749634
7            1                5.0          20.0         40.726326
8            1                5.0          21.0         40.759357
9            1                5.0          20.0         40.759365
10           1                5.0          20.0         40.728584
11           1                5.0          20.0         40.757217
12           1                5.0          20.0         40.707726
13           1                5.0          21.0         40.735210
14           1                5.0          20.0         40.739895
15           2                3.0          19.0         40.757889
16           2                3.0          19.0         40.786858
17           2                3.0          19.0         40.785782
18           2                3.0          19.0         40.786083
19           2                3.0          19.0         40.718590
20           2                3.0          19.0         40.714596
21           2                3.0          19.0         40.734650
22           2                3.0          19.0         40.735512
23           2                3.0          19.0         40.704220
24           2                3.0          19.0         40.761856
```

(continues on next page)

(continued from previous page)

```
25          2              3.0          19.0         40.811089
26          2              3.0          19.0         40.734890
27          2              3.0          19.0         40.743530
28          2              3.0          19.0         40.757721
29          2              3.0          19.0         40.704689
...       ...              ...           ...               ...
9970        1              4.0          11.0         40.719917
9971        1              4.0          10.0         40.720398
9972        1              4.0          11.0         40.755405
9973        2              1.0          19.0         40.763626
9974        2              1.0          19.0         40.772366
9975        2              1.0          19.0         40.733429
9976        2              1.0          19.0         40.774780
9977        2              1.0          19.0         40.751698
9978        2              1.0          19.0         40.752941
9979        2              1.0          19.0         40.735130
9980        2              1.0          19.0         40.745541
9981        2              1.0          19.0         40.793671
9982        2              1.0          19.0         40.754639
9983        2              1.0          18.0         40.723721
9984        2              1.0          19.0         40.774590
9985        2              1.0          19.0         40.774872
9986        2              1.0          19.0         40.787998
9987        2              1.0          19.0         40.790218
9988        2              1.0          19.0         40.739487
9989        2              1.0          19.0         40.780548
9990        2              1.0          19.0         40.761524
9991        2              1.0          19.0         40.720646
9992        2              1.0          19.0         40.795898
9993        2              1.0          18.0         40.769939
9994        2              4.0          18.0         40.773521
9995        2              4.0          18.0         40.774670
9996        2              4.0          18.0         40.758148
9997        2              4.0          18.0         40.768131
9998        2              4.0          18.0         40.759171
9999        2              4.0          18.0         40.752113

      dropoff_longitude  extra  fare_amount  improvement_surcharge  mta_tax  \
0            -73.974785    1.0         12.0                    0.3      0.5
1            -73.994415    0.5         14.5                    0.3      0.5
2            -73.951820    0.5          9.5                    0.3      0.5
3            -74.004326    0.5          3.5                    0.3      0.5
4            -74.004181    0.5         15.0                    0.3      0.5
5            -73.986977    0.5         27.0                    0.3      0.5
6            -73.992470    0.5         14.0                    0.3      0.5
7            -73.995010    0.5          7.0                    0.3      0.5
8            -73.987595    0.0         52.0                    0.3      0.5
9            -73.985916    0.5          6.5                    0.3      0.5
10           -74.004395    0.5          7.0                    0.3      0.5
11           -73.967407    0.5          7.5                    0.3      0.5
12           -74.009773    0.5          3.0                    0.3      0.5
13           -73.997345    0.5         19.0                    0.3      0.5
14           -73.995216    0.5          6.0                    0.3      0.5
15           -73.983978    1.0         16.5                    0.3      0.5
16           -73.955124    1.0         12.5                    0.3      0.5
17           -73.952713    1.0         26.0                    0.3      0.5
18           -73.980850    1.0         11.5                    0.3      0.5
```

(continues on next page)

(continued from previous page)

```
19          -73.952377   1.0         21.5                    0.3      0.5
20          -73.998924   1.0         17.5                    0.3      0.5
21          -73.999939   1.0          5.5                    0.3      0.5
22          -74.003563   1.0          5.5                    0.3      0.5
23          -74.007919   1.0          6.5                    0.3      0.5
24          -73.978172   1.0         11.5                    0.3      0.5
25          -73.953339   1.0          7.5                    0.3      0.5
26          -73.988609   1.0          9.0                    0.3      0.5
27          -73.985603   0.0         52.0                    0.3      0.5
28          -73.994514   1.0         10.0                    0.3      0.5
29          -74.009079   1.0         17.5                    0.3      0.5
...                ...   ...          ...                    ...      ...
9970        -73.955521   0.0         20.0                    0.3      0.5
9971        -73.984940   1.0          6.5                    0.3      0.5
9972        -74.002457   0.0          8.5                    0.3      0.5
9973        -73.969666   1.0         24.5                    0.3      0.5
9974        -73.960800   1.0          5.5                    0.3      0.5
9975        -73.984154   1.0          9.0                    0.3      0.5
9976        -73.957779   1.0         20.0                    0.3      0.5
9977        -73.989746   1.0          8.5                    0.3      0.5
9978        -73.977470   1.0          7.5                    0.3      0.5
9979        -73.976120   1.0          8.5                    0.3      0.5
9980        -73.984383   1.0          8.5                    0.3      0.5
9981        -73.974327   1.0          5.0                    0.3      0.5
9982        -73.986343   1.0         11.0                    0.3      0.5
9983        -73.989494   1.0          4.5                    0.3      0.5
9984        -73.963249   1.0          5.5                    0.3      0.5
9985        -73.982613   1.0          7.0                    0.3      0.5
9986        -73.953888   1.0          5.0                    0.3      0.5
9987        -73.975128   1.0         11.5                    0.3      0.5
9988        -73.989059   1.0          9.5                    0.3      0.5
9989        -73.959030   1.0          8.5                    0.3      0.5
9990        -73.960602   1.0         15.0                    0.3      0.5
9991        -73.989716   1.0          8.0                    0.3      0.5
9992        -73.972610   1.0         20.5                    0.3      0.5
9993        -73.981316   1.0          4.5                    0.3      0.5
9994        -73.955353   1.0         31.0                    0.3      0.5
9995        -73.947845   1.0         11.5                    0.3      0.5
9996        -73.985626   1.0          8.5                    0.3      0.5
9997        -73.964516   1.0         10.5                    0.3      0.5
9998        -73.975189   1.0          6.5                    0.3      0.5
9999        -73.975189   1.0          5.0                    0.3      0.5

      passenger_count      ...       pickup_dayofweek  pickup_hour  \
0                   1      ...                    3.0         19.0
1                   1      ...                    5.0         20.0
2                   1      ...                    5.0         20.0
3                   1      ...                    5.0         20.0
4                   1      ...                    5.0         20.0
5                   1      ...                    5.0         20.0
6                   1      ...                    5.0         20.0
7                   3      ...                    5.0         20.0
8                   3      ...                    5.0         20.0
9                   2      ...                    5.0         20.0
10                  1      ...                    5.0         20.0
11                  1      ...                    5.0         20.0
12                  1      ...                    5.0         20.0
```

(continues on next page)

(continued from previous page)

```
13                 1      ...                    5.0         20.0
14                 1      ...                    5.0         20.0
15                 1      ...                    3.0         19.0
16                 5      ...                    3.0         19.0
17                 5      ...                    3.0         19.0
18                 1      ...                    3.0         19.0
19                 2      ...                    3.0         19.0
20                 1      ...                    3.0         19.0
21                 1      ...                    3.0         19.0
22                 1      ...                    3.0         19.0
23                 2      ...                    3.0         19.0
24                 5      ...                    3.0         19.0
25                 5      ...                    3.0         19.0
26                 1      ...                    3.0         19.0
27                 1      ...                    3.0         19.0
28                 1      ...                    3.0         19.0
29                 6      ...                    3.0         19.0
...              ...      ...                    ...          ...
9970               1      ...                    4.0         10.0
9971               1      ...                    4.0         10.0
9972               2      ...                    4.0         10.0
9973               1      ...                    1.0         18.0
9974               5      ...                    1.0         18.0
9975               1      ...                    1.0         18.0
9976               3      ...                    1.0         18.0
9977               2      ...                    1.0         18.0
9978               1      ...                    1.0         18.0
9979               1      ...                    1.0         18.0
9980               1      ...                    1.0         18.0
9981               2      ...                    1.0         18.0
9982               1      ...                    1.0         18.0
9983               1      ...                    1.0         18.0
9984               5      ...                    1.0         18.0
9985               1      ...                    1.0         18.0
9986               2      ...                    1.0         18.0
9987               1      ...                    1.0         18.0
9988               1      ...                    1.0         18.0
9989               1      ...                    1.0         18.0
9990               1      ...                    1.0         18.0
9991               1      ...                    1.0         18.0
9992               1      ...                    1.0         18.0
9993               1      ...                    1.0         18.0
9994               1      ...                    4.0         18.0
9995               1      ...                    4.0         18.0
9996               2      ...                    4.0         18.0
9997               1      ...                    4.0         18.0
9998               3      ...                    4.0         18.0
9999               1      ...                    4.0         18.0

      pickup_latitude  pickup_longitude  tip_amount  tolls_amount  \
0           40.750111        -73.993896        3.25          0.00
1           40.724243        -74.001648        2.00          0.00
2           40.802788        -73.963341        0.00          0.00
3           40.713818        -74.009087        0.00          0.00
4           40.762428        -73.971176        0.00          0.00
5           40.774048        -73.874374        6.70          5.33
6           40.726009        -73.983276        0.00          0.00
```

(continues on next page)

(continued from previous page)

```
7         40.734142      -74.002663        1.66         0.00
8         40.644356      -73.783043        0.00         5.33
9         40.767948      -73.985588        1.55         0.00
10        40.723103      -73.988617        1.66         0.00
11        40.751419      -73.993782        1.00         0.00
12        40.704376      -74.008362        0.00         0.00
13        40.760448      -73.973946        3.00         0.00
14        40.731777      -74.006721        0.00         0.00
15        40.739811      -73.976425        4.38         0.00
16        40.754246      -73.968704        0.00         0.00
17        40.769581      -73.863060        8.08         5.33
18        40.779423      -73.945541        0.00         0.00
19        40.774010      -73.874458        4.50         0.00
20        40.751896      -73.976601        0.00         0.00
21        40.745079      -73.994957        1.62         0.00
22        40.747063      -74.000938        1.30         0.00
23        40.717892      -74.002777        1.50         0.00
24        40.736362      -73.997459        2.50         0.00
25        40.823994      -73.952278        1.70         0.00
26        40.750080      -73.991127        0.00         0.00
27        40.644127      -73.786575        6.00         5.33
28        40.741447      -73.993668        2.36         0.00
29        40.744083      -73.985291        3.70         0.00
...             ...             ...         ...          ...
9970      40.725979      -74.009071        4.00         0.00
9971      40.732452      -73.985001        1.65         0.00
9972      40.751358      -73.990479        1.00         0.00
9973      40.708790      -74.017281        5.10         0.00
9974      40.780003      -73.954681        1.00         0.00
9975      40.749680      -73.991531        0.00         0.00
9976      40.751801      -74.002327        2.00         0.00
9977      40.768433      -73.986137        0.00         0.00
9978      40.745071      -73.987068        1.00         0.00
9979      40.751259      -73.977814        0.00         0.00
9980      40.731110      -74.001350        0.00         0.00
9981      40.791222      -73.965118        0.00         0.00
9982      40.764175      -73.968994        1.00         0.00
9983      40.714985      -73.992409        2.00         0.00
9984      40.764881      -73.968529        1.30         0.00
9985      40.762344      -73.985695        1.60         0.00
9986      40.779526      -73.957619        1.20         0.00
9987      40.762226      -73.985916        2.50         0.00
9988      40.725056      -73.984329        2.10         0.00
9989      40.778542      -73.981949        1.00         0.00
9990      40.746319      -74.001114        0.00         0.00
9991      40.738167      -73.987434        1.00         0.00
9992      40.740582      -73.989738        4.30         0.00
9993      40.772015      -73.979416        1.10         0.00
9994      40.713215      -74.013542        5.00         0.00
9995      40.773186      -73.978043        0.00         0.00
9996      40.752003      -73.973198        0.00         0.00
9997      40.740456      -73.986252        2.46         0.00
9998      40.770500      -73.981323        2.08         0.00
9999      40.761505      -73.968452        0.00         0.00

      total_amount  tpep_dropoff_datetime tpep_pickup_datetime trip_distance
0            17.05    2015-01-15 19:23:42  2015-01-15 19:05:39          1.59
```

(continues on next page)

(continued from previous page)

```
1          17.80   2015-01-10 20:53:28  2015-01-10 20:33:38         3.30
2          10.80   2015-01-10 20:43:41  2015-01-10 20:33:38         1.80
3           4.80   2015-01-10 20:35:31  2015-01-10 20:33:39         0.50
4          16.30   2015-01-10 20:52:58  2015-01-10 20:33:39         3.00
5          40.33   2015-01-10 20:53:52  2015-01-10 20:33:39         9.00
6          15.30   2015-01-10 20:58:31  2015-01-10 20:33:39         2.20
7           9.96   2015-01-10 20:42:20  2015-01-10 20:33:39         0.80
8          58.13   2015-01-10 21:11:35  2015-01-10 20:33:39        18.20
9           9.35   2015-01-10 20:40:44  2015-01-10 20:33:40         0.90
10          9.96   2015-01-10 20:41:39  2015-01-10 20:33:40         0.90
11          9.80   2015-01-10 20:43:26  2015-01-10 20:33:41         1.10
12          4.30   2015-01-10 20:35:23  2015-01-10 20:33:41         0.30
13         23.30   2015-01-10 21:03:04  2015-01-10 20:33:41         3.10
14          7.30   2015-01-10 20:39:23  2015-01-10 20:33:41         1.10
15         22.68   2015-01-15 19:32:00  2015-01-15 19:05:39         2.38
16         14.30   2015-01-15 19:21:00  2015-01-15 19:05:40         2.83
17         41.21   2015-01-15 19:28:18  2015-01-15 19:05:40         8.33
18         13.30   2015-01-15 19:20:36  2015-01-15 19:05:41         2.37
19         27.80   2015-01-15 19:20:22  2015-01-15 19:05:41         7.13
20         19.30   2015-01-15 19:31:00  2015-01-15 19:05:41         3.60
21          8.92   2015-01-15 19:10:22  2015-01-15 19:05:41         0.89
22          8.60   2015-01-15 19:10:55  2015-01-15 19:05:41         0.96
23          9.80   2015-01-15 19:12:36  2015-01-15 19:05:41         1.25
24         15.80   2015-01-15 19:22:11  2015-01-15 19:05:41         2.11
25         11.00   2015-01-15 19:14:05  2015-01-15 19:05:41         1.15
26         10.80   2015-01-15 19:16:18  2015-01-15 19:05:42         1.53
27         64.13   2015-01-15 19:49:07  2015-01-15 19:05:42        18.06
28         14.16   2015-01-15 19:18:33  2015-01-15 19:05:42         1.76
29         23.00   2015-01-15 19:21:40  2015-01-15 19:05:42         5.19
...          ...                  ...                  ...          ...
9970       24.80   2015-01-30 11:20:08  2015-01-30 10:51:40         3.70
9971        9.95   2015-01-30 10:58:58  2015-01-30 10:51:40         1.10
9972       10.30   2015-01-30 11:03:41  2015-01-30 10:51:41         0.70
9973       31.40   2015-01-13 19:22:18  2015-01-13 18:55:41         7.08
9974        8.30   2015-01-13 19:02:03  2015-01-13 18:55:41         0.64
9975       10.80   2015-01-13 19:06:56  2015-01-13 18:55:41         1.67
9976       23.80   2015-01-13 19:18:39  2015-01-13 18:55:42         5.28
9977       10.30   2015-01-13 19:06:38  2015-01-13 18:55:42         1.38
9978       10.30   2015-01-13 19:05:34  2015-01-13 18:55:42         0.88
9979       10.30   2015-01-13 19:05:41  2015-01-13 18:55:42         1.58
9980       10.30   2015-01-13 19:05:32  2015-01-13 18:55:42         1.58
9981        6.80   2015-01-13 19:00:05  2015-01-13 18:55:42         0.63
9982       13.80   2015-01-13 19:11:57  2015-01-13 18:55:43         1.63
9983        8.30   2015-01-13 18:59:19  2015-01-13 18:55:43         0.70
9984        8.60   2015-01-13 19:01:19  2015-01-13 18:55:44         0.94
9985       10.40   2015-01-13 19:03:54  2015-01-13 18:55:44         1.04
9986        8.00   2015-01-13 19:00:06  2015-01-13 18:55:44         0.74
9987       15.80   2015-01-13 19:10:46  2015-01-13 18:55:44         2.19
9988       13.40   2015-01-13 19:08:40  2015-01-13 18:55:44         1.48
9989       11.30   2015-01-13 19:04:44  2015-01-13 18:55:45         1.83
9990       16.80   2015-01-13 19:14:59  2015-01-13 18:55:45         3.27
9991       10.80   2015-01-13 19:04:58  2015-01-13 18:55:45         1.56
9992       26.60   2015-01-13 19:18:18  2015-01-13 18:55:45         5.40
9993        7.40   2015-01-13 18:59:40  2015-01-13 18:55:45         0.34
9994       37.80   2015-01-23 18:59:52  2015-01-23 18:22:55         9.05
9995       13.30   2015-01-23 18:37:44  2015-01-23 18:22:55         2.32
9996       10.30   2015-01-23 18:34:48  2015-01-23 18:22:56         0.92
```

(continues on next page)

(continued from previous page)

```
9997          14.76    2015-01-23 18:33:58  2015-01-23 18:22:56          2.36
9998          10.38    2015-01-23 18:29:22  2015-01-23 18:22:56          1.05
9999           6.80    2015-01-23 18:27:58  2015-01-23 18:22:57          0.75

[10000 rows x 21 columns]
```

5.1.8 Tutorial

If you want to learn more on vaex, take a look at the tutorials to see what is possible.

5.2 Dask

5.2.1 Dask.array

A vaex dataframe can be lazily converted to a dask.array using *DataFrame.to_dask_array*.

[2]:
```
import vaex
df = vaex.example()
df
```

[2]:
```
#        x             y             z             vx            vy            vz      ␣
↪    E                 L                   Lz                    FeH
0        -0.777470767  2.10626292    1.93743467    53.276722     288.386047    -95.
↪2649078  -121238.171875   831.0799560546875   -336.426513671875    -2.
↪309227609164518
1        3.77427316    2.23387194    3.76209331    252.810791    -69.9498444   -56.
↪3121033  -100819.9140625  1435.1839599609375  -828.7567749023438   -1.
↪788735491591229
2        1.3757627     -6.3283844    2.63250017    96.276474     226.440201    -34.
↪7527161  -100559.9609375  1039.2989501953125  920.802490234375     -0.
↪7618109022478798
3        -7.06737804   1.31737781    -6.10543537   204.968842    -205.679016   -58.
↪9777031  -70174.8515625   2441.724853515625   1183.5899658203125   -1.
↪5208778422936413
4        0.243441463   -0.822781682  -0.206593871  -311.742371   -238.41217    186.
↪824127   -144138.75       374.8164367675781   -314.5353088378906   -2.
↪655341358427361
...      ...           ...           ...           ...           ...           ...     ␣
↪    ...               ...                 ...                   ...
329,995  3.76883793    4.66251659    -4.42904139   107.432999    -2.13771296   17.
↪5130272   -119687.3203125  746.8833618164062   -508.96484375        -1.
↪6499842518381402
329,996  9.17409325    -8.87091351   -8.61707687   32.0          108.089264    179.
↪060638   -68933.8046875   2395.633056640625   1275.490234375       -1.
↪4336036247720836
329,997  -1.14041007   -8.4957695    2.25749826    8.46711349    -38.2765236   -127.
↪541473  -112580.359375   1182.436279296875  115.58557891845703   -1.
↪9306227597361942
329,998  -14.2985935   -5.51750422   -8.65472317   110.221558    -31.3925591   86.
↪2726822   -74862.90625     1324.5926513671875  1057.017333984375    -1.
↪225019818838568
329,999  10.5450506    -8.86106777   -4.65835428   -2.10541415   -27.6108856   3.
↪80799961   -95361.765625    351.0955505371094   -309.81439208984375  -2.
↪5689636894079477
```

(continues on next page)

(continued from previous page)

```
[10]: # convert a set of columns in the dataframe to a 2d dask array
      A = df[['x', 'y', 'z']].to_dask_array()
      A
```

```
[10]: dask.array<vaex-df-d741baee-10eb-11ea-b19a, shape=(330000, 3), dtype=float64,␣
      ↪chunksize=(330000, 3), chunktype=numpy.ndarray>
```

```
[11]: import dask.array as da
      # lazily compute with dask
      r = da.sqrt(A[:,0]**2 + A[:,1]**2 + A[:,2]**2)
      r
```

```
[11]: dask.array<sqrt, shape=(330000,), dtype=float64, chunksize=(330000,), chunktype=numpy.
      ↪ndarray>
```

```
[12]: # materialize the data
      r_computed = r.compute()
      r_computed
```

```
[15]: # put it back in the dataframe
      df['r'] = r_computed
      df
```

```
[15]: #         x             y             z             vx            vy            vz          ␣
      ↪     E                   L                   Lz                  FeH                    r
      0         -0.777470767  2.10626292    1.93743467    53.276722     288.386047    -95.
      ↪2649078  -121238.171875   831.0799560546875   -336.426513671875    -2.
      ↪309227609164518   2.9655450396553587
      1         3.77427316    2.23387194    3.76209331    252.810791    -69.9498444   -56.
      ↪3121033  -100819.9140625  1435.1839599609375  -828.7567749023438   -1.
      ↪788735491591229   5.77829281049018
      2         1.3757627     -6.3283844    2.63250017    96.276474     226.440201    -34.
      ↪7527161  -100559.9609375  1039.2989501953125  920.802490234375     -0.
      ↪7618109022478798  6.99079603950256
      3         -7.06737804   1.31737781    -6.10543537   204.968842    -205.679016   -58.
      ↪9777031  -70174.8515625   2441.724853515625   1183.5899658203125   -1.
      ↪5208778422936413  9.431842752707537
      4         0.243441463   -0.822781682  -0.206593871  -311.742371   -238.41217    186.
      ↪824127   -144138.75       374.8164367675781   -314.5353088378906   -2.
      ↪655341358427361   0.8825613121347967
      ...       ...           ...           ...           ...           ...           ...         ␣
      ↪     ...                 ...                 ...                 ...                   ␣
      ↪...
      329,995   3.76883793    4.66251659    -4.42904139   107.432999    -2.13771296   17.
      ↪5130272   -119687.3203125  746.8833618164062   -508.96484375        -1.
      ↪6499842518381402  7.453831761514681
      329,996   9.17409325    -8.87091351   -8.61707687   32.0          108.089264    179.
      ↪060638   -68933.8046875   2395.633056640625   1275.490234375       -1.
      ↪4336036247720836  15.398412491068198
      329,997   -1.14041007   -8.4957695    2.25749826    8.46711349    -38.2765236   -127.
      ↪541473  -112580.359375  1182.436279296875   115.58557891845703   -1.
      ↪9306227597361942  8.864250273925633
      329,998   -14.2985935   -5.51750422   -8.65472317   110.221558    -31.3925591   86.
      ↪2726822   -74862.90625     1324.5926513671875  1057.017333984375    -1.
      ↪225019818838568   17.601047186042507
```

(continues on next page)

(continued from previous page)

```
329,999  10.5450506    -8.86107777   -4.65835428   -2.10541415  -27.6108856  3.
↪80799961   -95361.765625    351.0955505371094   -309.81439208984375  -2.
↪5689636894079477  14.540181524970293
```

[ ]:

```

```

5.3 GraphQL

vaex-graphql is a plugin package that exposes a DataFrame via a GraphQL interface. This allows easy sharing of data or aggregations/statistics or machine learning models to frontends or other programs with a standard query languages.

(Install with `$ pip install vaex-graphql`, no conda-forge support yet)

[3]:

```
import vaex.ml
df = vaex.ml.datasets.load_titanic()
df
```

[3]:

```
#      pclass    survived    name                                      sex    ␣
↪age    sibsp    parch    ticket    fare      cabin    embarked    boat    body   ␣
↪home_dest
0      1         True        Allen, Miss. Elisabeth Walton              female ␣
↪29.0    0        0       24160     211.3375  B5       S           2       nan    ␣
↪St Louis, MO
1      1         True        Allison, Master. Hudson Trevor             male   ␣
↪0.9167  1        2       113781    151.55    C22 C26  S           11      nan    ␣
↪Montreal, PQ / Chesterville, ON
2      1         False       Allison, Miss. Helen Loraine               female ␣
↪2.0     1        2       113781    151.55    C22 C26  S           None    nan    ␣
↪Montreal, PQ / Chesterville, ON
3      1         False       Allison, Mr. Hudson Joshua Creighton       male   ␣
↪30.0    1        2       113781    151.55    C22 C26  S           None    135.0  ␣
↪Montreal, PQ / Chesterville, ON
4      1         False       Allison, Mrs. Hudson J C (Bessie Waldo Daniels)  female ␣
↪25.0    1        2       113781    151.55    C22 C26  S           None    nan    ␣
↪Montreal, PQ / Chesterville, ON
...    ...       ...         ...                                        ...    ␣
↪...     ...      ...     ...       ...       ...      ...         ...     ...    ␣
↪...
1,304  3         False       Zabour, Miss. Hileni                       female ␣
↪14.5    1        0       2665      14.4542   None     C           None    328.0  ␣
↪None
1,305  3         False       Zabour, Miss. Thamine                      female ␣
↪nan     1        0       2665      14.4542   None     C           None    nan    ␣
↪None
1,306  3         False       Zakarian, Mr. Mapriededer                  male   ␣
↪26.5    0        0       2656      7.225     None     C           None    304.0  ␣
↪None
1,307  3         False       Zakarian, Mr. Ortin                        male   ␣
↪27.0    0        0       2670      7.225     None     C           None    nan    ␣
↪None
1,308  3         False       Zimmerman, Mr. Leo                         male   ␣
↪29.0    0        0       315082    7.875     None     S           None    nan    ␣
↪None
```

```
[10]: result = df.graphql.execute("""
          {
              df {
                  min {
                      age
                      fare
                  }
                  mean {
                      age
                      fare
                  }
                  max {
                      age
                      fare
                  }
                  groupby {
                      sex {
                         count
                         mean {
                             age
                         }
                      }
                  }
              }
          }
          """)
      result.data
```

```
[10]: OrderedDict([('df',
                    OrderedDict([('min',
                                  OrderedDict([('age', 0.1667), ('fare', 0.0)])),
                                 ('mean',
                                  OrderedDict([('age', 29.8811345124283),
                                               ('fare', 33.29547928134572)])),
                                 ('max',
                                  OrderedDict([('age', 80.0), ('fare', 512.3292)])),
                                 ('groupby',
                                  OrderedDict([('sex',
                                                OrderedDict([('count', [466, 843]),
                                                             ('mean',
                                                              OrderedDict([('age',
                                                                            [28.
       ↪6870706185567,
                                                                             30.
       ↪585232978723408])]))]))]))]))])
```

5.3.1 Pandas support

After importing vaex.graphql, vaex also installs a pandas accessor, so it is also accessible for Pandas DataFrames.

```
[11]: df_pandas = df.to_pandas_df()
```

```
[20]: df_pandas.graphql.execute("""
          {
              df(where: {age: {_gt: 20}}) {
                  row(offset: 3, limit: 2) {
```

(continues on next page)

(continued from previous page)

```
                name
                survived
            }
        }
    }
    """
).data
```

[20]:
```
OrderedDict([('df',
              OrderedDict([('row',
                            [OrderedDict([('name', 'Anderson, Mr. Harry'),
                                          ('survived', True)]),
                             OrderedDict([('name',
                                           'Andrews, Miss. Kornelia Theodosia'),
                                          ('survived', True)])])]))])
```

5.3.2 Server

The easiest way to learn to use the GraphQL language/vaex interface is to launch a server, and play with the GraphiQL graphical interface, its autocomplete, and the schema explorer.

We try to stay close to the Hasura API: https://docs.hasura.io/1.0/graphql/manual/api-reference/graphql-api/query.html

A server can be started from the command line:

```
$ python -m vaex.graphql myfile.hdf5
```

Or from within Python using df.graphql.serve

5.3.3 GraphiQL

See https://github.com/mariobuikhuizen/ipygraphql for a graphical widget, or a mybinder to try out a live example.

[ ]:
```

```

CHAPTER 6

API documentation for vaex library

6.1 Quick lists

6.1.1 Opening/reading in your data.

`vaex.open`(path[, convert, shuffle, copy_index])	Open a DataFrame from file given by path.
`vaex.from_arrow_table`(table)	Creates a vaex DataFrame from an arrow Table.
`vaex.from_arrays`(**arrays)	Create an in memory DataFrame from numpy arrays.
`vaex.from_dict`(data)	Create an in memory dataset from a dict with column names as keys and list/numpy-arrays as values
`vaex.from_csv`(filename_or_buffer[, copy_index])	Shortcut to read a csv file using pandas and convert to a DataFrame directly.
`vaex.from_ascii`(path[, seperator, names, . . .])	Create an in memory DataFrame from an ascii file (whitespace seperated by default).
`vaex.from_pandas`(df[, name, copy_index, . . .])	Create an in memory DataFrame from a pandas DataFrame.
`vaex.from_astropy_table`(table)	Create a vaex DataFrame from an Astropy Table.

6.1.2 Visualization.

`vaex.dataframe.DataFrame.plot`([x, y, z, . . .])	Viz data in a 2d histogram/heatmap.
`vaex.dataframe.DataFrame.plot1d`([x, what, . . .])	Viz data in 1d (histograms, running means etc)
`vaex.dataframe.DataFrame.scatter`(x, y[, . . .])	Viz (small amounts) of data in 2d using a scatter plot
`vaex.dataframe.DataFrame.plot_widget`(x, y[, . . .])	Viz 1d, 2d or 3d in a Jupyter notebook

Continued on next page

Table 2 – continued from previous page

vaex.dataframe.DataFrame.healpix_plot([. . .])	Viz data in 2d using a healpix column.

6.1.3 Statistics.

vaex.dataframe.DataFrame.count([expression, . . .])	Count the number of non-NaN values (or all, if expression is None or "*").
vaex.dataframe.DataFrame.mean(expression[, . . .])	Calculate the mean for expression, possibly on a grid defined by binby.
vaex.dataframe.DataFrame.std(expression[, . . .])	Calculate the standard deviation for the given expression, possible on a grid defined by binby
vaex.dataframe.DataFrame.var(expression[, . . .])	Calculate the sample variance for the given expression, possible on a grid defined by binby
vaex.dataframe.DataFrame.cov(x[, y, binby, . . .])	Calculate the covariance matrix for x and y or more expressions, possibly on a grid defined by binby.
vaex.dataframe.DataFrame.correlation(x[, y, . . .])	Calculate the correlation coefficient cov[x,y]/(std[x]*std[y]) between and x and y, possibly on a grid defined by binby.
vaex.dataframe.DataFrame.median_approx(. . .)	Calculate the median , possibly on a grid defined by binby.
vaex.dataframe.DataFrame.mode(expression[, . . .])	Calculate/estimate the mode.
vaex.dataframe.DataFrame.min(expression[, . . .])	Calculate the minimum for given expressions, possibly on a grid defined by binby.
vaex.dataframe.DataFrame.max(expression[, . . .])	Calculate the maximum for given expressions, possibly on a grid defined by binby.
vaex.dataframe.DataFrame.minmax(expression)	Calculate the minimum and maximum for expressions, possibly on a grid defined by binby.
vaex.dataframe.DataFrame.mutual_information(x)	Estimate the mutual information between and x and y on a grid with shape mi_shape and mi_limits, possibly on a grid defined by binby.

6.2 vaex-core

Vaex is a library for dealing with larger than memory DataFrames (out of core).

The most important class (datastructure) in vaex is the `DataFrame`. A DataFrame is obtained by either, opening the example dataset:

```
>>> import vaex
>>> df = vaex.example()
```

Or using `open()` to open a file.

```
>>> df1 = vaex.open("somedata.hdf5")
>>> df2 = vaex.open("somedata.fits")
>>> df2 = vaex.open("somedata.arrow")
>>> df4 = vaex.open("somedata.csv")
```

Or connecting to a remove server:

```
>>> df_remote = vaex.open("http://try.vaex.io/nyc_taxi_2015")
```

A few strong features of vaex are:

- Performance: Works with huge tabular data, process over a billion (> 10:sup:*9*) rows/second.
- Expression system / Virtual columns: compute on the fly, without wasting ram.
- Memory efficient: no memory copies when doing filtering/selections/subsets.
- Visualization: directly supported, a one-liner is often enough.
- User friendly API: You will only need to deal with a DataFrame object, and tab completion + docstring will help you out: *ds.mean<tab>*, feels very similar to Pandas.
- Very fast statiscs on N dimensional grids such as histograms, running mean, heatmaps.

Follow the tutorial at https://docs.vaex.io/en/latest/tutorial.html to learn how to use vaex.

`vaex.open`(*path*, *convert=False*, *shuffle=False*, *copy_index=True*, **args*, ***kwargs*)
Open a DataFrame from file given by path.

Example:

```
>>> df = vaex.open('sometable.hdf5')
>>> df = vaex.open('somedata*.csv', convert='bigdata.hdf5')
```

Parameters

- **or list path** (*str*) – local or absolute path to file, or glob string, or list of paths
- **convert** – convert files to an hdf5 file for optimization, can also be a path
- **shuffle** (*bool*) – shuffle converted DataFrame or not
- **args** – extra arguments for file readers that need it
- **kwargs** – extra keyword arguments
- **copy_index** (*bool*) – copy index when source is read via pandas

Returns return a DataFrame on succes, otherwise None

Return type *DataFrame*

S3 support:

Vaex supports streaming in hdf5 files from Amazon AWS object storage S3. Files are by default cached in $HOME/.vaex/file-cache/s3 such that successive access it as fast as native disk access. The following url parameters control S3 options:

- anon: Use anonymous access or not (false by default). (Allowed values are: true,True,1,false,False,0)
- use_cache: Use the disk cache or not, only set to false if the data should be accessed once. (Allowed values are: true,True,1,false,False,0)
- profile_name and other arguments are passed to `s3fs.core.S3FileSystem`

All arguments can also be passed as kwargs, but then arguments such as *anon* can only be a boolean, not a string.

Examples:

```
>>> df = vaex.open('s3://vaex/taxi/yellow_taxi_2015_f32s.hdf5?anon=true')
>>> df = vaex.open('s3://vaex/taxi/yellow_taxi_2015_f32s.hdf5', anon=True)  #␣
↪Note that anon is a boolean, not the string 'true'
>>> df = vaex.open('s3://mybucket/path/to/file.hdf5?profile_name=myprofile')
```

`vaex.from_arrays(**arrays)`

Create an in memory DataFrame from numpy arrays.

Example

```
>>> import vaex, numpy as np
>>> x = np.arange(5)
>>> y = x ** 2
>>> vaex.from_arrays(x=x, y=y)
  #    x    y
  0    0    0
  1    1    1
  2    2    4
  3    3    9
  4    4   16
>>> some_dict = {'x': x, 'y': y}
>>> vaex.from_arrays(**some_dict)  # in case you have your columns in a dict
  #    x    y
  0    0    0
  1    1    1
  2    2    4
  3    3    9
  4    4   16
```

Parameters **arrays** – keyword arguments with arrays

Return type *DataFrame*

`vaex.from_dict(data)`

Create an in memory dataset from a dict with column names as keys and list/numpy-arrays as values

Example

```
>>> data = {'A':[1,2,3],'B':['a','b','c']}
>>> vaex.from_dict(data)
  #    A    B
  0    1   'a'
  1    2   'b'
  2    3   'c'
```

Parameters **data** – A dict of {columns:[value, value,. . .]}

Return type *DataFrame*

`vaex.from_items(*items)`

Create an in memory DataFrame from numpy arrays, in contrast to from_arrays this keeps the order of columns intact (for Python < 3.6).

Example

```
>>> import vaex, numpy as np
>>> x = np.arange(5)
```

(continues on next page)

(continued from previous page)

```
>>> y = x ** 2
>>> vaex.from_items(('x', x), ('y', y))
  #    x    y
  0    0    0
  1    1    1
  2    2    4
  3    3    9
  4    4   16
```

Parameters **items** – list of [(name, numpy array), . . .]

Return type *DataFrame*

`vaex.from_arrow_table`(*table*)

Creates a vaex DataFrame from an arrow Table.

Return type *DataFrame*

`vaex.from_csv`(*filename_or_buffer*, *copy_index=True*, ***kwargs*)

Shortcut to read a csv file using pandas and convert to a DataFrame directly.

Return type *DataFrame*

`vaex.from_ascii`(*path*, *seperator=None*, *names=True*, *skip_lines=0*, *skip_after=0*, ***kwargs*)

Create an in memory DataFrame from an ascii file (whitespace seperated by default).

```
>>> ds = vx.from_ascii("table.asc")
>>> ds = vx.from_ascii("table.csv", seperator=",", names=["x", "y", "z"])
```

Parameters

- **path** – file path
- **seperator** – value seperator, by default whitespace, use "," for comma seperated values.
- **names** – If True, the first line is used for the column names, otherwise provide a list of strings with names
- **skip_lines** – skip lines at the start of the file
- **skip_after** – skip lines at the end of the file
- **kwargs** –

Return type *DataFrame*

`vaex.from_pandas`(*df*, *name='pandas'*, *copy_index=True*, *index_name='index'*)

Create an in memory DataFrame from a pandas DataFrame.

Param pandas.DataFrame df: Pandas DataFrame

Param name: unique for the DataFrame

```
>>> import vaex, pandas as pd
>>> df_pandas = pd.from_csv('test.csv')
>>> df = vaex.from_pandas(df_pandas)
```

Return type *DataFrame*

vaex.**from_astropy_table**(*table*)
Create a vaex DataFrame from an Astropy Table.

vaex.**from_samp**(*username=None*, *password=None*)
Connect to a SAMP Hub and wait for a single table load event, disconnect, download the table and return the DataFrame.

Useful if you want to send a single table from say TOPCAT to vaex in a python console or notebook.

vaex.**open_many**(*filenames*)
Open a list of filenames, and return a DataFrame with all DataFrames cocatenated.

Parameters **filenames** (list[str]) – list of filenames/paths

Return type *DataFrame*

vaex.**register_function**(*scope=None*, *as_property=False*, *name=None*, *on_expression=True*)
Decorator to register a new function with vaex.

If on_expression is True, the function will be available as a method on an Expression, where the first argument will be the expression itself.

Example:

```
>>> import vaex
>>> df = vaex.example()
>>> @vaex.register_function()
>>> def invert(x):
>>>     return 1/x
>>> df.x.invert()
```

```
>>> import numpy as np
>>> df = vaex.from_arrays(departure=np.arange('2015-01-01', '2015-12-05', dtype=
→'datetime64'))
>>> @vaex.register_function(as_property=True, scope='dt')
>>> def dt_relative_day(x):
>>>     return vaex.functions.dt_dayofyear(x)/365.
>>> df.departure.dt.relative_day
```

vaex.**server**(*url*, ***kwargs*)
Connect to hostname supporting the vaex web api.

Parameters **hostname** (str) – hostname or ip address of server

Return vaex.dataframe.ServerRest returns a server object, note that it does not connect to the server yet, so this will always succeed

Return type ServerRest

vaex.**example**(*download=True*)
Returns an example DataFrame which comes with vaex for testing/learning purposes.

Return type *DataFrame*

vaex.**app**(**args*, ***kwargs*)
Create a vaex app, the QApplication mainloop must be started.

In ipython notebook/jupyter do the following:

```
>>> import vaex.ui.main # this causes the qt api level to be set properly
>>> import vaex
```

Next cell:

```
>>> %gui qt
```

Next cell:

```
>>> app = vaex.app()
```

From now on, you can run the app along with jupyter

`vaex.delayed(f)`

Decorator to transparantly accept delayed computation.

Example:

```
>>> delayed_sum = ds.sum(ds.E, binby=ds.x, limits=limits,
>>>                   shape=4, delay=True)
>>> @vaex.delayed
>>> def total_sum(sums):
>>>     return sums.sum()
>>> sum_of_sums = total_sum(delayed_sum)
>>> ds.execute()
>>> sum_of_sums.get()
See the tutorial for a more complete example https://docs.vaex.io/en/latest/
↪tutorial.html#Parallel-computations
```

6.2.1 DataFrame class

class `vaex.dataframe.DataFrame`(*name*, *column_names*, *executor=None*)

Bases: `object`

All local or remote datasets are encapsulated in this class, which provides a pandas like API to your dataset.

Each DataFrame (df) has a number of columns, and a number of rows, the length of the DataFrame.

All DataFrames have multiple 'selection', and all calculations are done on the whole DataFrame (default) or for the selection. The following example shows how to use the selection.

```
>>> df.select("x < 0")
>>> df.sum(df.y, selection=True)
>>> df.sum(df.y, selection=[df.x < 0, df.x > 0])
```

`__delitem__`(*item*)

Removes a (virtual) column from the DataFrame.

Note: this does not remove check if the column is used in a virtual expression or in the filter and may lead to issues. It is safer to use `drop()`.

`__getitem__`(*item*)

Convenient way to get expressions, (shallow) copies of a few columns, or to apply filtering.

Example:

```
>>> df['Lz']  # the expression 'Lz
>>> df['Lz/2'] # the expression 'Lz/2'
>>> df[["Lz", "E"]] # a shallow copy with just two columns
>>> df[df.Lz < 0]  # a shallow copy with the filter Lz < 0 applied
```

__init__(*name*, *column_names*, *executor=None*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
Iterator over the column names.

__len__()
Returns the number of rows in the DataFrame (filtering applied).

__repr__()
Return repr(self).

__setitem__(*name*, *value*)
Convenient way to add a virtual column / expression to this DataFrame.

Example:

```
>>> import vaex, numpy as np
>>> df = vaex.example()
>>> df['r'] = np.sqrt(df.x**2 + df.y**2 + df.z**2)
>>> df.r
<vaex.expression.Expression(expressions='r')> instance at 0x121687e80␣
↪values=[2.9655450396553587, 5.77829281049018, 6.99079603950256, 9.
↪431842752707537, 0.8825613121347967 ... (total 330000 values) ... 7.
↪453831761514681, 15.398412491068198, 8.864250273925633, 17.601047186042507,␣
↪14.540181524970293]
```

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

add_column(*name*, *f_or_array*, *dtype=None*)
Add an in memory array as a column.

add_variable(*name*, *expression*, *overwrite=True*, *unique=True*)
Add a variable to to a DataFrame.

A variable may refer to other variables, and virtual columns and expression may refer to variables.

Example

```
>>> df.add_variable('center', 0)
>>> df.add_virtual_column('x_prime', 'x-center')
>>> df.select('x_prime < 0')
```

Param str name: name of virtual varible

Param expression: expression for the variable

add_virtual_column(*name*, *expression*, *unique=False*)
Add a virtual column to the DataFrame.

Example:

```
>>> df.add_virtual_column("r", "sqrt(x**2 + y**2 + z**2)")
>>> df.select("r < 10")
```

Param str name: name of virtual column

Param expression: expression for the column

Parameters **unique** (*str*) – if name is already used, make it unique by adding a postfix, e.g. _1, or _2

apply (*f*, *arguments=None*, *dtype=None*, *delay=False*, *vectorize=False*)
Apply a function on a per row basis across the entire DataFrame.

Example:

```
>>> import vaex
>>> df = vaex.example()
>>> def func(x, y):
...     return (x+y)/(x-y)
...
>>> df.apply(func, arguments=[df.x, df.y])
Expression = lambda_function(x, y)
Length: 330,000 dtype: float64 (expression)
-------------------------------------------
     0  -0.460789
     1    3.90038
     2  -0.642851
     3   0.685768
     4  -0.543357
```

Parameters

- **f** – The function to be applied
- **arguments** – List of arguments to be passed on the the function f.

Returns A function that is lazily evaluated.

byte_size (*selection=False*, *virtual=False*)
Return the size in bytes the whole DataFrame requires (or the selection), respecting the active_fraction.

cat (*i1*, *i2*, *format='html'*)
Display the DataFrame from row i1 till i2

For format, see https://pypi.org/project/tabulate/

Parameters

- **i1** (*int*) – Start row
- **i2** (*int*) – End row.
- **format** (*str*) – Format to use, e.g. 'html', 'plain', 'latex'

close_files ()
Close any possible open file handles, the DataFrame will not be in a usable state afterwards.

col
Gives direct access to the columns only (useful for tab completion).

Convenient when working with ipython in combination with small DataFrames, since this gives tab-completion.

Columns can be accesed by there names, which are attributes. The attribues are currently expressions, so you can do computations with them.

Example

```
>>> ds = vaex.example()
>>> df.plot(df.col.x, df.col.y)
```

column_count ()
Returns the number of columns (including virtual columns).

combinations (*expressions_list=None*, *dimension=2*, *exclude=None*, ***kwargs*)
Generate a list of combinations for the possible expressions for the given dimension.

Parameters

- **expressions_list** – list of list of expressions, where the inner list defines the subspace
- **dimensions** – if given, generates a subspace with all possible combinations for that dimension
- **exclude** – list of

correlation (*x*, *y=None*, *binby=[]*, *limits=None*, *shape=128*, *sort=False*, *sort_key=<ufunc 'absolute'>*, *selection=False*, *delay=False*, *progress=None*)
Calculate the correlation coefficient cov[x,y]/(std[x]*std[y]) between and x and y, possibly on a grid defined by binby.

Example:

```
>>> df.correlation("x**2+y**2+z**2", "-log(-E+1)")
array(0.6366637382215669)
>>> df.correlation("x**2+y**2+z**2", "-log(-E+1)", binby="Lz", shape=4)
array([ 0.40594394,  0.69868851,  0.61394099,  0.65266318])
```

Parameters

- **x** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **y** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

count (*expression=None*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *edges=False*, *progress=None*)
Count the number of non-NaN values (or all, if expression is None or "*").

Example:

```
>>> df.count()
330000
>>> df.count("*")
330000.0
>>> df.count("*", binby=["x"], shape=4)
array([  10925.,  155427.,  152007.,   10748.])
```

Parameters

- **expression** – Expression or column for which to count non-missing values, or None or '*' for counting the rows
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False
- **edges** – Currently for internal use only (it includes nan's and values outside the limits at borders, nan and 0, smaller than at 1, and larger at -1

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

cov (*x*, *y=None*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Calculate the covariance matrix for x and y or more expressions, possibly on a grid defined by binby.

Either x and y are expressions, e.g:

```
>>> df.cov("x", "y")
```

Or only the x argument is given with a list of expressions, e,g.:

```
>>> df.cov(["x, "y, "z"])
```

Example:

```
>>> df.cov("x", "y")
array([[ 53.54521742,  -3.8123135 ],
[ -3.8123135 ,  60.62257881]])
>>> df.cov(["x", "y", "z"])
array([[ 53.54521742,  -3.8123135 ,  -0.98260511],
[ -3.8123135 ,  60.62257881,   1.21381057],
[ -0.98260511,   1.21381057,  25.55517638]])
```

```
>>> df.cov("x", "y", binby="E", shape=2)
array([[[  9.74852878e+00,  -3.02004780e-02],
[ -3.02004780e-02,   9.99288215e+00]],
```

(continues on next page)

(continued from previous page)

```
[[  8.43996546e+01,  -6.51984181e+00],
[ -6.51984181e+00,   9.68938284e+01]]]])
```

Parameters

- **x** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **y** – if previous argument is not a list, this argument should be given
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimensions are of shape (2,2)

covar (*x*, *y*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Calculate the covariance cov[x,y] between and x and y, possibly on a grid defined by binby.

Example:

```
>>> df.covar("x**2+y**2+z**2", "-log(-E+1)")
array(52.69461456005138)
>>> df.covar("x**2+y**2+z**2", "-log(-E+1)")/(df.std("x**2+y**2+z**2") * df.
↪std("-log(-E+1)"))
0.63666373822156686
>>> df.covar("x**2+y**2+z**2", "-log(-E+1)", binby="Lz", shape=4)
array([ 10.17387143,  51.94954078,  51.24902796,  20.2163929 ])
```

Parameters

- **x** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **y** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

delete_variable (*name*)
Deletes a variable from a DataFrame.

delete_virtual_column (*name*)
Deletes a virtual column from a DataFrame.

describe (*strings=True*, *virtual=True*, *selection=None*)
Give a description of the DataFrame.

```
>>> import vaex
>>> df = vaex.example()[['x', 'y', 'z']]
>>> df.describe()
                 x          y          z
dtype      float64    float64    float64
count       330000     330000     330000
missing          0          0          0
mean    -0.0671315 -0.0535899  0.0169582
std        7.31746    7.78605    5.05521
min       -128.294   -71.5524   -44.3342
max        271.366    146.466    50.7185
>>> df.describe(selection=df.x > 0)
                   x         y          z
dtype        float64   float64    float64
count         164060    164060     164060
missing       165940    165940     165940
mean         5.13572 -0.486786 -0.0868073
std          5.18701   7.61621    5.02831
min      1.51635e-05  -71.5524   -44.3342
max          271.366   78.0724    40.2191
```

Parameters

- **strings** (*bool*) – Describe string columns or not
- **virtual** (*bool*) – Describe virtual columns or not
- **selection** – Optional selection to use.

Returns Pandas dataframe

drop (*columns*, *inplace=False*, *check=True*)
Drop columns (or a single column).

Parameters

- **columns** – List of columns or a single column name
- **inplace** – Make modifications to self or return a new DataFrame
- **check** – When true, it will check if the column is used in virtual columns or the filter, and hide it instead.

drop_filter (*inplace=False*)
Removes all filters from the DataFrame

dropmissing (*column_names=None*)
Create a shallow copy of a DataFrame, with filtering set using ismissing.

Parameters **column_names** – The columns to consider, default: all (real, non-virtual) columns

Return type *DataFrame*

dropna (*column_names=None*)
Create a shallow copy of a DataFrame, with filtering set using isna.

Parameters **column_names** – The columns to consider, default: all (real, non-virtual) columns

Return type *DataFrame*

dropnan (*column_names=None*)
Create a shallow copy of a DataFrame, with filtering set using isnan.

Parameters **column_names** – The columns to consider, default: all (real, non-virtual) columns

Return type *DataFrame*

dtype (*expression, internal=False*)
Return the numpy dtype for the given expression, if not a column, the first row will be evaluated to get the dtype.

dtypes
Gives a Pandas series object containing all numpy dtypes of all columns (except hidden).

evaluate (*expression, i1=None, i2=None, out=None, selection=None, parallel=True*)
Evaluate an expression, and return a numpy array with the results for the full column or a part of it.

Note that this is not how vaex should be used, since it means a copy of the data needs to fit in memory.

To get partial results, use i1 and i2

Parameters

- **expression** (*str*) – Name/expression to evaluate
- **i1** (*int*) – Start row index, default is the start (0)
- **i2** (*int*) – End row index, default is the length of the DataFrame
- **out** (*ndarray*) – Output array, to which the result may be written (may be used to reuse an array, or write to a memory mapped array)
- **selection** – selection to apply

Returns

evaluate_variable (*name*)
Evaluates the variable given by name.

execute ()
Execute all delayed jobs.

extract ()
Return a DataFrame containing only the filtered rows.

Note: Note that no copy of the underlying data is made, only a view/reference is make.

The resulting DataFrame may be more efficient to work with when the original DataFrame is heavily filtered (contains just a small number of rows).

If no filtering is applied, it returns a trimmed view. For the returned df, len(df) == df.length_original() == df.length_unfiltered()

Return type *DataFrame*

fillna(*value*, *fill_nan=True*, *fill_masked=True*, *column_names=None*, *prefix='__original_'*, *inplace=False*)

Return a DataFrame, where missing values/NaN are filled with 'value'.

The original columns will be renamed, and by default they will be hidden columns. No data is lost.

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Note: Note that filtering will be ignored (since they may change), you may want to consider running `extract()` first.

Example:

```
>>> import vaex
>>> import numpy as np
>>> x = np.array([3, 1, np.nan, 10, np.nan])
>>> df = vaex.from_arrays(x=x)
>>> df_filled = df.fillna(value=-1, column_names=['x'])
>>> df_filled
  #    x
  0    3
  1    1
  2   -1
  3   10
  4   -1
```

Parameters

- **value** (*float*) – The value to use for filling nan or masked values.
- **fill_na** (*bool*) – If True, fill np.nan values with *value*.
- **fill_masked** (*bool*) – If True, fill masked values with *values*.
- **column_names** (*list*) – List of column names in which to fill missing values.
- **prefix** (*str*) – The prefix to give the original columns.
- **inplace** – Make modifications to self or return a new DataFrame

first(*expression*, *order_expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *edges=False*, *progress=None*)

Return the first element of a binned *expression*, where the values each bin are sorted by *order_expression*.

Example:

```
>>> import vaex
>>> df = vaex.example()
>>> df.first(df.x, df.y, shape=8)
>>> df.first(df.x, df.y, shape=8, binby=[df.y])
>>> df.first(df.x, df.y, shape=8, binby=[df.y])
array([-4.81883764, 11.65378   ,  9.70084476, -7.3025589 ,  4.84954977,
        8.47446537, -5.73602629, 10.18783   ])
```

Parameters

- **expression** – The value to be placed in the bin.
- **order_expression** – Order the values in the bins by this expression.
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False
- **edges** – Currently for internal use only (it includes nan's and values outside the limits at borders, nan and 0, smaller than at 1, and larger at -1

Returns Ndarray containing the first elements.

Return type numpy.array

get_active_fraction()

Value in the range (0, 1], to work only with a subset of rows.

get_column_names(*virtual=True*, *strings=True*, *hidden=False*, *regex=None*)

Return a list of column names

Example:

```
>>> import vaex
>>> df = vaex.from_scalars(x=1, x2=2, y=3, s='string')
>>> df['r'] = (df.x**2 + df.y**2)**2
>>> df.get_column_names()
['x', 'x2', 'y', 's', 'r']
>>> df.get_column_names(virtual=False)
['x', 'x2', 'y', 's']
>>> df.get_column_names(regex='x.*')
['x', 'x2']
```

Parameters

- **virtual** – If False, skip virtual columns
- **hidden** – If False, skip hidden columns
- **strings** – If False, skip string columns
- **regex** – Only return column names matching the (optional) regular expression

Return type list of str

Example: >>> import vaex >>> df = vaex.from_scalars(x=1, x2=2, y=3, s='string') >>> df['r'] = (df.x**2 + df.y**2)**2 >>> df.get_column_names() ['x', 'x2', 'y', 's', 'r'] >>>

df.get_column_names(virtual=False) ['x', 'x2', 'y', 's'] >>> df.get_column_names(regex='x.*') ['x', 'x2']

get_current_row()
Individual rows can be 'picked', this is the index (integer) of the current row, or None there is nothing picked.

get_private_dir(*create=False*)
Each DataFrame has a directory where files are stored for metadata etc.

Example

```
>>> import vaex
>>> ds = vaex.example()
>>> vaex.get_private_dir()
'/Users/users/breddels/.vaex/dfs/_Users_users_breddels_vaex-testing_data_
↪helmi-dezeeuw-2000-10p.hdf5'
```

Parameters **create** (*bool*) – is True, it will create the directory if it does not exist

get_selection(*name='default'*)
Get the current selection object (mostly for internal use atm).

get_variable(*name*)
Returns the variable given by name, it will not evaluate it.

For evaluation, see `DataFrame.evaluate_variable()`, see also `DataFrame.set_variable()`

has_current_row()
Returns True/False is there currently is a picked row.

has_selection(*name='default'*)
Returns True if there is a selection with the given name.

head(*n=10*)
Return a shallow copy a DataFrame with the first n rows.

head_and_tail_print(*n=5*)
Display the first and last n elements of a DataFrame.

healpix_count(*expression=None*, *healpix_expression=None*, *healpix_max_level=12*, *healpix_level=8*, *binby=None*, *limits=None*, *shape=128*, *delay=False*, *progress=None*, *selection=None*)
Count non missing value for expression on an array which represents healpix data.

Parameters

- **expression** – Expression or column for which to count non-missing values, or None or '*' for counting the rows
- **healpix_expression** – {healpix_max_level}
- **healpix_max_level** – {healpix_max_level}
- **healpix_level** – {healpix_level}
- **binby** – {binby}, these dimension follow the first healpix dimension.
- **limits** – {limits}
- **shape** – {shape}
- **selection** – {selection}

- **delay** – {delay}
- **progress** – {progress}

Returns

healpix_plot(*healpix_expression='source_id/34359738368', healpix_max_level=12, healpix_level=8, what='count(*)', selection=None, grid=None, healpix_input='equatorial', healpix_output='galactic', f=None, colormap='afmhot', grid_limits=None, image_size=800, nest=True, figsize=None, interactive=False, title='', smooth=None, show=False, colorbar=True, rotation=(0, 0, 0), **kwargs*)

Viz data in 2d using a healpix column.

Parameters

- **healpix_expression** – {healpix_max_level}
- **healpix_max_level** – {healpix_max_level}
- **healpix_level** – {healpix_level}
- **what** – {what}
- **selection** – {selection}
- **grid** – {grid}
- **healpix_input** – Specificy if the healpix index is in "equatorial", "galactic" or "ecliptic".
- **healpix_output** – Plot in "equatorial", "galactic" or "ecliptic".
- **f** – function to apply to the data
- **colormap** – matplotlib colormap
- **grid_limits** – Optional sequence [minvalue, maxvalue] that determine the min and max value that map to the colormap (values below and above these are clipped to the the min/max). (default is [min(f(grid)), max(f(grid)))
- **image_size** – size for the image that healpy uses for rendering
- **nest** – If the healpix data is in nested (True) or ring (False)
- **figsize** – If given, modify the matplotlib figure size. Example (14,9)
- **interactive** – (Experimental, uses healpy.mollzoom is True)
- **title** – Title of figure
- **smooth** – apply gaussian smoothing, in degrees
- **show** – Call matplotlib's show (True) or not (False, defaut)
- **rotation** – Rotatate the plot, in format (lon, lat, psi) such that (lon, lat) is the center, and rotate on the screen by angle psi. All angles are degrees.

Returns

is_category(*column*)

Returns true if column is a category.

is_local()

Returns True if the DataFrame is local, False when a DataFrame is remote.

is_masked(*column*)

Return if a column is a masked (numpy.ma) column.

length_original()
the full length of the DataFrame, independent what active_fraction is, or filtering. This is the real length of the underlying ndarrays.

length_unfiltered()
The length of the arrays that should be considered (respecting active range), but without filtering.

limits(*expression*, *value=None*, *square=False*, *selection=None*, *delay=False*, *shape=None*)
Calculate the [min, max] range for expression, as described by value, which is '99.7%' by default.

If value is a list of the form [minvalue, maxvalue], it is simply returned, this is for convenience when using mixed forms.

Example:

```
>>> df.limits("x")
array([-28.86381927,  28.9261226 ])
>>> df.limits(["x", "y"])
(array([-28.86381927,  28.9261226 ]), array([-28.60476934,  28.96535249]))
>>> df.limits(["x", "y"], "minmax")
(array([-128.293991,  271.365997]), array([ -71.5523682,  146.465836 ]))
>>> df.limits(["x", "y"], ["minmax", "90%"])
(array([-128.293991,  271.365997]), array([-13.37438402,  13.4224423 ]))
>>> df.limits(["x", "y"], ["minmax", [0, 10]])
(array([-128.293991,  271.365997]), [0, 10])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **value** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns List in the form [[xmin, xmax], [ymin, ymax], ... ,[zmin, zmax]] or [xmin, xmax] when expression is not a list

limits_percentage(*expression*, *percentage=99.73*, *square=False*, *delay=False*)
Calculate the [min, max] range for expression, containing approximately a percentage of the data as defined by percentage.

The range is symmetric around the median, i.e., for a percentage of 90, this gives the same results as:

Example:

```
>>> df.limits_percentage("x", 90)
array([-12.35081376,  12.14858052]
>>> df.percentile_approx("x", 5), df.percentile_approx("x", 95)
(array([-12.36813152]), array([ 12.13275818]))
```

NOTE: this value is approximated by calculating the cumulative distribution on a grid. NOTE 2: The values above are not exactly the same, since percentile and limits_percentage do not share the same code

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']

- **percentage** (*float*) – Value between 0 and 100
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns List in the form [[xmin, xmax], [ymin, ymax], ... ,[zmin, zmax]] or [xmin, xmax] when expression is not a list

materialize (*virtual_column*, *inplace=False*)

Returns a new DataFrame where the virtual column is turned into an in memory numpy array.

Example:

```
>>> x = np.arange(1,4)
>>> y = np.arange(2,5)
>>> df = vaex.from_arrays(x=x, y=y)
>>> df['r'] = (df.x**2 + df.y**2)**0.5 # 'r' is a virtual column (computed on␣
↪the fly)
>>> df = df.materialize('r')  # now 'r' is a 'real' column (i.e. a numpy␣
↪array)
```

Parameters **inplace** – {inplace}

max (*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*, *edges=False*)

Calculate the maximum for given expressions, possibly on a grid defined by binby.

Example:

```
>>> df.max("x")
array(271.365997)
>>> df.max(["x", "y"])
array([ 271.365997,  146.465836])
>>> df.max("x", binby="x", shape=5, limits=[-10, 10])
array([-6.00010443, -2.00002384,  1.99998057,  5.99983597,  9.99984646])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimension is of shape (2)

mean(*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*, *edges=False*)

Calculate the mean for expression, possibly on a grid defined by binby.

Example:

```
>>> df.mean("x")
-0.067131491264005971
>>> df.mean("(x**2+y**2)**0.5", binby="E", shape=4)
array([  2.43483742,   4.41840721,   8.26742458,  15.53846476])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

median_approx(*expression*, *percentage=50.0*, *binby=[]*, *limits=None*, *shape=128*, *percentile_shape=256*, *percentile_limits='minmax'*, *selection=False*, *delay=False*)

Calculate the median , possibly on a grid defined by binby.

NOTE: this value is approximated by calculating the cumulative distribution on a grid defined by percentile_shape and percentile_limits

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **percentile_limits** – description for the min and max values to use for the cumulative histogram, should currently only be 'minmax'
- **percentile_shape** – shape for the array where the cumulative histogram is calculated on, integer type
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections

- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

min(*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*, *edges=False*)

Calculate the minimum for given expressions, possibly on a grid defined by binby.

Example:

```
>>> df.min("x")
array(-128.293991)
>>> df.min(["x", "y"])
array([-128.293991 ,  -71.5523682])
>>> df.min("x", binby="x", shape=5, limits=[-10, 10])
array([-9.99919128, -5.99972439, -1.99991322,  2.0000093 ,  6.0004878 ])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimension is of shape (2)

minmax(*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)

Calculate the minimum and maximum for expressions, possibly on a grid defined by binby.

Example:

```
>>> df.minmax("x")
array([-128.293991,  271.365997])
>>> df.minmax(["x", "y"])
array([[-128.293991 ,  271.365997 ],
           [ -71.5523682,  146.465836 ]])
>>> df.minmax("x", binby="x", shape=5, limits=[-10, 10])
array([[-9.99919128, -6.00010443],
           [-5.99972439, -2.00002384],
           [-1.99991322,  1.99998057],
           [ 2.0000093 ,  5.99983597],
           [ 6.0004878 ,  9.99984646]])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic, the last dimension is of shape (2)

mode(*expression*, *binby=[]*, *limits=None*, *shape=256*, *mode_shape=64*, *mode_limits=None*, *progressbar=False*, *selection=None*)

Calculate/estimate the mode.

mutual_information(*x*, *y=None*, *mi_limits=None*, *mi_shape=256*, *binby=[]*, *limits=None*, *shape=128*, *sort=False*, *selection=False*, *delay=False*)

Estimate the mutual information between and x and y on a grid with shape mi_shape and mi_limits, possibly on a grid defined by binby.

If sort is True, the mutual information is returned in sorted (descending) order and the list of expressions is returned in the same order.

Example:

```
>>> df.mutual_information("x", "y")
array(0.1511814526380327)
>>> df.mutual_information([["x", "y"], ["x", "z"], ["E", "Lz"]])
array([ 0.15118145,  0.18439181,  1.07067379])
>>> df.mutual_information([["x", "y"], ["x", "z"], ["E", "Lz"]], sort=True)
(array([ 1.07067379,  0.18439181,  0.15118145]),
[['E', 'Lz'], ['x', 'z'], ['x', 'y']])
```

Parameters

- **x** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **y** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']

- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **sort** – return mutual information in sorted (descending) order, and also return the correspond list of expressions when sorted is True
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic,

nbytes
Alias for *df.byte_size()*, see `DataFrame.byte_size()`.

nop(*expression*, *progress=False*, *delay=False*)
Evaluates expression, and drop the result, usefull for benchmarking, since vaex is usually lazy

percentile_approx(*expression*, *percentage=50.0*, *binby=[]*, *limits=None*, *shape=128*, *percentile_shape=1024*, *percentile_limits='minmax'*, *selection=False*, *delay=False*)
Calculate the percentile given by percentage, possibly on a grid defined by binby.

NOTE: this value is approximated by calculating the cumulative distribution on a grid defined by percentile_shape and percentile_limits.

Example:

```
>>> df.percentile_approx("x", 10), df.percentile_approx("x", 90)
(array([-8.3220355]), array([ 7.92080358]))
>>> df.percentile_approx("x", 50, binby="x", shape=5, limits=[-10, 10])
array([[-7.56462982],
           [-3.61036641],
           [-0.01296306],
           [ 3.56697863],
           [ 7.45838367]])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **percentile_limits** – description for the min and max values to use for the cumulative histogram, should currently only be 'minmax'
- **percentile_shape** – shape for the array where the cumulative histogram is calculated on, integer type
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections

- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

plot (*x=None, y=None, z=None, what='count(*)', vwhat=None, reduce=['colormap'], f=None, normalize='normalize', normalize_axis='what', vmin=None, vmax=None, shape=256, vshape=32, limits=None, grid=None, colormap='afmhot', figsize=None, xlabel=None, ylabel=None, aspect='auto', tight_layout=True, interpolation='nearest', show=False, colorbar=True, colorbar_label=None, selection=None, selection_labels=None, title=None, background_color='white', pre_blend=False, background_alpha=1.0, visual={'column': 'what', 'fade': 'selection', 'layer': 'z', 'row': 'subspace', 'x': 'x', 'y': 'y'}, smooth_pre=None, smooth_post=None, wrap=True, wrap_columns=4, return_extra=False, hardcopy=None*)
Viz data in a 2d histogram/heatmap.

Declarative plotting of statistical plots using matplotlib, supports subplots, selections, layers.

Instead of passing x and y, pass a list as x argument for multiple panels. Give what a list of options to have multiple panels. When both are present then will be origanized in a column/row order.

This methods creates a 6 dimensional 'grid', where each dimension can map the a visual dimension. The grid dimensions are:

- x: shape determined by shape, content by x argument or the first dimension of each space
- y: ,,
- z: related to the z argument
- selection: shape equals length of selection argument
- what: shape equals length of what argument
- space: shape equals length of x argument if multiple values are given

By default, this its shape is (1, 1, 1, 1, shape, shape) (where x is the last dimension)

The visual dimensions are

- x: x coordinate on a plot / image (default maps to grid's x)
- y: y ,, (default maps to grid's y)
- layer: each image in this dimension is blended togeher to one image (default maps to z)
- fade: each image is shown faded after the next image (default mapt to selection)
- row: rows of subplots (default maps to space)
- columns: columns of subplot (default maps to what)

All these mappings can be changes by the visual argument, some examples:

```
>>> df.plot('x', 'y', what=['mean(x)', 'correlation(vx, vy)'])
```

Will plot each 'what' as a column.

```
>>> df.plot('x', 'y', selection=['FeH < -3', '(FeH >= -3) & (FeH < -2)'],␣
↪visual=dict(column='selection'))
```

Will plot each selection as a column, instead of a faded on top of each other.

Parameters

- **x** – Expression to bin in the x direction (by default maps to x), or list of pairs, like [['x', 'y'], ['x', 'z']], if multiple pairs are given, this dimension maps to rows by default
- **y** – y (by default maps to y)
- **z** – Expression to bin in the z direction, followed by a :start,end,shape signature, like 'FeH:-3,1:5' will produce 5 layers between -10 and 10 (by default maps to layer)
- **what** – What to plot, count(*) will show a N-d histogram, mean('x'), the mean of the x column, sum('x') the sum, std('x') the standard deviation, correlation('vx', 'vy') the correlation coefficient. Can also be a list of values, like ['count(x)', std('vx')], (by default maps to column)
- **reduce** –
- **f** – transform values by: 'identity' does nothing 'log' or 'log10' will show the log of the value
- **normalize** – normalization function, currently only 'normalize' is supported
- **normalize_axis** – which axes to normalize on, None means normalize by the global maximum.
- **vmin** – instead of automatic normalization, (using normalize and normalization_axis) scale the data between vmin and vmax to [0, 1]
- **vmax** – see vmin
- **shape** – shape/size of the n-D histogram grid
- **limits** – list of [[xmin, xmax], [ymin, ymax]], or a description such as 'minmax', '99%'
- **grid** – if the binning is done before by yourself, you can pass it
- **colormap** – matplotlib colormap to use
- **figsize** – (x, y) tuple passed to pylab.figure for setting the figure size
- **xlabel** –
- **ylabel** –
- **aspect** –
- **tight_layout** – call pylab.tight_layout or not
- **colorbar** – plot a colorbar or not
- **interpolation** – interpolation for imshow, possible options are: 'nearest', 'bilinear', 'bicubic', see matplotlib for more
- **return_extra** –

Returns

plot1d(*x=None, what='count(*)', grid=None, shape=64, facet=None, limits=None, figsize=None, f='identity', n=None, normalize_axis=None, xlabel=None, ylabel=None, label=None, selection=None, show=False, tight_layout=True, hardcopy=None, progress=None, **kwargs*)

Viz data in 1d (histograms, running means etc)

Example

```
>>> df.plot1d(df.x)
>>> df.plot1d(df.x, limits=[0, 100], shape=100)
>>> df.plot1d(df.x, what='mean(y)', limits=[0, 100], shape=100)
```

If you want to do a computation yourself, pass the grid argument, but you are responsible for passing the same limits arguments:

```
>>> counts = df.mean(df.y, binby=df.x, limits=[0, 100], shape=100)/100.
>>> df.plot1d(df.x, limits=[0, 100], shape=100, grid=means, label='mean(y)/100
↪')
```

Parameters

- **x** – Expression to bin in the x direction
- **what** – What to plot, count(*) will show a N-d histogram, mean('x'), the mean of the x column, sum('x') the sum
- **grid** – If the binning is done before by yourself, you can pass it
- **facet** – Expression to produce facetted plots (facet='x:0,1,12' will produce 12 plots with x in a range between 0 and 1)
- **limits** – list of [xmin, xmax], or a description such as 'minmax', '99%'
- **figsize** – (x, y) tuple passed to pylab.figure for setting the figure size
- **f** – transform values by: 'identity' does nothing 'log' or 'log10' will show the log of the value
- **n** – normalization function, currently only 'normalize' is supported, or None for no normalization
- **normalize_axis** – which axes to normalize on, None means normalize by the global maximum.
- **normalize_axis** –
- **xlabel** – String for label on x axis (may contain latex)
- **ylabel** – Same for y axis
- **kwargs** – extra argument passed to pylab.plot

Param tight_layout: call pylab.tight_layout or not

Returns

plot2d_contour(*x=None, y=None, what='count(*)', limits=None, shape=256, selection=None, f='identity', figsize=None, xlabel=None, ylabel=None, aspect='auto', levels=None, fill=False, colorbar=False, colorbar_label=None, colormap=None, colors=None, linewidths=None, linestyles=None, vmin=None, vmax=None, grid=None, show=None, **kwargs*)

Plot conting contours on 2D grid.

Parameters

- **x** – {expression}
- **y** – {expression}
- **what** – What to plot, count(*) will show a N-d histogram, mean('x'), the mean of the x column, sum('x') the sum, std('x') the standard deviation, correlation('vx', 'vy') the correlation coefficient. Can also be a list of values, like ['count(x)', std('vx')], (by default maps to column)
- **limits** – {limits}
- **shape** – {shape}

- **selection** – {selection}
- **f** – transform values by: 'identity' does nothing 'log' or 'log10' will show the log of the value
- **figsize** – (x, y) tuple passed to pylab.figure for setting the figure size
- **xlabel** – label of the x-axis (defaults to param x)
- **ylabel** – label of the y-axis (defaults to param y)
- **aspect** – the aspect ratio of the figure
- **levels** – the contour levels to be passed on pylab.contour or pylab.contourf
- **colorbar** – plot a colorbar or not
- **colorbar_label** – the label of the colourbar (defaults to param what)
- **colormap** – matplotlib colormap to pass on to pylab.contour or pylab.contourf
- **colors** – the colours of the contours
- **linewidths** – the widths of the contours
- **linestyles** – the style of the contour lines
- **vmin** – instead of automatic normalization, scale the data between vmin and vmax
- **vmax** – see vmin
- **grid** – {grid}
- **show** –

plot3d(*x, y, z, vx=None, vy=None, vz=None, vwhat=None, limits=None, grid=None, what='count(*)', shape=128, selection=[None, True], f=None, vcount_limits=None, smooth_pre=None, smooth_post=None, grid_limits=None, normalize='normalize', colormap='afmhot', figure_key=None, fig=None, lighting=True, level=[0.1, 0.5, 0.9], opacity=[0.01, 0.05, 0.1], level_width=0.1, show=True, **kwargs*)

Use at own risk, requires ipyvolume

plot_bq(*x, y, grid=None, shape=256, limits=None, what='count(*)', figsize=None, f='identity', figure_key=None, fig=None, axes=None, xlabel=None, ylabel=None, title=None, show=True, selection=[None, True], colormap='afmhot', grid_limits=None, normalize='normalize', grid_before=None, what_kwargs={}, type='default', scales=None, tool_select=False, bq_cleanup=True, **kwargs*)

Deprecated: use plot_widget

plot_widget(*x, y, z=None, grid=None, shape=256, limits=None, what='count(*)', figsize=None, f='identity', figure_key=None, fig=None, axes=None, xlabel=None, ylabel=None, title=None, show=True, selection=[None, True], colormap='afmhot', grid_limits=None, normalize='normalize', grid_before=None, what_kwargs={}, type='default', scales=None, tool_select=False, bq_cleanup=True, backend='bqplot', **kwargs*)

Viz 1d, 2d or 3d in a Jupyter notebook

Note: This API is not fully settled and may change in the future

Example:

```
>>> df.plot_widget(df.x, df.y, backend='bqplot')
>>> df.plot_widget(df.pickup_longitude, df.pickup_latitude, backend=
↪'ipyleaflet')
```

Parameters backend – Widget backend to use: 'bqplot', 'ipyleaflet', 'ipyvolume', 'matplotlib'

propagate_uncertainties(*columns*, *depending_variables=None*, *cov_matrix='auto'*, *covariance_format='{}_{}_covariance'*, *uncertainty_format='{}_uncertainty'*)
Propagates uncertainties (full covariance matrix) for a set of virtual columns.

Covariance matrix of the depending variables is guessed by finding columns prefixed by "e" or "*e_*" or postfixed by "_error", "_uncertainty", "e" and "*_e*". Off diagonals (covariance or correlation) by postfixes with "_correlation" or "_corr" for correlation or "_covariance" or "_cov" for covariances. (Note that x_y_cov = x_e * y_e * x_y_correlation.)

Example

```
>>> df = vaex.from_scalars(x=1, y=2, e_x=0.1, e_y=0.2)
>>> df["u"] = df.x + df.y
>>> df["v"] = np.log10(df.x)
>>> df.propagate_uncertainties([df.u, df.v])
>>> df.u_uncertainty, df.v_uncertainty
```

Parameters

- **columns** – list of columns for which to calculate the covariance matrix.
- **depending_variables** – If not given, it is found out automatically, otherwise a list of columns which have uncertainties.
- **cov_matrix** – List of list with expressions giving the covariance matrix, in the same order as depending_variables. If 'full' or 'auto', the covariance matrix for the depending_variables will be guessed, where 'full' gives an error if an entry was not found.

remove_virtual_meta()
Removes the file with the virtual column etc, it does not change the current virtual columns etc.

rename_column(*name*, *new_name*, *unique=False*, *store_in_state=True*)
Renames a column, not this is only the in memory name, this will not be reflected on disk

sample(*n=None*, *frac=None*, *replace=False*, *weights=None*, *random_state=None*)
Returns a DataFrame with a random set of rows

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Provide either n or frac.

Example:

```
>>> import vaex, numpy as np
>>> df = vaex.from_arrays(s=np.array(['a', 'b', 'c', 'd']), x=np.arange(1,5))
>>> df
  #  s      x
  0  a      1
  1  b      2
  2  c      3
  3  d      4
>>> df.sample(n=2, random_state=42) # 2 random rows, fixed seed
  #  s      x
```

(continues on next page)

(continued from previous page)

```
  0  b      2
  1  d      4
>>> df.sample(frac=1, random_state=42) # 'shuffling'
  #  s      x
  0  c      3
  1  a      1
  2  d      4
  3  b      2
>>> df.sample(frac=1, replace=True, random_state=42) # useful for bootstrap␣
↪(may contain repeated samples)
  #  s      x
  0  d      4
  1  a      1
  2  a      1
  3  d      4
```

Parameters

- **n** (*int*) – number of samples to take (default 1 if frac is None)
- **frac** (*float*) – fractional number of takes to take
- **replace** (*bool*) – If true, a row may be drawn multiple times
- **or expression weights** (*str*) – (unnormalized) probability that a row can be drawn
- **or RandomState** (*int*) – seed or RandomState for reproducability, when None a random seed it chosen

Returns Returns a new DataFrame with a shallow copy/view of the underlying data

Return type *DataFrame*

scatter(*x, y, xerr=None, yerr=None, cov=None, corr=None, s_expr=None, c_expr=None, labels=None, selection=None, length_limit=50000, length_check=True, label=None, xlabel=None, ylabel=None, errorbar_kwargs={}, ellipse_kwargs={}, **kwargs*)

Viz (small amounts) of data in 2d using a scatter plot

Convenience wrapper around pylab.scatter when for working with small DataFrames or selections

Parameters

- **x** – Expression for x axis
- **y** – Idem for y
- **s_expr** – When given, use if for the s (size) argument of pylab.scatter
- **c_expr** – When given, use if for the c (color) argument of pylab.scatter
- **labels** – Annotate the points with these text values
- **selection** – Single selection expression, or None
- **length_limit** – maximum number of rows it will plot
- **length_check** – should we do the maximum row check or not?
- **label** – label for the legend
- **xlabel** – label for x axis, if None .label(x) is used
- **ylabel** – label for y axis, if None .label(y) is used

- **errorbar_kwargs** – extra dict with arguments passed to plt.errorbar
- **kwargs** – extra arguments passed to pylab.scatter

Returns

select (*boolean_expression*, *mode='replace'*, *name='default'*, *executor=None*)
Perform a selection, defined by the boolean expression, and combined with the previous selection using the given mode.

Selections are recorded in a history tree, per name, undo/redo can be done for them separately.

Parameters

- **boolean_expression** (*str*) – Any valid column expression, with comparison operators
- **mode** (*str*) – Possible boolean operator: replace/and/or/xor/subtract
- **name** (*str*) – history tree or selection 'slot' to use
- **executor** –

Returns

select_box (*spaces*, *limits*, *mode='replace'*, *name='default'*)
Select a n-dimensional rectangular box bounded by limits.

The following examples are equivalent:

```
>>> df.select_box(['x', 'y'], [(0, 10), (0, 1)])
>>> df.select_rectangle('x', 'y', [(0, 10), (0, 1)])
```

Parameters

- **spaces** – list of expressions
- **limits** – sequence of shape [(x1, x2), (y1, y2)]
- **mode** –
- **name** –

Returns

select_circle (*x*, *y*, *xc*, *yc*, *r*, *mode='replace'*, *name='default'*, *inclusive=True*)
Select a circular region centred on xc, yc, with a radius of r.

Example:

```
>>> df.select_circle('x','y',2,3,1)
```

Parameters

- **x** – expression for the x space
- **y** – expression for the y space
- **xc** – location of the centre of the circle in x
- **yc** – location of the centre of the circle in y
- **r** – the radius of the circle
- **name** – name of the selection

- **mode** –

Returns

select_ellipse(*x*, *y*, *xc*, *yc*, *width*, *height*, *angle=0*, *mode='replace'*, *name='default'*, *radians=False*, *inclusive=True*)

Select an elliptical region centred on xc, yc, with a certain width, height and angle.

Example:

```
>>> df.select_ellipse('x','y', 2, -1, 5,1, 30, name='my_ellipse')
```

Parameters

- **x** – expression for the x space
- **y** – expression for the y space
- **xc** – location of the centre of the ellipse in x
- **yc** – location of the centre of the ellipse in y
- **width** – the width of the ellipse (diameter)
- **height** – the width of the ellipse (diameter)
- **angle** – (degrees) orientation of the ellipse, counter-clockwise measured from the y axis
- **name** – name of the selection
- **mode** –

Returns

select_inverse(*name='default'*, *executor=None*)

Invert the selection, i.e. what is selected will not be, and vice versa

Parameters

- **name** (*str*) –
- **executor** –

Returns

select_lasso(*expression_x*, *expression_y*, *xsequence*, *ysequence*, *mode='replace'*, *name='default'*, *executor=None*)

For performance reasons, a lasso selection is handled differently.

Parameters

- **expression_x** (*str*) – Name/expression for the x coordinate
- **expression_y** (*str*) – Name/expression for the y coordinate
- **xsequence** – list of x numbers defining the lasso, together with y
- **ysequence** –
- **mode** (*str*) – Possible boolean operator: replace/and/or/xor/subtract
- **name** (*str*) –
- **executor** –

Returns

select_non_missing(*drop_nan=True*, *drop_masked=True*, *column_names=None*, *mode='replace'*, *name='default'*)
Create a selection that selects rows having non missing values for all columns in column_names.

The name reflect Panda's, no rows are really dropped, but a mask is kept to keep track of the selection

Parameters

- **drop_nan** – drop rows when there is a NaN in any of the columns (will only affect float values)
- **drop_masked** – drop rows when there is a masked value in any of the columns
- **column_names** – The columns to consider, default: all (real, non-virtual) columns
- **mode** (*str*) – Possible boolean operator: replace/and/or/xor/subtract
- **name** (*str*) – history tree or selection 'slot' to use

Returns

select_nothing(*name='default'*)
Select nothing.

select_rectangle(*x*, *y*, *limits*, *mode='replace'*, *name='default'*)
Select a 2d rectangular box in the space given by x and y, bounds by limits.

Example:

```
>>> df.select_box('x', 'y', [(0, 10), (0, 1)])
```

Parameters

- **x** – expression for the x space
- **y** – expression fo the y space
- **limits** – sequence of shape [(x1, x2), (y1, y2)]
- **mode** –

selected_length()
Returns the number of rows that are selected.

selection_can_redo(*name='default'*)
Can selection name be redone?

selection_can_undo(*name='default'*)
Can selection name be undone?

selection_redo(*name='default'*, *executor=None*)
Redo selection, for the name.

selection_undo(*name='default'*, *executor=None*)
Undo selection, for the name.

set_active_fraction(*value*)
Sets the active_fraction, set picked row to None, and remove selection.

TODO: we may be able to keep the selection, if we keep the expression, and also the picked row

set_active_range(*i1*, *i2*)
Sets the active_fraction, set picked row to None, and remove selection.

TODO: we may be able to keep the selection, if we keep the expression, and also the picked row

set_current_row (*value*)
Set the current row, and emit the signal signal_pick.

set_selection (*selection*, *name='default'*, *executor=None*)
Sets the selection object

Parameters

- **selection** – Selection object
- **name** – selection 'slot'
- **executor** –

Returns

set_variable (*name*, *expression_or_value*, *write=True*)
Set the variable to an expression or value defined by expression_or_value.

Example

```
>>> df.set_variable("a", 2.)
>>> df.set_variable("b", "a**2")
>>> df.get_variable("b")
'a**2'
>>> df.evaluate_variable("b")
4.0
```

Parameters

- **name** – Name of the variable
- **write** – write variable to meta file
- **expression** – value or expression

sort (*by*, *ascending=True*, *kind='quicksort'*)
Return a sorted DataFrame, sorted by the expression 'by'

The kind keyword is ignored if doing multi-key sorting.

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Note: Note that filtering will be ignored (since they may change), you may want to consider running `extract()` first.

Example:

```
>>> import vaex, numpy as np
>>> df = vaex.from_arrays(s=np.array(['a', 'b', 'c', 'd']), x=np.arange(1,5))
>>> df['y'] = (df.x-1.8)**2
>>> df
  #  s      x     y
  0  a      1  0.64
  1  b      2  0.04
  2  c      3  1.44
  3  d      4  4.84
>>> df.sort('y', ascending=False)  # Note: passing '(x-1.8)**2' gives the
↪same result
```

(continues on next page)

(continued from previous page)

```
 #  s      x     y
 0  d      4  4.84
 1  c      3  1.44
 2  a      1  0.64
 3  b      2  0.04
```

Parameters

- **or expression by** (*str*) – expression to sort by
- **ascending** (*bool*) – ascending (default, True) or descending (False)
- **kind** (*str*) – kind of algorithm to use (passed to numpy.argsort)

split (*frac*)

Returns a list containing ordered subsets of the DataFrame.

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> for dfs in df.split(frac=0.3):
...     print(dfs.x.values)
...
[0 1 3]
[3 4 5 6 7 8 9]
>>> for split in df.split(frac=[0.2, 0.3, 0.5]):
...     print(dfs.x.values)
[0 1]
[2 3 4]
[5 6 7 8 9]
```

Parameters frac (*int/list*) – If int will split the DataFrame in two portions, the first of which will have size as specified by this parameter. If list, the generator will generate as many portions as elements in the list, where each element defines the relative fraction of that portion.

Returns A list of DataFrames.

Return type list

split_random (*frac, random_state=None*)

Returns a list containing random portions of the DataFrame.

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Example:

```
>>> import vaex, import numpy as np
>>> np.random.seed(111)
>>> df = vaex.from_arrays(x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

(continues on next page)

(continued from previous page)

```
>>> for dfs in df.split_random(frac=0.3, random_state=42):
...     print(dfs.x.values)
...
[8 1 5]
[0 7 2 9 4 3 6]
>>> for split in df.split_random(frac=[0.2, 0.3, 0.5], random_state=42):
...     print(dfs.x.values)
[8 1]
[5 0 7]
[2 9 4 3 6]
```

Parameters

- **frac** (*int/list*) – If int will split the DataFrame in two portions, the first of which will have size as specified by this parameter. If list, the generator will generate as many portions as elements in the list, where each element defines the relative fraction of that portion.
- **random_state** (*int*) – (default, None) Random number seed for reproducibility.

Returns A list of DataFrames.

Return type list

state_get()
Return the internal state of the DataFrame in a dictionary

Example:

```
>>> import vaex
>>> df = vaex.from_scalars(x=1, y=2)
>>> df['r'] = (df.x**2 + df.y**2)**0.5
>>> df.state_get()
{'active_range': [0, 1],
'column_names': ['x', 'y', 'r'],
'description': None,
'descriptions': {},
'functions': {},
'renamed_columns': [],
'selections': {'__filter__': None},
'ucds': {},
'units': {},
'variables': {},
'virtual_columns': {'r': '(((x ** 2) + (y ** 2)) ** 0.5)'}}
```

state_load(*f*, *use_active_range=False*)
Load a state previously stored by `DataFrame.state_store()`, see also `DataFrame.state_set()`.

state_set(*state*, *use_active_range=False*, *trusted=True*)
Sets the internal state of the df

Example:

```
>>> import vaex
>>> df = vaex.from_scalars(x=1, y=2)
>>> df
  #    x    y        r
```

(continues on next page)

(continued from previous page)

```
  0    1    2  2.23607
>>> df['r'] = (df.x**2 + df.y**2)**0.5
>>> state = df.state_get()
>>> state
{'active_range': [0, 1],
'column_names': ['x', 'y', 'r'],
'description': None,
'descriptions': {},
'functions': {},
'renamed_columns': [],
'selections': {'__filter__': None},
'ucds': {},
'units': {},
'variables': {},
'virtual_columns': {'r': '(((x ** 2) + (y ** 2)) ** 0.5)'}}
>>> df2 = vaex.from_scalars(x=3, y=4)
>>> df2.state_set(state)  # now the virtual functions are 'copied'
>>> df2
  #    x    y    r
  0    3    4    5
```

Parameters

- **state** – dict as returned by *DataFrame.state_get()*.
- **use_active_range** (*bool*) – Whether to use the active range or not.

state_write (*f*)

Write the internal state to a json or yaml file (see *DataFrame.state_get()*)

Example

```
>>> import vaex
>>> df = vaex.from_scalars(x=1, y=2)
>>> df['r'] = (df.x**2 + df.y**2)**0.5
>>> df.state_write('state.json')
>>> print(open('state.json').read())
{
"virtual_columns": {
    "r": "(((x ** 2) + (y ** 2)) ** 0.5)"
},
"column_names": [
    "x",
    "y",
    "r"
],
"renamed_columns": [],
"variables": {
    "pi": 3.141592653589793,
    "e": 2.718281828459045,
    "km_in_au": 149597870.7,
    "seconds_per_year": 31557600
},
"functions": {},
"selections": {
    "__filter__": null
},
```

(continues on next page)

(continued from previous page)

```
"ucds": {},
"units": {},
"descriptions": {},
"description": null,
"active_range": [
    0,
    1
]
}
>>> df.state_write('state.yaml')
>>> print(open('state.yaml').read())
active_range:
- 0
- 1
column_names:
- x
- y
- r
description: null
descriptions: {}
functions: {}
renamed_columns: []
selections:
__filter__: null
ucds: {}
units: {}
variables:
pi: 3.141592653589793
e: 2.718281828459045
km_in_au: 149597870.7
seconds_per_year: 31557600
virtual_columns:
r: (((x ** 2) + (y ** 2)) ** 0.5)
```

Parameters **f** (*str*) – filename (ending in .json or .yaml)

std(*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Calculate the standard deviation for the given expression, possible on a grid defined by binby

```
>>> df.std("vz")
110.31773397535071
>>> df.std("vz", binby=["(x**2+y**2)**0.5"], shape=4)
array([ 123.57954851,   85.35190177,   61.14345748,   38.0740619 ])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]

- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

sum(*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*, *edges=False*)

Calculate the sum for the given expression, possible on a grid defined by binby

Example:

```
>>> df.sum("L")
304054882.49378014
>>> df.sum("L", binby="E", shape=4)
array([  8.83517994e+06,   5.92217598e+07,   9.55218726e+07,
                 1.40008776e+08])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

tail(*n=10*)

Return a shallow copy a DataFrame with the last n rows.

take(*indices*, *filtered=True*, *dropfilter=True*)

Returns a DataFrame containing only rows indexed by indices

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Example:

```
>>> import vaex, numpy as np
>>> df = vaex.from_arrays(s=np.array(['a', 'b', 'c', 'd']), x=np.arange(1,5))
>>> df.take([0,2])
 #  s      x
 0  a      1
 1  c      3
```

Parameters

- **indices** – sequence (list or numpy array) with row numbers
- **filtered** – (for internal use) The indices refer to the filtered data.
- **dropfilter** – (for internal use) Drop the filter, set to False when indices refer to unfiltered, but may contain rows that still need to be filtered out.

Returns DataFrame which is a shallow copy of the original data.

Return type *DataFrame*

to_arrays(*column_names=None*, *selection=None*, *strings=True*, *virtual=True*, *parallel=True*)
Return a list of ndarrays

Parameters

- **column_names** – list of column names, to export, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None

Returns list of (name, ndarray) pairs

to_arrow_table(*column_names=None*, *selection=None*, *strings=True*, *virtual=False*)
Returns an arrow Table object containing the arrays corresponding to the evaluated data

Parameters

- **column_names** – list of column names, to export, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None

Returns pyarrow.Table object

to_astropy_table(*column_names=None*, *selection=None*, *strings=True*, *virtual=False*, *index=None*, *parallel=True*)
Returns a astropy table object containing the ndarrays corresponding to the evaluated data

Parameters

- **column_names** – list of column names, to export, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None
- **index** – if this column is given it is used for the index of the DataFrame

Returns astropy.table.Table object

to_copy (*column_names=None*, *selection=None*, *strings=True*, *virtual=False*, *selections=True*)
Return a copy of the DataFrame, if selection is None, it does not copy the data, it just has a reference

Parameters

- **column_names** – list of column names, to copy, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None
- **selections** – copy selections to a new DataFrame

Returns dict

to_dask_array (*chunks='auto'*)
Lazily expose the DataFrame as a dask.array

Example

```
>>> df = vaex.example()
>>> A = df[['x', 'y', 'z']].to_dask_array()
>>> A
dask.array<vaex-df-1f048b40-10ec-11ea-9553, shape=(330000, 3), dtype=float64,␣
↪chunksize=(330000, 3), chunktype=numpy.ndarray>
>>> A+1
dask.array<add, shape=(330000, 3), dtype=float64, chunksize=(330000, 3),␣
↪chunktype=numpy.ndarray>
```

Parameters **chunks** – How to chunk the array, similar to `dask.array.from_array()`.

Returns `dask.array.Array` object.

to_dict (*column_names=None*, *selection=None*, *strings=True*, *virtual=False*, *parallel=True*)
Return a dict containing the ndarray corresponding to the evaluated data

Parameters

- **column_names** – list of column names, to export, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used

- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None

Returns dict

to_items(*column_names=None*, *selection=None*, *strings=True*, *virtual=False*, *parallel=True*)
Return a list of [(column_name, ndarray), . . .)] pairs where the ndarray corresponds to the evaluated data

Parameters

- **column_names** – list of column names, to export, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None

Returns list of (name, ndarray) pairs

to_pandas_df(*column_names=None*, *selection=None*, *strings=True*, *virtual=False*, *index_name=None*, *parallel=True*)
Return a pandas DataFrame containing the ndarray corresponding to the evaluated data

If index is given, that column is used for the index of the dataframe.

Example

```
>>> df_pandas = df.to_pandas_df(["x", "y", "z"])
>>> df_copy = vaex.from_pandas(df_pandas)
```

Parameters

- **column_names** – list of column names, to export, when None DataFrame.get_column_names(strings=strings, virtual=virtual) is used
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **strings** – argument passed to DataFrame.get_column_names when column_names is None
- **virtual** – argument passed to DataFrame.get_column_names when column_names is None
- **index_column** – if this column is given it is used for the index of the DataFrame

Returns pandas.DataFrame object

trim(*inplace=False*)
Return a DataFrame, where all columns are 'trimmed' by the active range.

For the returned DataFrame, df.get_active_range() returns (0, df.length_original()).

Note: Note that no copy of the underlying data is made, only a view/reference is make.

Parameters **inplace** – Make modifications to self or return a new DataFrame

Return type *DataFrame*

ucd_find(*ucds*, *exclude=[]*)

Find a set of columns (names) which have the ucd, or part of the ucd.

Prefixed with a ^, it will only match the first part of the ucd.

Example

```
>>> df.ucd_find('pos.eq.ra', 'pos.eq.dec')
['RA', 'DEC']
>>> df.ucd_find('pos.eq.ra', 'doesnotexist')
>>> df.ucds[df.ucd_find('pos.eq.ra')]
'pos.eq.ra;meta.main'
>>> df.ucd_find('meta.main')]
'dec'
>>> df.ucd_find('^meta.main')]
```

unit(*expression*, *default=None*)

Returns the unit (an astropy.unit.Units object) for the expression.

Example

```
>>> import vaex
>>> ds = vaex.example()
>>> df.unit("x")
Unit("kpc")
>>> df.unit("x*L")
Unit("km kpc2 / s")
```

Parameters

- **expression** – Expression, which can be a column name
- **default** – if no unit is known, it will return this

Returns The resulting unit of the expression

Return type astropy.units.Unit

validate_expression(*expression*)

Validate an expression (may throw Exceptions)

var(*expression*, *binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)

Calculate the sample variance for the given expression, possible on a grid defined by binby

Example:

```
>>> df.var("vz")
12170.002429456246
>>> df.var("vz", binby=["(x**2+y**2)**0.5"], shape=4)
array([ 15271.90481083,   7284.94713504,   3738.52239232,   1449.63418988])
>>> df.var("vz", binby=["(x**2+y**2)**0.5"], shape=4)**0.5
array([ 123.57954851,   85.35190177,   61.14345748,   38.0740619 ])
```

(continues on next page)

(continued from previous page)

```
>>> df.std("vz", binby=["(x**2+y**2)**0.5"], shape=4)
array([ 123.57954851,   85.35190177,   61.14345748,   38.0740619 ])
```

Parameters

- **expression** – expression or list of expressions, e.g. 'x', or ['x, 'y']
- **binby** – List of expressions for constructing a binned grid
- **limits** – description for the min and max values for the expressions, e.g. 'minmax', '99.7%', [0, 10], or a list of, e.g. [[0, 10], [0, 20], 'minmax']
- **shape** – shape for the array where the statistic is calculated on, if only an integer is given, it is used for all dimensions, e.g. shape=128, shape=[128, 256]
- **selection** – Name of selection to use (or True for the 'default'), or all the data (when selection is None or False), or a list of selections
- **delay** – Do not return the result, but a proxy for delayhronous calculations (currently only for internal use)
- **progress** – A callable that takes one argument (a floating point value between 0 and 1) indicating the progress, calculations are cancelled when this callable returns False

Returns Numpy array with the given shape, or a scalar when no binby argument is given, with the statistic

6.2.2 DataFrameLocal class

class vaex.dataframe.**DataFrameLocal**(*name*, *path*, *column_names*)

Bases: *vaex.dataframe.DataFrame*

Base class for DataFrames that work with local file/data

__array__(*dtype=None*, *parallel=True*)

Gives a full memory copy of the DataFrame into a 2d numpy array of shape (n_rows, n_columns). Note that the memory order is fortran, so all values of 1 column are contiguous in memory for performance reasons.

Note this returns the same result as:

```
>>> np.array(ds)
```

If any of the columns contain masked arrays, the masks are ignored (i.e. the masked elements are returned as well).

__call__(**expressions*, ***kwargs*)

The local implementation of `DataFrame.__call__()`

__init__(*name*, *path*, *column_names*)

Initialize self. See help(type(self)) for accurate signature.

binby(*by=None*, *agg=None*)

Return a `BinBy` or `DataArray` object when agg is not None

The binby operations does not return a 'flat' DataFrame, instead it returns an N-d grid in the form of an xarray.

Parameters **list or agg agg** (*dict,*) – Aggregate operation in the form of a string, vaex.agg object, a dictionary where the keys indicate the target column names, and the values the operations, or the a list of aggregates. When not given, it will return the binby object.

Returns `DataArray` or `BinBy` object.

categorize(*column*, *labels=None*, *check=True*)
Mark column as categorical, with given labels, assuming zero indexing

compare(*other*, *report_missing=True*, *report_difference=False*, *show=10*, *orderby=None*, *column_names=None*)
Compare two DataFrames and report their difference, use with care for large DataFrames

concat(*other*)
Concatenates two DataFrames, adding the rows of one the other DataFrame to the current, returned in a new DataFrame.

No copy of the data is made.

Parameters **other** – The other DataFrame that is concatenated with this DataFrame

Returns New DataFrame with the rows concatenated

Return type DataFrameConcatenated

data
Gives direct access to the data as numpy arrays.

Convenient when working with IPython in combination with small DataFrames, since this gives tab-completion. Only real columns (i.e. no virtual) columns can be accessed, for getting the data from virtual columns, use DataFrame.evalulate(. . .).

Columns can be accesed by there names, which are attributes. The attribues are of type numpy.ndarray.

Example:

```
>>> df = vaex.example()
>>> r = np.sqrt(df.data.x**2 + df.data.y**2)
```

evaluate(*expression*, *i1=None*, *i2=None*, *out=None*, *selection=None*, *filtered=True*, *internal=False*, *parallel=True*)
The local implementation of `DataFrame.evaluate()`

export(*path*, *column_names=None*, *byteorder='='*, *shuffle=False*, *selection=False*, *progress=None*, *virtual=False*, *sort=None*, *ascending=True*)
Exports the DataFrame to a file written with arrow

Parameters

- **df** (`DataFrameLocal`) – DataFrame to export
- **path** (*str*) – path for file
- **column_names** (*lis[str]*) – list of column names to export or None for all columns
- **byteorder** (*str*) – = for native, < for little endian and > for big endian (not supported for fits)
- **shuffle** (*bool*) – export rows in random order
- **selection** (*bool*) – export selection or not
- **progress** – progress callback that gets a progress fraction as argument and should return True to continue, or a default progress bar when progress=True
- **sort** (*str*) – expression used for sorting the output

- **ascending** (*bool*) – sort ascending (True) or descending

Param bool virtual: When True, export virtual columns

Returns

export_arrow(*path*, *column_names=None*, *byteorder='='*, *shuffle=False*, *selection=False*, *progress=None*, *virtual=False*, *sort=None*, *ascending=True*)

Exports the DataFrame to a file written with arrow

Parameters

- **df** (DataFrameLocal) – DataFrame to export
- **path** (*str*) – path for file
- **column_names** (*lis[str]*) – list of column names to export or None for all columns
- **byteorder** (*str*) – = for native, < for little endian and > for big endian
- **shuffle** (*bool*) – export rows in random order
- **selection** (*bool*) – export selection or not
- **progress** – progress callback that gets a progress fraction as argument and should return True to continue, or a default progress bar when progress=True
- **sort** (*str*) – expression used for sorting the output
- **ascending** (*bool*) – sort ascending (True) or descending

Param bool virtual: When True, export virtual columns

Returns

export_fits(*path*, *column_names=None*, *shuffle=False*, *selection=False*, *progress=None*, *virtual=False*, *sort=None*, *ascending=True*)

Exports the DataFrame to a fits file that is compatible with TOPCAT colfits format

Parameters

- **df** (DataFrameLocal) – DataFrame to export
- **path** (*str*) – path for file
- **column_names** (*lis[str]*) – list of column names to export or None for all columns
- **shuffle** (*bool*) – export rows in random order
- **selection** (*bool*) – export selection or not
- **progress** – progress callback that gets a progress fraction as argument and should return True to continue, or a default progress bar when progress=True
- **sort** (*str*) – expression used for sorting the output
- **ascending** (*bool*) – sort ascending (True) or descending

Param bool virtual: When True, export virtual columns

Returns

export_hdf5(*path*, *column_names=None*, *byteorder='='*, *shuffle=False*, *selection=False*, *progress=None*, *virtual=False*, *sort=None*, *ascending=True*)

Exports the DataFrame to a vaex hdf5 file

Parameters

- **df** (DataFrameLocal) – DataFrame to export

- **path** (*str*) – path for file
- **column_names** (*lis[str]*) – list of column names to export or None for all columns
- **byteorder** (*str*) – = for native, < for little endian and > for big endian
- **shuffle** (*bool*) – export rows in random order
- **selection** (*bool*) – export selection or not
- **progress** – progress callback that gets a progress fraction as argument and should return True to continue, or a default progress bar when progress=True
- **sort** (*str*) – expression used for sorting the output
- **ascending** (*bool*) – sort ascending (True) or descending

Param bool virtual: When True, export virtual columns

Returns

export_parquet (*path*, *column_names=None*, *byteorder='='*, *shuffle=False*, *selection=False*, *progress=None*, *virtual=False*, *sort=None*, *ascending=True*)

Exports the DataFrame to a parquet file

Parameters

- **df** (*DataFrameLocal*) – DataFrame to export
- **path** (*str*) – path for file
- **column_names** (*lis[str]*) – list of column names to export or None for all columns
- **byteorder** (*str*) – = for native, < for little endian and > for big endian
- **shuffle** (*bool*) – export rows in random order
- **selection** (*bool*) – export selection or not
- **progress** – progress callback that gets a progress fraction as argument and should return True to continue, or a default progress bar when progress=True
- **sort** (*str*) – expression used for sorting the output
- **ascending** (*bool*) – sort ascending (True) or descending

Param bool virtual: When True, export virtual columns

Returns

groupby (*by=None*, *agg=None*)

Return a GroupBy or *DataFrame* object when agg is not None

Examples:

```
>>> import vaex
>>> import numpy as np
>>> np.random.seed(42)
>>> x = np.random.randint(1, 5, 10)
>>> y = x**2
>>> df = vaex.from_arrays(x=x, y=y)
>>> df.groupby(df.x, agg='count')
#    x    y_count
0    3          4
1    4          2
2    1          3
```

(continues on next page)

(continued from previous page)

```
3    2           1
>>> df.groupby(df.x, agg=[vaex.agg.count('y'), vaex.agg.mean('y')])
#    x    y_count    y_mean
0    3          4         9
1    4          2        16
2    1          3         1
3    2          1         4
>>> df.groupby(df.x, agg={'z': [vaex.agg.count('y'), vaex.agg.mean('y')]})
#    x    z_count    z_mean
0    3          4         9
1    4          2        16
2    1          3         1
3    2          1         4
```

Example using datetime:

```
>>> import vaex
>>> import numpy as np
>>> t = np.arange('2015-01-01', '2015-02-01', dtype=np.datetime64)
>>> y = np.arange(len(t))
>>> df = vaex.from_arrays(t=t, y=y)
>>> df.groupby(vaex.BinnerTime.per_week(df.t)).agg({'y' : 'sum'})
#  t                      y
0  2015-01-01 00:00:00   21
1  2015-01-08 00:00:00   70
2  2015-01-15 00:00:00  119
3  2015-01-22 00:00:00  168
4  2015-01-29 00:00:00   87
```

Parameters **list or agg agg** (`dict,`) – Aggregate operation in the form of a string, vaex.agg object, a dictionary where the keys indicate the target column names, and the values the operations, or the a list of aggregates. When not given, it will return the groupby object.

Returns `DataFrame` or `GroupBy` object.

is_local()

The local implementation of `DataFrame.evaluate()`, always returns True.

join(*other*, *on=None*, *left_on=None*, *right_on=None*, *lprefix=''*, *rprefix=''*, *lsuffix=''*, *rsuffix=''*, *how='left'*, *allow_duplication=False*, *inplace=False*)

Return a DataFrame joined with other DataFrames, matched by columns/expression on/left_on/right_on

If neither on/left_on/right_on is given, the join is done by simply adding the columns (i.e. on the implicit row index).

Note: The filters will be ignored when joining, the full DataFrame will be joined (since filters may change). If either DataFrame is heavily filtered (contains just a small number of rows) consider running `DataFrame.extract()` first.

Example:

```
>>> a = np.array(['a', 'b', 'c'])
>>> x = np.arange(1,4)
>>> ds1 = vaex.from_arrays(a=a, x=x)
>>> b = np.array(['a', 'b', 'd'])
>>> y = x**2
```

(continues on next page)

(continued from previous page)

```
>>> ds2 = vaex.from_arrays(b=b, y=y)
>>> ds1.join(ds2, left_on='a', right_on='b')
```

Parameters

- **other** – Other DataFrame to join with (the right side)
- **on** – default key for the left table (self)
- **left_on** – key for the left table (self), overrides on
- **right_on** – default key for the right table (other), overrides on
- **lprefix** – prefix to add to the left column names in case of a name collision
- **rprefix** – similar for the right
- **lsuffix** – suffix to add to the left column names in case of a name collision
- **rsuffix** – similar for the right
- **how** – how to join, 'left' keeps all rows on the left, and adds columns (with possible missing values) 'right' is similar with self and other swapped. 'inner' will only return rows which overlap.
- **allow_duplication** (*bool*) – Allow duplication of rows when the joined column contains non-unique values.
- **inplace** – Make modifications to self or return a new DataFrame

Returns

label_encode(*column*, *values=None*, *inplace=False*)
Deprecated: use is_category

Encode column as ordinal values and mark it as categorical.

The existing column is renamed to a hidden column and replaced by a numerical columns with values between [0, len(values)-1].

length(*selection=False*)
Get the length of the DataFrames, for the selection of the whole DataFrame.

If selection is False, it returns len(df).

TODO: Implement this in DataFrameRemote, and move the method up in `DataFrame.length()`

Parameters **selection** – When True, will return the number of selected rows

Returns

ordinal_encode(*column*, *values=None*, *inplace=False*)
Deprecated: use is_category

Encode column as ordinal values and mark it as categorical.

The existing column is renamed to a hidden column and replaced by a numerical columns with values between [0, len(values)-1].

selected_length(*selection='default'*)
The local implementation of `DataFrame.selected_length()`

shallow_copy (*virtual=True*, *variables=True*)
Creates a (shallow) copy of the DataFrame.

It will link to the same data, but will have its own state, e.g. virtual columns, variables, selection etc.

6.2.3 Expression class

class vaex.expression.**Expression** (*ds*, *expression*, *ast=None*)
Bases: object

Expression class

__abs__ ()
Returns the absolute value of the expression

__init__ (*ds*, *expression*, *ast=None*)
Initialize self. See help(type(self)) for accurate signature.

__repr__ ()
Return repr(self).

__str__ ()
Return str(self).

__weakref__
list of weak references to the object (if defined)

abs (***kwargs*)
Lazy wrapper around `numpy.abs`

apply (*f*)
Apply a function along all values of an Expression.

Example:

```
>>> df = vaex.example()
>>> df.x
Expression = x
Length: 330,000 dtype: float64 (column)
---------------------------------------
     0  -0.777471
     1    3.77427
     2    1.37576
     3   -7.06738
     4   0.243441
```

```
>>> def func(x):
...     return x**2
```

```
>>> df.x.apply(func)
Expression = lambda_function(x)
Length: 330,000 dtype: float64 (expression)
-------------------------------------------
     0   0.604461
     1    14.2451
     2    1.89272
     3    49.9478
     4  0.0592637
```

Parameters **f** – A function to be applied on the Expression values

Returns A function that is lazily evaluated when called.

arccos(***kwargs*)
Lazy wrapper around `numpy.arccos`

arccosh(***kwargs*)
Lazy wrapper around `numpy.arccosh`

arcsin(***kwargs*)
Lazy wrapper around `numpy.arcsin`

arcsinh(***kwargs*)
Lazy wrapper around `numpy.arcsinh`

arctan(***kwargs*)
Lazy wrapper around `numpy.arctan`

arctan2(***kwargs*)
Lazy wrapper around `numpy.arctan2`

arctanh(***kwargs*)
Lazy wrapper around `numpy.arctanh`

ast
Returns the abstract syntax tree (AST) of the expression

clip(***kwargs*)
Lazy wrapper around `numpy.clip`

copy(*df=None*)
Efficiently copies an expression.

Expression objects have both a string and AST representation. Creating the AST representation involves parsing the expression, which is expensive.

Using copy will deepcopy the AST when the expression was already parsed.

Parameters **df** – DataFrame for which the expression will be evaluated (self.df if None)

cos(***kwargs*)
Lazy wrapper around `numpy.cos`

cosh(***kwargs*)
Lazy wrapper around `numpy.cosh`

count(*binby=[], limits=None, shape=128, selection=False, delay=False, edges=False, progress=None*)
Shortcut for ds.count(expression, . . .), see *Dataset.count*

countmissing()
Returns the number of missing values in the expression.

countna()
Returns the number of Not Availiable (N/A) values in the expression. This includes missing values and np.nan values.

countnan()
Returns the number of NaN values in the expression.

deg2rad(***kwargs*)
Lazy wrapper around `numpy.deg2rad`

dt
Gives access to datetime operations via *DateTime*

exp (***kwargs*)
Lazy wrapper around numpy.exp

expand (*stop=[]*)
Expand the expression such that no virtual columns occurs, only normal columns.

Example:

```
>>> df = vaex.example()
>>> r = np.sqrt(df.data.x**2 + df.data.y**2)
>>> r.expand().expression
'sqrt(((x ** 2) + (y ** 2)))'
```

expm1 (***kwargs*)
Lazy wrapper around numpy.expm1

fillmissing (*value*)
Returns an array where missing values are replaced by value. See *:ismissing* for the definition of missing values.

fillna (*value*)
Returns an array where NA values are replaced by value. See *:isna* for the definition of missing values.

fillnan (*value*)
Returns an array where nan values are replaced by value. See *:isnan* for the definition of missing values.

format (*format*)
Uses http://www.cplusplus.com/reference/string/to_string/ for formatting

isfinite (***kwargs*)
Lazy wrapper around numpy.isfinite

isin (*values*)
Lazily tests if each value in the expression is present in values.

Parameters **values** – List/array of values to check

Returns *Expression* with the lazy expression.

ismissing ()
Returns True where there are missing values (masked arrays), missing strings or None

isna ()
Returns a boolean expression indicating if the values are Not Availiable (missing or NaN).

isnan ()
Returns an array where there are NaN values

log (***kwargs*)
Lazy wrapper around numpy.log

log10 (***kwargs*)
Lazy wrapper around numpy.log10

log1p (***kwargs*)
Lazy wrapper around numpy.log1p

map (*mapper, nan_value=None, missing_value=None, default_value=None, allow_missing=False*)
Map values of an expression or in memory column accoring to an input dictionary or a custom callable function.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(color=['red', 'red', 'blue', 'red', 'green'])
>>> mapper = {'red': 1, 'blue': 2, 'green': 3}
>>> df['color_mapped'] = df.color.map(mapper)
>>> df
#  color      color_mapped
0  red                   1
1  red                   1
2  blue                  2
3  red                   1
4  green                 3
>>> import numpy as np
>>> df = vaex.from_arrays(type=[0, 1, 2, 2, 2, np.nan])
>>> df['role'] = df['type'].map({0: 'admin', 1: 'maintainer', 2: 'user', np.
↪nan: 'unknown'})
>>> df
#    type  role
0       0  admin
1       1  maintainer
2       2  user
3       2  user
4       2  user
5     nan  unknown
>>> import vaex
>>> import numpy as np
>>> df = vaex.from_arrays(type=[0, 1, 2, 2, 2, 4])
>>> df['role'] = df['type'].map({0: 'admin', 1: 'maintainer', 2: 'user'},␣
↪default_value='unknown')
>>> df
#    type  role
0       0  admin
1       1  maintainer
2       2  user
3       2  user
4       2  user
5       4  unknown
:param mapper: dict like object used to map the values from keys to values
:param nan_value: value to be used when a nan is present (and not in the␣
↪mapper)
:param missing_value: value to use used when there is a missing value
:param default_value: value to be used when a value is not in the mapper␣
↪(like dict.get(key, default))
:param allow_missing: used to signal that values in the mapper should map to␣
↪a masked array with missing values,
    assumed True when default_value is not None.
:return: A vaex expression
:rtype: vaex.expression.Expression
```

masked
Alias to df.is_masked(expression)

max (*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Shortcut for ds.max(expression, ...), see *Dataset.max*

maximum (***kwargs*)
Lazy wrapper around `numpy.maximum`

mean (*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)

Shortcut for ds.mean(expression, ...), see *Dataset.mean*

min (*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Shortcut for ds.min(expression, ...), see *Dataset.min*

minimum (***kwargs*)
Lazy wrapper around `numpy.minimum`

minmax (*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Shortcut for ds.minmax(expression, ...), see *Dataset.minmax*

nop ()
Evaluates expression, and drop the result, usefull for benchmarking, since vaex is usually lazy

notna ()
Opposite of isna

nunique (*dropna=False*, *dropnan=False*, *dropmissing=False*, *selection=None*, *delay=False*)
Counts number of unique values, i.e. *len(df.x.unique()) == df.x.nunique()*.

Parameters

- **dropmissing** – do not count missing values
- **dropnan** – do not count nan values
- **dropna** – short for any of the above, (see `Expression.isna()`)

rad2deg (***kwargs*)
Lazy wrapper around `numpy.rad2deg`

searchsorted (***kwargs*)
Lazy wrapper around `numpy.searchsorted`

sin (***kwargs*)
Lazy wrapper around `numpy.sin`

sinc (***kwargs*)
Lazy wrapper around `numpy.sinc`

sinh (***kwargs*)
Lazy wrapper around `numpy.sinh`

sqrt (***kwargs*)
Lazy wrapper around `numpy.sqrt`

std (*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Shortcut for ds.std(expression, ...), see *Dataset.std*

str
Gives access to string operations via `StringOperations`

str_pandas
Gives access to string operations via `StringOperationsPandas` (using Pandas Series)

sum (*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Shortcut for ds.sum(expression, ...), see *Dataset.sum*

tan (***kwargs*)
Lazy wrapper around `numpy.tan`

tanh (***kwargs*)
Lazy wrapper around `numpy.tanh`

td
Gives access to timedelta operations via *TimeDelta*

to_numpy()
Return a numpy representation of the data

to_pandas_series()
Return a pandas.Series representation of the expression.

Note: Pandas is likely to make a memory copy of the data.

tolist()
Short for expr.evaluate().tolist()

transient
If this expression is not transient (e.g. on disk) optimizations can be made

unique(*dropna=False*, *dropnan=False*, *dropmissing=False*, *selection=None*, *delay=False*)
Returns all unique values.

Parameters

- **dropmissing** – do not count missing values
- **dropnan** – do not count nan values
- **dropna** – short for any of the above, (see *Expression.isna()*)

value_counts(*dropna=False*, *dropnan=False*, *dropmissing=False*, *ascending=False*, *progress=False*)
Computes counts of unique values.

WARNING:

- If the expression/column is not categorical, it will be converted on the fly
- dropna is False by default, it is True by default in pandas

Parameters

- **dropna** – when True, it will not report the NA (see *Expression.isna()*)
- **dropnan** – when True, it will not report the nans(see *Expression.isnan()*)
- **dropmissing** – when True, it will not report the missing values (see *Expression.ismissing()*)
- **ascending** – when False (default) it will report the most frequent occuring item first

Returns Pandas series containing the counts

var(*binby=[]*, *limits=None*, *shape=128*, *selection=False*, *delay=False*, *progress=None*)
Shortcut for ds.std(expression, . . .), see *Dataset.var*

variables(*ourself=False*, *expand_virtual=True*, *include_virtual=True*)
Return a set of variables this expression depends on.

Example:

```
>>> df = vaex.example()
>>> r = np.sqrt(df.data.x**2 + df.data.y**2)
>>> r.variables()
{'x', 'y'}
```

where (***kwargs*)
Lazy wrapper around `numpy.where`

6.2.4 Aggregation and statistics

class vaex.stat.**Expression**
Bases: `object`

Describes an expression for a statistic

calculate (*ds*, *binby=[]*, *shape=256*, *limits=None*, *selection=None*)
Calculate the statistic for a `Dataset`

vaex.stat.**correlation** (*x*, *y*)
Creates a standard deviation statistic

vaex.stat.**count** (*expression='*'*)
Creates a count statistic

vaex.stat.**covar** (*x*, *y*)
Creates a standard deviation statistic

vaex.stat.**mean** (*expression*)
Creates a mean statistic

vaex.stat.**std** (*expression*)
Creates a standard deviation statistic

vaex.stat.**sum** (*expression*)
Creates a sum statistic

class vaex.agg.**AggregatorDescriptorMean** (*name*, *expression*, *short_name='mean'*, *selection=None*)
Bases: `vaex.agg.AggregatorDescriptorMulti`

class vaex.agg.**AggregatorDescriptorMulti** (*name*, *expression*, *short_name*, *selection=None*)
Bases: `vaex.agg.AggregatorDescriptor`

Uses multiple operations/aggregation to calculate the final aggretation

class vaex.agg.**AggregatorDescriptorStd** (*name*, *expression*, *short_name='var'*, *ddof=0*, *selection=None*)
Bases: `vaex.agg.AggregatorDescriptorVar`

class vaex.agg.**AggregatorDescriptorVar** (*name*, *expression*, *short_name='var'*, *ddof=0*, *selection=None*)
Bases: `vaex.agg.AggregatorDescriptorMulti`

vaex.agg.**count** (*expression='*'*, *selection=None*)
Creates a count aggregation

vaex.agg.**first** (*expression*, *order_expression*, *selection=None*)
Creates a max aggregation

vaex.agg.**max** (*expression*, *selection=None*)
Creates a max aggregation

vaex.agg.**mean** (*expression*, *selection=None*)
Creates a mean aggregation

vaex.agg.**min** (*expression*, *selection=None*)
Creates a min aggregation

vaex.agg.**nunique**(*expression*, *dropna=False*, *dropnan=False*, *dropmissing=False*, *selection=None*)
Aggregator that calculates the number of unique items per bin.

Parameters

- **expression** – Expression for which to calculate the unique items
- **dropmissing** – do not count missing values
- **dropnan** – do not count nan values
- **dropna** – short for any of the above, (see Expression.isna())

vaex.agg.**std**(*expression*, *ddof=0*, *selection=None*)
Creates a standard deviation aggregation

vaex.agg.**sum**(*expression*, *selection=None*)
Creates a sum aggregation

vaex.agg.**var**(*expression*, *ddof=0*, *selection=None*)
Creates a variance aggregation

6.3 Extensions

6.3.1 String operations

class vaex.expression.**StringOperations**(*expression*)
Bases: object

String operations.

Usually accessed using e.g. *df.name.str.lower()*

__init__(*expression*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

byte_length()
Returns the number of bytes in a string sample.

Returns an expression contains the number of bytes in each sample of a string column.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.byte_length()
Expression = str_byte_length(text)
Length: 5 dtype: int64 (expression)
```

(continues on next page)

(continued from previous page)

```
---------------------------------
0    9
1   11
2    9
3    3
4    4
```

capitalize()

Capitalize the first letter of a string sample.

Returns an expression containing the capitalized strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.capitalize()
Expression = str_capitalize(text)
Length: 5 dtype: str (expression)
---------------------------------
0    Something
1  Very pretty
2    Is coming
3          Our
4         Way.
```

cat(*other*)

Concatenate two string columns on a row-by-row basis.

Parameters **other** (*expression*) – The expression of the other column to be concatenated.

Returns an expression containing the concatenated columns.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.cat(df.text)
Expression = str_cat(text, text)
```

(continues on next page)

(continued from previous page)

```
Length: 5 dtype: str (expression)
---------------------------------
0      SomethingSomething
1  very prettyvery pretty
2      is comingis coming
3                  ourour
4                way.way.
```

center (*width*, *fillchar=' '*)

Fills the left and right side of the strings with additional characters, such that the sample has a total of *width* characters.

Parameters

- **width** (*int*) – The total number of characters of the resulting string sample.
- **fillchar** (*str*) – The character used for filling.

Returns an expression containing the filled strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.center(width=11, fillchar='!')
Expression = str_center(text, width=11, fillchar='!')
Length: 5 dtype: str (expression)
---------------------------------
0  !Something!
1  very pretty
2  !is coming!
3  !!!!our!!!!
4  !!!!way.!!!
```

contains (*pattern*, *regex=True*)

Check if a string pattern or regex is contained within a sample of a string column.

Parameters

- **pattern** (*str*) – A string or regex pattern
- **regex** (*bool*) – If True,

Returns an expression which is evaluated to True if the pattern is found in a given sample, and it is False otherwise.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
```

(continues on next page)

(continued from previous page)

```
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.contains('very')
Expression = str_contains(text, 'very')
Length: 5 dtype: bool (expression)
----------------------------------
0  False
1   True
2  False
3  False
4  False
```

count (*pat*, *regex=False*)

Count the occurences of a pattern in sample of a string column.

Parameters

- **pat** (*str*) – A string or regex pattern
- **regex** (*bool*) – If True,

Returns an expression containing the number of times a pattern is found in each sample.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.count(pat="et", regex=False)
Expression = str_count(text, pat='et', regex=False)
Length: 5 dtype: int64 (expression)
-----------------------------------
0  1
1  1
2  0
3  0
4  0
```

endswith (*pat*)

Check if the end of each string sample matches the specified pattern.

Parameters **pat** (*str*) – A string pattern or a regex

Returns an expression evaluated to True if the pattern is found at the end of a given sample, False otherwise.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.endswith(pat="ing")
Expression = str_endswith(text, pat='ing')
Length: 5 dtype: bool (expression)
----------------------------------
0   True
1  False
2   True
3  False
4  False
```

equals (*y*)

Tests if strings x and y are the same

Returns a boolean expression

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.equals(df.text)
Expression = str_equals(text, text)
Length: 5 dtype: bool (expression)
----------------------------------
0  True
1  True
2  True
3  True
4  True
```

```
>>> df.text.str.equals('our')
Expression = str_equals(text, 'our')
Length: 5 dtype: bool (expression)
----------------------------------
```

(continues on next page)

(continued from previous page)

```
0  False
1  False
2  False
3   True
4  False
```

find (*sub*, *start=0*, *end=None*)

Returns the lowest indices in each string in a column, where the provided substring is fully contained between within a sample. If the substring is not found, -1 is returned.

Parameters

- **sub** (*str*) – A substring to be found in the samples
- **start** (*int*) –
- **end** (*int*) –

Returns an expression containing the lowest indices specifying the start of the substring.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.find(sub="et")
Expression = str_find(text, sub='et')
Length: 5 dtype: int64 (expression)
-----------------------------------
0   3
1   7
2  -1
3  -1
4  -1
```

get (*i*)

Extract a character from each sample at the specified position from a string column. Note that if the specified position is out of bound of the string sample, this method returns '', while pandas retunrs nan.

Parameters **i** (*int*) – The index location, at which to extract the character.

Returns an expression containing the extracted characters.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
```

(continues on next page)

(continued from previous page)

```
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.get(5)
Expression = str_get(text, 5)
Length: 5 dtype: str (expression)
---------------------------------
0    h
1    p
2    m
3
4
```

index (*sub*, *start=0*, *end=None*)

Returns the lowest indices in each string in a column, where the provided substring is fully contained between within a sample. If the substring is not found, -1 is returned. It is the same as *str.find*.

Parameters

- **sub** (*str*) – A substring to be found in the samples
- **start** (*int*) –
- **end** (*int*) –

Returns an expression containing the lowest indices specifying the start of the substring.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.index(sub="et")
Expression = str_find(text, sub='et')
Length: 5 dtype: int64 (expression)
-----------------------------------
0   3
1   7
2  -1
3  -1
4  -1
```

isalnum ()

Check if all characters in a string sample are alphanumeric.

Returns an expression evaluated to True if a sample contains only alphanumeric characters, otherwise False.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.isalnum()
Expression = str_isalnum(text)
Length: 5 dtype: bool (expression)
----------------------------------
0   True
1  False
2  False
3   True
4  False
```

isalpha()

Check if all characters in a string sample are alphabetic.

Returns an expression evaluated to True if a sample contains only alphabetic characters, otherwise False.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.isalpha()
Expression = str_isalpha(text)
Length: 5 dtype: bool (expression)
----------------------------------
0   True
1  False
2  False
3   True
4  False
```

isdigit()

Check if all characters in a string sample are digits.

Returns an expression evaluated to True if a sample contains only digits, otherwise False.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', '6']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  6
```

```
>>> df.text.str.isdigit()
Expression = str_isdigit(text)
Length: 5 dtype: bool (expression)
----------------------------------
0  False
1  False
2  False
3  False
4   True
```

islower()

Check if all characters in a string sample are lowercase characters.

Returns an expression evaluated to True if a sample contains only lowercase characters, otherwise False.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.islower()
Expression = str_islower(text)
Length: 5 dtype: bool (expression)
----------------------------------
0  False
1   True
2   True
3   True
4   True
```

isspace()

Check if all characters in a string sample are whitespaces.

Returns an expression evaluated to True if a sample contains only whitespaces, otherwise False.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', '       ', ' ']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3
  4
```

```
>>> df.text.str.isspace()
Expression = str_isspace(text)
Length: 5 dtype: bool (expression)
----------------------------------
0  False
1  False
2  False
3   True
4   True
```

isupper()

Check if all characters in a string sample are lowercase characters.

Returns an expression evaluated to True if a sample contains only lowercase characters, otherwise False.

Example:

```
>>> import vaex
>>> text = ['SOMETHING', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  SOMETHING
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.isupper()
Expression = str_isupper(text)
Length: 5 dtype: bool (expression)
----------------------------------
0   True
1  False
2  False
3  False
4  False
```

join(*sep*)

Same as find (difference with pandas is that it does not raise a ValueError)

len()

Returns the length of a string sample.

Returns an expression contains the length of each sample of a string column.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.len()
Expression = str_len(text)
Length: 5 dtype: int64 (expression)
-----------------------------------
0   9
1  11
2   9
3   3
4   4
```

ljust (*width, fillchar=' '*)

Fills the right side of string samples with a specified character such that the strings are right-hand justified.

Parameters

- **width** (*int*) – The minimal width of the strings.
- **fillchar** (*str*) – The character used for filling.

Returns an expression containing the filled strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.ljust(width=10, fillchar='!')
Expression = str_ljust(text, width=10, fillchar='!')
Length: 5 dtype: str (expression)
---------------------------------
0   Something!
1  very pretty
2   is coming!
3   our!!!!!!!
4   way.!!!!!!
```

lower ()

Converts string samples to lower case.

Returns an expression containing the converted strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.lower()
Expression = str_lower(text)
Length: 5 dtype: str (expression)
---------------------------------
0    something
1  very pretty
2    is coming
3          our
4         way.
```

lstrip(*to_strip=None*)

Remove leading characters from a string sample.

Parameters **to_strip** (*str*) – The string to be removed

Returns an expression containing the modified string column.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.lstrip(to_strip='very ')
Expression = str_lstrip(text, to_strip='very ')
Length: 5 dtype: str (expression)
---------------------------------
0  Something
1     pretty
2  is coming
3        our
4       way.
```

match(*pattern*)

Check if a string sample matches a given regular expression.

Parameters **pattern** (*str*) – a string or regex to match to a string sample.

Returns an expression which is evaluated to True if a match is found, False otherwise.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.match(pattern='our')
Expression = str_match(text, pattern='our')
Length: 5 dtype: bool (expression)
----------------------------------
0  False
1  False
2  False
3   True
4  False
```

pad(*width*, *side='left'*, *fillchar=' '*)
Pad strings in a given column.

Parameters

- **width** (*int*) – The total width of the string
- **side** (*str*) – If 'left' than pad on the left, if 'right' than pad on the right side the string.
- **fillchar** (*str*) – The character used for padding.

Returns an expression containing the padded strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.pad(width=10, side='left', fillchar='!')
Expression = str_pad(text, width=10, side='left', fillchar='!')
Length: 5 dtype: str (expression)
---------------------------------
0   !Something
1  very pretty
2   !is coming
3   !!!!!!!our
4   !!!!!!way.
```

repeat (*repeats*)
Duplicate each string in a column.

Parameters **repeats** (*int*) – number of times each string sample is to be duplicated.

Returns an expression containing the duplicated strings

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.repeat(3)
Expression = str_repeat(text, 3)
Length: 5 dtype: str (expression)
---------------------------------
0        SomethingSomethingSomething
1  very prettyvery prettyvery pretty
2        is comingis comingis coming
3                          ourourour
4                       way.way.way.
```

replace (*pat*, *repl*, *n=-1*, *flags=0*, *regex=False*)
Replace occurences of a pattern/regex in a column with some other string.

Parameters

- **pattern** (*str*) – string or a regex pattern
- **replace** (*str*) – a replacement string
- **n** (*int*) – number of replacements to be made from the start. If -1 make all replacements.
- **flags** (*int*) – ??
- **regex** (*bool*) – If True, . . . ?

Returns an expression containing the string replacements.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.replace(pat='et', repl='__')
Expression = str_replace(text, pat='et', repl='__')
Length: 5 dtype: str (expression)
---------------------------------
0    Som__hing
1  very pr__ty
2    is coming
3          our
4         way.
```

rfind (*sub*, *start=0*, *end=None*)

Returns the highest indices in each string in a column, where the provided substring is fully contained between within a sample. If the substring is not found, -1 is returned.

Parameters

- **sub** (*str*) – A substring to be found in the samples
- **start** (*int*) –
- **end** (*int*) –

Returns an expression containing the highest indices specifying the start of the substring.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.rfind(sub="et")
Expression = str_rfind(text, sub='et')
Length: 5 dtype: int64 (expression)
-----------------------------------
0   3
1   7
2  -1
3  -1
4  -1
```

rindex (*sub*, *start=0*, *end=None*)

Returns the highest indices in each string in a column, where the provided substring is fully contained between within a sample. If the substring is not found, -1 is returned. Same as *str.rfind*.

Parameters

- **sub** (*str*) – A substring to be found in the samples
- **start** (*int*) –
- **end** (*int*) –

Returns an expression containing the highest indices specifying the start of the substring.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.rindex(sub="et")
Expression = str_rindex(text, sub='et')
Length: 5 dtype: int64 (expression)
-----------------------------------
0   3
1   7
2  -1
3  -1
4  -1
```

rjust (*width*, *fillchar=' '*)

Fills the left side of string samples with a specified character such that the strings are left-hand justified.

Parameters

- **width** (*int*) – The minimal width of the strings.
- **fillchar** (*str*) – The character used for filling.

Returns an expression containing the filled strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.rjust(width=10, fillchar='!')
Expression = str_rjust(text, width=10, fillchar='!')
Length: 5 dtype: str (expression)
---------------------------------
0   !Something
1  very pretty
2   !is coming
3   !!!!!!!our
4   !!!!!!way.
```

rstrip (*to_strip=None*)

Remove trailing characters from a string sample.

Parameters **to_strip** (*str*) – The string to be removed

Returns an expression containing the modified string column.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.rstrip(to_strip='ing')
Expression = str_rstrip(text, to_strip='ing')
Length: 5 dtype: str (expression)
---------------------------------
0       Someth
1  very pretty
2       is com
3          our
4         way.
```

slice(*start=0*, *stop=None*)

Slice substrings from each string element in a column.

Parameters

- **start** (*int*) – The start position for the slice operation.
- **end** (*int*) – The stop position for the slice operation.

Returns an expression containing the sliced substrings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.slice(start=2, stop=5)
Expression = str_pandas_slice(text, start=2, stop=5)
Length: 5 dtype: str (expression)
---------------------------------
0  met
1   ry
2   co
3    r
4   y.
```

startswith (*pat*)

Check if a start of a string matches a pattern.

Parameters pat (*str*) – A string pattern. Regular expressions are not supported.

Returns an expression which is evaluated to True if the pattern is found at the start of a string sample, False otherwise.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.startswith(pat='is')
Expression = str_startswith(text, pat='is')
Length: 5 dtype: bool (expression)
----------------------------------
0  False
1  False
2   True
3  False
4  False
```

strip (*to_strip=None*)

Removes leading and trailing characters.

Strips whitespaces (including new lines), or a set of specified characters from each string saple in a column, both from the left right sides.

Parameters

- **to_strip** (*str*) – The characters to be removed. All combinations of the characters will be removed. If None, it removes whitespaces.
- **returns** – an expression containing the modified string samples.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.strip(to_strip='very')
Expression = str_strip(text, to_strip='very')
Length: 5 dtype: str (expression)
```

(continues on next page)

(continued from previous page)

```
---------------------------------
0  Something
1      prett
2  is coming
3         ou
4       way.
```

title()

Converts all string samples to titlecase.

Returns an expression containing the converted strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.title()
Expression = str_title(text)
Length: 5 dtype: str (expression)
---------------------------------
0    Something
1  Very Pretty
2    Is Coming
3          Our
4         Way.
```

upper()

Converts all strings in a column to uppercase.

Returns an expression containing the converted strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.upper()
Expression = str_upper(text)
Length: 5 dtype: str (expression)
---------------------------------
```

(continues on next page)

(continued from previous page)

```
0    SOMETHING
1  VERY PRETTY
2    IS COMING
3          OUR
4         WAY.
```

zfill(*width*)

Pad strings in a column by prepanding "0" characters.

Parameters **width** (*int*) – The minimum length of the resulting string. Strings shorter less than *width* will be prepended with zeros.

Returns an expression containing the modified strings.

Example:

```
>>> import vaex
>>> text = ['Something', 'very pretty', 'is coming', 'our', 'way.']
>>> df = vaex.from_arrays(text=text)
>>> df
  #  text
  0  Something
  1  very pretty
  2  is coming
  3  our
  4  way.
```

```
>>> df.text.str.zfill(width=12)
Expression = str_zfill(text, width=12)
Length: 5 dtype: str (expression)
---------------------------------
0  000Something
1  0very pretty
2  000is coming
3  000000000our
4  00000000way.
```

6.3.2 String (pandas) operations

class vaex.expression.**StringOperationsPandas**(*expression*)

Bases: object

String operations using Pandas Series (much slower)

__init__(*expression*)

Initialize self. See help(type(self)) for accurate signature.

__weakref__

list of weak references to the object (if defined)

byte_length(****kwargs**)

Wrapper around pandas.Series.byte_length

capitalize(****kwargs**)

Wrapper around pandas.Series.capitalize

cat(****kwargs**)

Wrapper around pandas.Series.cat

center(***kwargs*)
Wrapper around pandas.Series.center

contains(***kwargs*)
Wrapper around pandas.Series.contains

count(***kwargs*)
Wrapper around pandas.Series.count

endswith(***kwargs*)
Wrapper around pandas.Series.endswith

equals(***kwargs*)
Wrapper around pandas.Series.equals

find(***kwargs*)
Wrapper around pandas.Series.find

get(***kwargs*)
Wrapper around pandas.Series.get

index(***kwargs*)
Wrapper around pandas.Series.index

isalnum(***kwargs*)
Wrapper around pandas.Series.isalnum

isalpha(***kwargs*)
Wrapper around pandas.Series.isalpha

isdigit(***kwargs*)
Wrapper around pandas.Series.isdigit

islower(***kwargs*)
Wrapper around pandas.Series.islower

isspace(***kwargs*)
Wrapper around pandas.Series.isspace

isupper(***kwargs*)
Wrapper around pandas.Series.isupper

join(***kwargs*)
Wrapper around pandas.Series.join

len(***kwargs*)
Wrapper around pandas.Series.len

ljust(***kwargs*)
Wrapper around pandas.Series.ljust

lower(***kwargs*)
Wrapper around pandas.Series.lower

lstrip(***kwargs*)
Wrapper around pandas.Series.lstrip

match(***kwargs*)
Wrapper around pandas.Series.match

pad(***kwargs*)
Wrapper around pandas.Series.pad

repeat (***kwargs*)
Wrapper around pandas.Series.repeat

replace (***kwargs*)
Wrapper around pandas.Series.replace

rfind (***kwargs*)
Wrapper around pandas.Series.rfind

rindex (***kwargs*)
Wrapper around pandas.Series.rindex

rjust (***kwargs*)
Wrapper around pandas.Series.rjust

rstrip (***kwargs*)
Wrapper around pandas.Series.rstrip

slice (***kwargs*)
Wrapper around pandas.Series.slice

split (***kwargs*)
Wrapper around pandas.Series.split

startswith (***kwargs*)
Wrapper around pandas.Series.startswith

strip (***kwargs*)
Wrapper around pandas.Series.strip

title (***kwargs*)
Wrapper around pandas.Series.title

upper (***kwargs*)
Wrapper around pandas.Series.upper

zfill (***kwargs*)
Wrapper around pandas.Series.zfill

6.3.3 Date/time operations

class vaex.expression.**DateTime** (*expression*)
Bases: `object`

DateTime operations

Usually accessed using e.g. *df.birthday.dt.dayofweek*

__init__ (*expression*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

day
Extracts the day from a datetime sample.

Returns an expression containing the day extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.day
Expression = dt_day(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0  12
1  11
2  12
```

day_name

Returns the day names of a datetime sample in English.

Returns an expression containing the day names extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.day_name
Expression = dt_day_name(date)
Length: 3 dtype: str (expression)
---------------------------------
0    Monday
1  Thursday
2  Thursday
```

dayofweek

Obtain the day of the week with Monday=0 and Sunday=6

Returns an expression containing the day of week.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
```

(continues on next page)

(continued from previous page)

```
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.dayofweek
Expression = dt_dayofweek(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0  0
1  3
2  3
```

dayofyear

The ordinal day of the year.

Returns an expression containing the ordinal day of the year.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.dayofyear
Expression = dt_dayofyear(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0  285
1   42
2  316
```

hour

Extracts the hour out of a datetime samples.

Returns an expression containing the hour extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.hour
Expression = dt_hour(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0    3
1   10
2   11
```

is_leap_year

Check whether a year is a leap year.

Returns an expression which evaluates to True if a year is a leap year, and to False otherwise.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.is_leap_year
Expression = dt_is_leap_year(date)
Length: 3 dtype: bool (expression)
----------------------------------
0  False
1   True
2  False
```

minute

Extracts the minute out of a datetime samples.

Returns an expression containing the minute extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.minute
Expression = dt_minute(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0   31
```

(continues on next page)

(continued from previous page)

```
1  17
2  34
```

month

Extracts the month out of a datetime sample.

Returns an expression containing the month extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.month
Expression = dt_month(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0  10
1   2
2  11
```

month_name

Returns the month names of a datetime sample in English.

Returns an expression containing the month names extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.month_name
Expression = dt_month_name(date)
Length: 3 dtype: str (expression)
---------------------------------
0   October
1  February
2  November
```

second

Extracts the second out of a datetime samples.

Returns an expression containing the second extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.second
Expression = dt_second(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0   0
1  34
2  22
```

weekofyear

Returns the week ordinal of the year.

Returns an expression containing the week ordinal of the year, extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.weekofyear
Expression = dt_weekofyear(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0  42
1   6
2  46
```

year

Extracts the year out of a datetime sample.

Returns an expression containing the year extracted from a datetime column.

Example:

```
>>> import vaex
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> date = np.array(['2009-10-12T03:31:00', '2016-02-11T10:17:34', '2015-11-
↪12T11:34:22'], dtype=np.datetime64)
>>> df = vaex.from_arrays(date=date)
>>> df
  #  date
  0  2009-10-12 03:31:00
  1  2016-02-11 10:17:34
  2  2015-11-12 11:34:22
```

```
>>> df.date.dt.year
Expression = dt_year(date)
Length: 3 dtype: int64 (expression)
-----------------------------------
0  2009
1  2016
2  2015
```

6.3.4 Timedelta operations

class vaex.expression.**TimeDelta**(*expression*)

Bases: object

TimeDelta operations

Usually accessed using e.g. *df.delay.td.days*

__init__(*expression*)

Initialize self. See help(type(self)) for accurate signature.

__weakref__

list of weak references to the object (if defined)

days

Number of days in each timedelta sample.

Returns an expression containing the number of days in a timedelta sample.

Example:

```
>>> import vaex
>>> import numpy as np
>>> delta = np.array([17658720110,   11047049384039, 40712636304958, -
↪18161254954], dtype='timedelta64[s]')
>>> df = vaex.from_arrays(delta=delta)
>>> df
  #  delta
  0  204 days +9:12:00
  1  1 days +6:41:10
  2  471 days +5:03:56
  3  -22 days +23:31:15
```

```
>>> df.delta.td.days
Expression = td_days(delta)
Length: 4 dtype: int64 (expression)
-----------------------------------
0  204
```

(continues on next page)

(continued from previous page)

```
1    1
2  471
3  -22
```

microseconds

Number of microseconds (>= 0 and less than 1 second) in each timedelta sample.

Returns an expression containing the number of microseconds in a timedelta sample.

Example:

```
>>> import vaex
>>> import numpy as np
>>> delta = np.array([17658720110,   11047049384039, 40712636304958, -
↪18161254954], dtype='timedelta64[s]')
>>> df = vaex.from_arrays(delta=delta)
>>> df
  #  delta
  0  204 days +9:12:00
  1  1 days +6:41:10
  2  471 days +5:03:56
  3  -22 days +23:31:15
```

```
>>> df.delta.td.microseconds
Expression = td_microseconds(delta)
Length: 4 dtype: int64 (expression)
-----------------------------------
0  290448
1  978582
2   19583
3  709551
```

nanoseconds

Number of nanoseconds (>= 0 and less than 1 microsecond) in each timedelta sample.

Returns an expression containing the number of nanoseconds in a timedelta sample.

Example:

```
>>> import vaex
>>> import numpy as np
>>> delta = np.array([17658720110,   11047049384039, 40712636304958, -
↪18161254954], dtype='timedelta64[s]')
>>> df = vaex.from_arrays(delta=delta)
>>> df
  #  delta
  0  204 days +9:12:00
  1  1 days +6:41:10
  2  471 days +5:03:56
  3  -22 days +23:31:15
```

```
>>> df.delta.td.nanoseconds
Expression = td_nanoseconds(delta)
Length: 4 dtype: int64 (expression)
-----------------------------------
0  384
1   16
```

(continues on next page)

(continued from previous page)

```
2   488
3   616
```

seconds

Number of seconds (>= 0 and less than 1 day) in each timedelta sample.

Returns an expression containing the number of seconds in a timedelta sample.

Example:

```
>>> import vaex
>>> import numpy as np
>>> delta = np.array([17658720110,    11047049384039, 40712636304958, -
↪18161254954], dtype='timedelta64[s]')
>>> df = vaex.from_arrays(delta=delta)
>>> df
  #  delta
  0  204 days +9:12:00
  1  1 days +6:41:10
  2  471 days +5:03:56
  3  -22 days +23:31:15
```

```
>>> df.delta.td.seconds
Expression = td_seconds(delta)
Length: 4 dtype: int64 (expression)
-----------------------------------
0  30436
1  39086
2  28681
3  23519
```

total_seconds()

Total duration of each timedelta sample expressed in seconds.

Returns an expression containing the total number of seconds in a timedelta sample.

Example: >>> import vaex >>> import numpy as np >>> delta = np.array([17658720110, 11047049384039, 40712636304958, -18161254954], dtype='timedelta64[s]') >>> df = vaex.from_arrays(delta=delta) >>> df

delta 0 204 days +9:12:00 1 1 days +6:41:10 2 471 days +5:03:56 3 -22 days +23:31:15

```
>>> df.delta.td.total_seconds()
Expression = td_total_seconds(delta)
Length: 4 dtype: float64 (expression)
-------------------------------------
0  -7.88024e+08
1  -2.55032e+09
2   6.72134e+08
3   2.85489e+08
```

6.3.5 Geo operations

class vaex.geo.**DataFrameAccessorGeo**(*df*)

Bases: object

Geometry/geographic helper methods

Example:

```
>>> df_xyz = df.geo.spherical2cartesian(df.longitude, df.latitude, df.distance)
>>> df_xyz.x.mean()
```

__init__(*df*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

bearing(*lon1, lat1, lon2, lat2, bearing='bearing', inplace=False*)
Calculates a bearing, based on http://www.movable-type.co.uk/scripts/latlong.html

cartesian2spherical(*x='x', y='y', z='z', alpha='l', delta='b', distance='distance', radians=False, center=None, center_name='solar_position', inplace=False*)
Convert cartesian to spherical coordinates.

Parameters

- **x** –
- **y** –
- **z** –
- **alpha** –
- **delta** – name for polar angle, ranges from -90 to 90 (or -pi to pi when radians is True).
- **distance** –
- **radians** –
- **center** –
- **center_name** –

Returns

cartesian_to_polar(*x='x', y='y', radius_out='r_polar', azimuth_out='phi_polar', propagate_uncertainties=False, radians=False, inplace=False*)
Convert cartesian to polar coordinates

Parameters

- **x** – expression for x
- **y** – expression for y
- **radius_out** – name for the virtual column for the radius
- **azimuth_out** – name for the virtual column for the azimuth angle
- **propagate_uncertainties** – {propagate_uncertainties}
- **radians** – if True, azimuth is in radians, defaults to degrees

Returns

project_aitoff(*alpha, delta, x, y, radians=True, inplace=False*)
Add aitoff (https://en.wikipedia.org/wiki/Aitoff_projection) projection

Parameters

- **alpha** – azimuth angle
- **delta** – polar angle

- **x** – output name for x coordinate
- **y** – output name for y coordinate
- **radians** – input and output in radians (True), or degrees (False)

Returns

project_gnomic(*alpha*, *delta*, *alpha0=0*, *delta0=0*, *x='x'*, *y='y'*, *radians=False*, *postfix="*, *inplace=False*)
Adds a gnomic projection to the DataFrame

rotation_2d(*x*, *y*, *xnew*, *ynew*, *angle_degrees*, *propagate_uncertainties=False*, *inplace=False*)
Rotation in 2d.

Parameters

- **x** (*str*) – Name/expression of x column
- **y** (*str*) – idem for y
- **xnew** (*str*) – name of transformed x column
- **ynew** (*str*) –
- **angle_degrees** (*float*) – rotation in degrees, anti clockwise

Returns

spherical2cartesian(*alpha, delta, distance, xname='x', yname='y', zname='z', propagate_uncertainties=False, center=[0, 0, 0], radians=False, inplace=False*)
Convert spherical to cartesian coordinates.

Parameters

- **alpha** –
- **delta** – polar angle, ranging from the -90 (south pole) to 90 (north pole)
- **distance** – radial distance, determines the units of x, y and z
- **xname** –
- **yname** –
- **zname** –
- **propagate_uncertainties** – {propagate_uncertainties}
- **center** –
- **radians** –

Returns New dataframe (in inplace is False) with new x,y,z columns

velocity_cartesian2polar(*x='x'*, *y='y'*, *vx='vx'*, *radius_polar=None*, *vy='vy'*, *vr_out='vr_polar'*, *vazimuth_out='vphi_polar'*, *propagate_uncertainties=False*, *inplace=False*)
Convert cartesian to polar velocities.

Parameters

- **x** –
- **y** –
- **vx** –

- **radius_polar** – Optional expression for the radius, may lead to a better performance when given.
- **vy** –
- **vr_out** –
- **vazimuth_out** –
- **propagate_uncertainties** – {propagate_uncertainties}

Returns

velocity_cartesian2spherical (*x='x'*, *y='y'*, *z='z'*, *vx='vx'*, *vy='vy'*, *vz='vz'*, *vr='vr'*, *vlong='vlong'*, *vlat='vlat'*, *distance=None*, *inplace=False*)

Convert velocities from a cartesian to a spherical coordinate system

TODO: uncertainty propagation

Parameters

- **x** – name of x column (input)
- **y** – y
- **z** – z
- **vx** – vx
- **vy** – vy
- **vz** – vz
- **vr** – name of the column for the radial velocity in the r direction (output)
- **vlong** – name of the column for the velocity component in the longitude direction (output)
- **vlat** – name of the column for the velocity component in the latitude direction, positive points to the north pole (output)
- **distance** – Expression for distance, if not given defaults to sqrt(x**2+y**2+z**2), but if this column already exists, passing this expression may lead to a better performance

Returns

velocity_polar2cartesian (*x='x'*, *y='y'*, *azimuth=None*, *vr='vr_polar'*, *vazimuth='vphi_polar'*, *vx_out='vx'*, *vy_out='vy'*, *propagate_uncertainties=False*, *inplace=False*)

Convert cylindrical polar velocities to Cartesian.

Parameters

- **x** –
- **y** –
- **azimuth** – Optional expression for the azimuth in degrees , may lead to a better performance when given.
- **vr** –
- **vazimuth** –
- **vx_out** –
- **vy_out** –
- **propagate_uncertainties** – {propagate_uncertainties}

6.3.6 GraphQL operations

class vaex.graphql.**DataFrameAccessorGraphQL**(*df*)

Bases: object

Exposes a GraphQL layer to a DataFrame

See the GraphQL example for more usage.

The easiest way to learn to use the GraphQL language/vaex interface is to launch a server, and play with the GraphiQL graphical interface, its autocomplete, and the schema explorer.

We try to stay close to the Hasura API: https://docs.hasura.io/1.0/graphql/manual/api-reference/graphql-api/query.html

__init__(*df*)

Initialize self. See help(type(self)) for accurate signature.

__weakref__

list of weak references to the object (if defined)

execute(**args*, ***kwargs*)

Creates a schema, and execute the query (first argument)

query(*name='df'*)

Creates a graphene query object exposing this DataFrame named *name*

schema(*name='df'*, *auto_camelcase=False*, ***kwargs*)

Creates a graphene schema for this DataFrame

serve(*port=9001*, *address=''*, *name='df'*, *verbose=True*)

Serve the DataFrame via a http server

6.4 Machine learning with vaex.ml

6.4.1 Clustering

class vaex.ml.cluster.**KMeans**(*cluster_centers=traitlets.Undefined*, *features=traitlets.Undefined*, *inertia=None*, *init='random'*, *max_iter=300*, *n_clusters=2*, *n_init=1*, *prediction_label='prediction_kmeans'*, *random_state=None*, *verbose=False*)

Bases: vaex.ml.state.HasState

The KMeans clustering algorithm.

Example:

```
>>> import vaex.ml
>>> import vaex.ml.cluster
>>> df = vaex.ml.datasets.load_iris()
>>> features = ['sepal_width', 'petal_length', 'sepal_length', 'petal_width']
>>> cls = vaex.ml.cluster.KMeans(n_clusters=3, features=features, init='random', max_iter=10)
>>> cls.fit(df)
>>> df = cls.transform(df)
>>> df.head(5)
 #    sepal_width    petal_length    sepal_length    petal_width    class_    prediction_kmeans
 0              3             4.2             5.9            1.5         1                    2
```

(continues on next page)

(continued from previous page)

```
1                3                  4.6                6.1               1.4            1         ␣
↪                2
2                2.9                4.6                6.6               1.3            1         ␣
↪                2
3                3.3                5.7                6.7               2.1            2         ␣
↪                0
4                4.2                1.4                5.5               0.2            0         ␣
↪                1
```

Parameters

- **cluster_centers** – Coordinates of cluster centers.
- **features** – List of features to cluster.
- **inertia** – Sum of squared distances of samples to their closest cluster center.
- **init** – Method for initializing the centroids.
- **max_iter** – Maximum number of iterations of the KMeans algorithm for a single run.
- **n_clusters** – Number of clusters to form.
- **n_init** – Number of centroid initializations. The KMeans algorithm will be run for each initialization, and the final results will be the best output of the n_init consecutive runs in terms of inertia.
- **prediction_label** – The name of the virtual column that houses the cluster labels for each point.
- **random_state** – Random number generation for centroid initialization. If an int is specified, the randomness becomes deterministic.
- **verbose** – If True, enable verbosity mode.

fit (*dataframe*)
Fit the KMeans model to the dataframe.

Parameters **dataframe** – A vaex DataFrame.

transform (*dataframe*)
Label a DataFrame with a fitted KMeans model.

Parameters **dataframe** – A vaex DataFrame.

Returns copy A shallow copy of the DataFrame that includes the cluster labels.

Return type *DataFrame*

6.4.2 PCA

class vaex.ml.transformations.**PCA** (*features=traitlets.Undefined, n_components=0, prefix='PCA_', progress=False*)

Bases: vaex.ml.transformations.Transformer

Transform a set of features using a Principal Component Analysis.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(x=[2,5,7,2,15], y=[-2,3,0,0,10])
>>> df
 #   x   y
 0   2   -2
 1   5   3
 2   7   0
 3   2   0
 4   15  10
>>> pca = vaex.ml.PCA(n_components=2, features=['x', 'y'])
>>> pca.fit_transform(df)
 #    x    y       PCA_0      PCA_1
 0    2   -2     5.92532    0.413011
 1    5    3     0.380494  -1.39112
 2    7    0     0.840049   2.18502
 3    2    0     4.61287   -1.09612
 4   15   10   -11.7587    -0.110794
```

Parameters

- **features** – List of features to transform.
- **n_components** – Number of components to retain. If None, all the components will be retained.
- **prefix** – Prefix for the names of the transformed features.
- **progress** – If True, display a progressbar of the PCA fitting process.

fit (*df*)

Fit the PCA model to the DataFrame.

Parameters df – A vaex DataFrame.

transform (*df*, *n_components=None*)

Apply the PCA transformation to the DataFrame.

Parameters

- **df** – A vaex DataFrame.
- **n_components** – The number of PCA components to retain.

Return copy A shallow copy of the DataFrame that includes the PCA components.

Return type *DataFrame*

6.4.3 Encoders

class vaex.ml.transformations.**LabelEncoder** (*allow_unseen=False*, *features=traitlets.Undefined*, *prefix='label_encoded_'*)

Bases: vaex.ml.transformations.Transformer

Encode categorical columns with integer values between 0 and num_classes-1.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(color=['red', 'green', 'green', 'blue', 'red'])
>>> df
 #  color
 0  red
 1  green
 2  green
 3  blue
 4  red
>>> encoder = vaex.ml.LabelEncoder(features=['color'])
>>> encoder.fit_transform(df)
 #  color      label_encoded_color
 0  red                          2
 1  green                        1
 2  green                        1
 3  blue                         0
 4  red                          2
```

Parameters

- **`allow_unseen`** – If True, unseen values will be encoded with -1, otherwise an error is raised
- **`features`** – List of features to transform.
- **`prefix`** – Prefix for the names of the transformed features.

`fit`(*df*)

Fit LabelEncoder to the DataFrame.

Parameters **`df`** – A vaex DataFrame.

`transform`(*df*)

Transform a DataFrame with a fitted LabelEncoder.

Parameters **`df`** – A vaex DataFrame.

Returns: :return copy: A shallow copy of the DataFrame that includes the encodings. :rtype: DataFrame

class `vaex.ml.transformations.`**`OneHotEncoder`**(*features=traitlets.Undefined*, *one=1*, *prefix=''*, *zero=0*)

Bases: `vaex.ml.transformations.Transformer`

Encode categorical columns according ot the One-Hot scheme.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(color=['red', 'green', 'green', 'blue', 'red'])
>>> df
 #  color
 0  red
 1  green
 2  green
 3  blue
 4  red
>>> encoder = vaex.ml.OneHotEncoder(features=['color'])
>>> encoder.fit_transform(df)
 #  color      color_blue    color_green    color_red
 0  red                 0              0            1
```

(continues on next page)

(continued from previous page)

```
1  green              0            1            0
2  green              0            1            0
3  blue               1            0            0
4  red                0            0            1
```

Parameters

- **features** – List of features to transform.
- **one** – Value to encode when a category is present.
- **prefix** – Prefix for the names of the transformed features.
- **zero** – Value to encode when category is absent.

fit (*df*)

Fit OneHotEncoder to the DataFrame.

Parameters df – A vaex DataFrame.

transform (*df*)

Transform a DataFrame with a fitted OneHotEncoder.

Parameters df – A vaex DataFrame.

Returns A shallow copy of the DataFrame that includes the encodings.

Return type *DataFrame*

class vaex.ml.transformations.**StandardScaler** (*features=traitlets.Undefined, prefix='standard_scaled_', with_mean=True, with_std=True*)

Bases: vaex.ml.transformations.Transformer

Standardize features by removing thir mean and scaling them to unit variance.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(x=[2,5,7,2,15], y=[-2,3,0,0,10])
>>> df
 #    x    y
 0    2   -2
 1    5    3
 2    7    0
 3    2    0
 4   15   10
>>> scaler = vaex.ml.StandardScaler(features=['x', 'y'])
>>> scaler.fit_transform(df)
 #    x    y    standard_scaled_x    standard_scaled_y
 0    2   -2            -0.876523            -0.996616
 1    5    3            -0.250435             0.189832
 2    7    0             0.166957            -0.522037
 3    2    0            -0.876523            -0.522037
 4   15   10             1.83652              1.85086
```

Parameters

- **features** – List of features to transform.
- **prefix** – Prefix for the names of the transformed features.

- **with_mean** – If True, remove the mean from each feature.
- **with_std** – If True, scale each feature to unit variance.

fit (*df*)
Fit StandardScaler to the DataFrame.

Parameters df – A vaex DataFrame.

transform (*df*)
Transform a DataFrame with a fitted StandardScaler.

Parameters df – A vaex DataFrame.

Returns copy a shallow copy of the DataFrame that includes the scaled features.

Return type *DataFrame*

class vaex.ml.transformations.**MinMaxScaler** (*feature_range=traitlets.Undefined, features=traitlets.Undefined, prefix='minmax_scaled_'*)

Bases: vaex.ml.transformations.Transformer

Will scale a set of features to a given range.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(x=[2,5,7,2,15], y=[-2,3,0,0,10])
>>> df
 #    x    y
 0    2   -2
 1    5    3
 2    7    0
 3    2    0
 4   15   10
>>> scaler = vaex.ml.MinMaxScaler(features=['x', 'y'])
>>> scaler.fit_transform(df)
 #    x    y    minmax_scaled_x    minmax_scaled_y
 0    2   -2           0                  0
 1    5    3           0.230769           0.416667
 2    7    0           0.384615           0.166667
 3    2    0           0                  0.166667
 4   15   10           1                  1
```

Parameters

- **feature_range** – The range the features are scaled to.
- **features** – List of features to transform.
- **prefix** – Prefix for the names of the transformed features.

fit (*df*)
Fit MinMaxScaler to the DataFrame.

Parameters df – A vaex DataFrame.

transform (*df*)
Transform a DataFrame with a fitted MinMaxScaler.

Parameters df – A vaex DataFrame.

Return copy a shallow copy of the DataFrame that includes the scaled features.

Return type *DataFrame*

class vaex.ml.transformations.**MaxAbsScaler**(*features=traitlets.Undefined, prefix='absmax_scaled_'*)

Bases: vaex.ml.transformations.Transformer

Scale features by their maximum absolute value.

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(x=[2,5,7,2,15], y=[-2,3,0,0,10])
>>> df
 #    x    y
 0    2   -2
 1    5    3
 2    7    0
 3    2    0
 4   15   10
>>> scaler = vaex.ml.MaxAbsScaler(features=['x', 'y'])
>>> scaler.fit_transform(df)
 #    x    y    absmax_scaled_x    absmax_scaled_y
 0    2   -2           0.133333               -0.2
 1    5    3           0.333333                0.3
 2    7    0           0.466667                0
 3    2    0           0.133333                0
 4   15   10           1                       1
```

Parameters

- **features** – List of features to transform.
- **prefix** – Prefix for the names of the transformed features.

fit(*df*)

Fit MinMaxScaler to the DataFrame.

Parameters df – A vaex DataFrame.

transform(*df*)

Transform a DataFrame with a fitted MaxAbsScaler.

Parameters df – A vaex DataFrame.

Return copy a shallow copy of the DataFrame that includes the scaled features.

Return type *DataFrame*

class vaex.ml.transformations.**RobustScaler**(*features=traitlets.Undefined, percentile_range=traitlets.Undefined, prefix='robust_scaled_', with_centering=True, with_scaling=True*)

Bases: vaex.ml.transformations.Transformer

The RobustScaler removes the median and scales the data according to a given percentile range. By default, the scaling is done between the 25th and the 75th percentile. Centering and scaling happens independently for each feature (column).

Example:

```
>>> import vaex
>>> df = vaex.from_arrays(x=[2,5,7,2,15], y=[-2,3,0,0,10])
>>> df
 #    x    y
 0    2   -2
 1    5    3
 2    7    0
 3    2    0
 4   15   10
>>> scaler = vaex.ml.MaxAbsScaler(features=['x', 'y'])
>>> scaler.fit_transform(df)
 #    x    y    robust_scaled_x    robust_scaled_y
 0    2   -2       -0.333686            -0.266302
 1    5    3       -0.000596934          0.399453
 2    7    0        0.221462             0
 3    2    0       -0.333686             0
 4   15   10        1.1097               1.33151
```

Parameters

- **features** – List of features to transform.
- **percentile_range** – The percentile range to which to scale each feature to.
- **prefix** – Prefix for the names of the transformed features.
- **with_centering** – If True, remove the median.
- **with_scaling** – If True, scale each feature between the specified percentile range.

fit (*df*)

Fit RobustScaler to the DataFrame.

Parameters df – A vaex DataFrame.

transform (*df*)

Transform a DataFrame with a fitted RobustScaler.

Parameters df – A vaex DataFrame.

Returns copy a shallow copy of the DataFrame that includes the scaled features.

Return type *DataFrame*

6.4.4 Boosted trees

class vaex.ml.lightgbm.**LightGBMModel** (*features=traitlets.Undefined, num_boost_round=0, params=traitlets.Undefined, prediction_name='lightgbm_prediction'*)

Bases: vaex.ml.state.HasState

The LightGBM algorithm.

This class provides an interface to the LightGBM algorithm, with some optimizations for better memory efficiency when training large datasets. The algorithm itself is not modified at all.

LightGBM is a fast gradient boosting algorithm based on decision trees and is mainly used for classification, regression and ranking tasks. It is under the umbrella of the Distributed Machine Learning Toolkit (DMTK) project of Microsoft. For more information, please visit https://github.com/Microsoft/LightGBM/.

Example:

```
>>> import vaex.ml
>>> import vaex.ml.lightgbm
>>> df = vaex.ml.datasets.load_iris()
>>> features = ['sepal_width', 'petal_length', 'sepal_length', 'petal_width']
>>> df_train, df_test = vaex.ml.train_test_split(df)
>>> params = {
    'boosting': 'gbdt',
    'max_depth': 5,
    'learning_rate': 0.1,
    'application': 'multiclass',
    'num_class': 3,
    'subsample': 0.80,
    'colsample_bytree': 0.80}
>>> booster = vaex.ml.lightgbm.LightGBMModel(features=features, num_boost_
↪round=100, params=params)
>>> booster.fit(df_train, 'class_')
>>> df_train = booster.transform(df_train)
>>> df_train.head(3)
 #    sepal_width    petal_length    sepal_length    petal_width    class_   ␣
↪lightgbm_prediction
 0              3             4.5             5.4            1.5         1    [0.
↪00165619 0.98097899 0.01736482]
 1            3.4             1.6             4.8            0.2         0    [9.
↪99803930e-01 1.17346471e-04 7.87235133e-05]
 2            3.1             4.9             6.9            1.5         1    [0.
↪00107541 0.9848717  0.01405289]
>>> df_test = booster.transform(df_test)
>>> df_test.head(3)
 #    sepal_width    petal_length    sepal_length    petal_width    class_   ␣
↪lightgbm_prediction
 0              3             4.2             5.9            1.5         1    [0.
↪00208904 0.9821348  0.01577616]
 1              3             4.6             6.1            1.4         1    [0.
↪00182039 0.98491357 0.01326604]
 2            2.9             4.6             6.6            1.3         1    [2.
↪50915444e-04 9.98431777e-01 1.31730785e-03]
```

Parameters

- **features** – List of features to use when fitting the LightGBMModel.
- **num_boost_round** – Number of boosting iterations.
- **params** – parameters to be passed on the to the LightGBM model.
- **prediction_name** – The name of the virtual column housing the predictions.

fit (*df*, *target*, *valid_sets=None*, *valid_names=None*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=None*, *copy=False*, ***kwargs*)
Fit the LightGBMModel to the DataFrame.

The model will train until the validation score stops improving. Validation score needs to improve at least every *early_stopping_rounds* rounds to continue training. Requires at least one validation DataFrame, metric specified. If there's more than one, will check all of them, but the training data is ignored anyway. If early stopping occurs, the model will add `best_iteration` field to the booster object. :param dict evals_result: A dictionary storing the evaluation results of all *valid_sets*. :param bool verbose_eval: Requires at least one item in *evals*. If *verbose_eval* is True then the evaluation metric on the validation set is printed at each boosting stage. :param bool copy: (default, False) If True, make an in memory copy of the data before passing it to LightGBMModel.

Parameters

- **df** – A vaex DataFrame.
- **target** – The name of the column containing the target variable.
- **valid_sets** (*list*) – A list of DataFrames to be used for validation.
- **valid_names** (*list*) – A list of strings to label the validation sets.
- **int** (*early_stopping_rounds*) – Activates early stopping.

predict (*df*, ***kwargs*)

Get an in-memory numpy array with the predictions of the LightGBMModel on a vaex DataFrame. This method accepts the key word arguments of the predict method from LightGBM.

Parameters **df** – A vaex DataFrame.

Returns A in-memory numpy array containing the LightGBMModel predictions.

Return type numpy.array

transform (*df*)

Transform a DataFrame such that it contains the predictions of the LightGBMModel in form of a virtual column.

Parameters **df** – A vaex DataFrame.

Return copy A shallow copy of the DataFrame that includes the LightGBMModel prediction as a virtual column.

Return type *DataFrame*

class vaex.ml.xgboost.**XGBoostModel** (*features=traitlets.Undefined*, *num_boost_round=0*, *params=traitlets.Undefined*, *prediction_name='xgboost_prediction'*)

Bases: vaex.ml.state.HasState

The XGBoost algorithm.

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. (https://github.com/dmlc/xgboost)

Example:

```
>>> import vaex
>>> import vaex.ml.xgboost
>>> df = vaex.ml.datasets.load_iris()
>>> features = ['sepal_width', 'petal_length', 'sepal_length', 'petal_width']
>>> df_train, df_test = vaex.ml.train_test_split(df)
>>> params = {
    'max_depth': 5,
    'learning_rate': 0.1,
    'objective': 'multi:softmax',
    'num_class': 3,
    'subsample': 0.80,
    'colsample_bytree': 0.80,
    'silent': 1}
>>> booster = vaex.ml.xgboost.XGBoostModel(features=features, num_boost_round=100,
↪ params=params)
```

(continues on next page)

(continued from previous page)

```
>>> booster.fit(df_train, 'class_')
>>> df_train = booster.transform(df_train)
>>> df_train.head(3)
#    sepal_length    sepal_width    petal_length    petal_width    class_  ␣
↪xgboost_prediction
0             5.4            3             4.5            1.5         1        ␣
↪             1
1             4.8            3.4           1.6            0.2         0        ␣
↪             0
2             6.9            3.1           4.9            1.5         1        ␣
↪             1
>>> df_test = booster.transform(df_test)
>>> df_test.head(3)
#    sepal_length    sepal_width    petal_length    petal_width    class_  ␣
↪xgboost_prediction
0             5.9            3             4.2            1.5         1        ␣
↪             1
1             6.1            3             4.6            1.4         1        ␣
↪             1
2             6.6            2.9           4.6            1.3         1        ␣
↪             1
```

Parameters

- **features** – List of features to use when fitting the XGBoostModel.
- **num_boost_round** – Number of boosting iterations.
- **params** – A dictionary of parameters to be passed on to the XGBoost model.
- **prediction_name** – The name of the virtual column housing the predictions.

fit (*df*, *target*, *evals=()*, *early_stopping_rounds=None*, *evals_result=None*, *verbose_eval=False*, ***kwargs*)

Fit the XGBoost model given a DataFrame.

This method accepts all key word arguments for the xgboost.train method.

Parameters

- **df** – A vaex DataFrame containing the training features.
- **target** – The column name of the target variable.
- **evals** – A list of pairs (DataFrame, string). List of items to be evaluated during training, this allows user to watch performance on the validation set.
- **early_stopping_rounds** (*int*) – Activates early stopping. Validation error needs to decrease at least every *early_stopping_rounds* round(s) to continue training. Requires at least one item in *evals*. If there's more than one, will use the last. Returns the model from the last iteration (not the best one).
- **evals_result** (*dict*) – A dictionary storing the evaluation results of all the items in *evals*.
- **verbose_eval** (*bool*) – Requires at least one item in *evals*. If *verbose_eval* is True then the evaluation metric on the validation set is printed at each boosting stage.

predict (*df*, ***kwargs*)

Provided a vaex DataFrame, get an in-memory numpy array with the predictions from the XGBoost model. This method accepts the key word arguments of the predict method from XGBoost.

Returns A in-memory numpy array containing the XGBoostModel predictions.

Return type numpy.array

transform(*df*)

Transform a DataFrame such that it contains the predictions of the XGBoostModel in form of a virtual column.

Parameters **df** – A vaex DataFrame. It should have the same columns as the DataFrame used to train the model.

Return copy A shallow copy of the DataFrame that includes the XGBoostModel prediction as a virtual column.

Return type *DataFrame*

6.4.5 Nearest neighbour

Annoy support is in the incubator phase, which means support may disappear in future versions

class vaex.ml.incubator.annoy.**ANNOYModel**(*features=traitlets.Undefined*, *metric='euclidean'*, *n_neighbours=10*, *n_trees=10*, *predcition_name='annoy_prediction'*, *prediction_name='annoy_prediction'*, *search_k=-1*)

Bases: vaex.ml.state.HasState

Parameters

- **features** – List of features to use.
- **metric** – Metric to use for distance calculations
- **n_neighbours** – Now many neighbours
- **n_trees** – Number of trees to build.
- **predcition_name** – Output column name for the neighbours when transforming a DataFrame
- **prediction_name** – Output column name for the neighbours when transforming a DataFrame
- **search_k** – Jovan?

Note: vaex.ml is under heavy development, consider this document as a sneak preview.

CHAPTER 7

Vaex-ml - Machine Learning

The `vaex.ml` package brings some machine learning algorithms to vaex. Install it by running `pip install vaex-ml`.

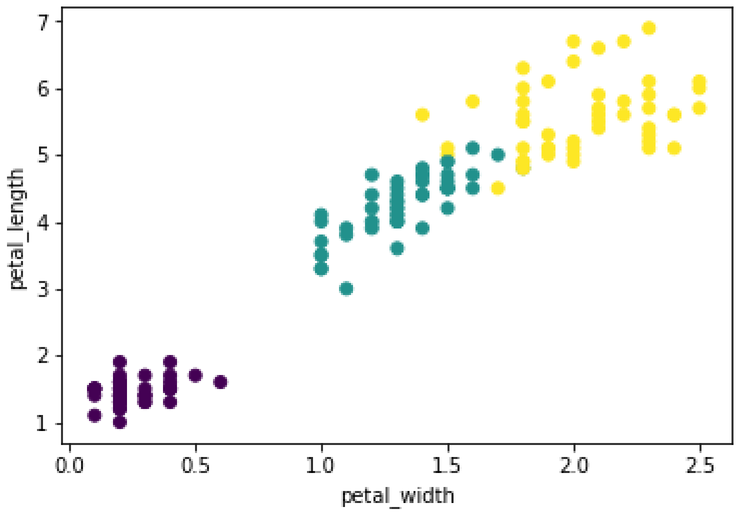
Vaex.ml stays close to the authoritative ML package: scikit-learn. We will first show two examples, KMeans and PCA, to see how they compare and differ, and what the gain is in performance.

[1]:
```
import vaex.ml.cluster
import vaex.ml.datasets
import numpy as np
%matplotlib inline
```

We use the well known iris flower dataset, a classical for machine learning.

[6]:
```
df = vaex.ml.datasets.load_iris()
df.scatter(df.petal_width, df.petal_length, c_expr=df.class_)
```

[6]:
```
<matplotlib.collections.PathCollection at 0x11fecbdd8>
```



[8]:
```
df
```

[8]:
```
#    sepal_width    petal_length    sepal_length    petal_width    class_    random_
↪index
0    3.0            4.2             5.9             1.5            1         114
1    3.0            4.6             6.1             1.4            1         74
2    2.9            4.6             6.6             1.3            1         37
3    3.3            5.7             6.7             2.1            2         116
4    4.2            1.4             5.5             0.2            0         61
...  ...            ...             ...             ...            ...       ...
145  3.4            1.4             5.2             0.2            0         119
146  3.8            1.6             5.1             0.2            0         15
147  2.6            4.0             5.8             1.2            1         22
148  3.8            1.7             5.7             0.3            0         144
149  2.9            4.3             6.2             1.3            1         102
```

7.1 KMeans

We use two features to do a KMeans, and roughly put the two features on the same scale by a simple division. We then construct a KMeans object, quite similar to what you would do in sklearn, and fit it.

[10]:
```
features = ['petal_width/2', 'petal_length/5']
init = [[0, 1/5], [1.2/2, 4/5], [2.5/2, 6/5]] #
kmeans = vaex.ml.cluster.KMeans(features=features, init=init, verbose=True)
kmeans.fit(df)
```

```
Iteration    0, inertia  6.2609999999999975
Iteration    1, inertia  2.5062184444444435
Iteration    2, inertia  2.443455900151798
Iteration    3, inertia  2.418136327962199
Iteration    4, inertia  2.4161501474358995
Iteration    5, inertia  2.4161501474358995
```

We now transform the original DataFrame, similar to sklearn. However, we now end up with a new DataFrame, which contains an extra column (prediction_kmeans).

[11]:
```
df_predict = kmeans.transform(df)
df_predict
```

[11]:
```
#    sepal_width    petal_length    sepal_length    petal_width    class_    random_
↪index    prediction_kmeans
0    3.0            4.2             5.9             1.5            1         114        ␣
↪        1
1    3.0            4.6             6.1             1.4            1         74         ␣
↪        1
2    2.9            4.6             6.6             1.3            1         37         ␣
↪        1
3    3.3            5.7             6.7             2.1            2         116        ␣
↪        2
4    4.2            1.4             5.5             0.2            0         61         ␣
↪        0
...  ...            ...             ...             ...            ...       ...        ␣
↪        ...
145  3.4            1.4             5.2             0.2            0         119        ␣
↪        0
146  3.8            1.6             5.1             0.2            0         15         ␣
↪        0
147  2.6            4.0             5.8             1.2            1         22         ␣
↪        1
148  3.8            1.7             5.7             0.3            0         144        ␣
↪        0
149  2.9            4.3             6.2             1.3            1         102        ␣
↪        1
```

Although this column is special, it is actually a virtual column, it does not use up any memory and will be computed on the fly when needed, saving us precious ram. Note that the other columns reference the original data as well, so this new DataFrame (ds_predict) almost takes up no memory at all, which is ideal for very large datasets, and quite different from what sklearn will do.

[12]:
```
df_predict.virtual_columns['prediction_kmeans']
```

[12]:
```
'kmean_predict_function(petal_width/2, petal_length/5)'
```

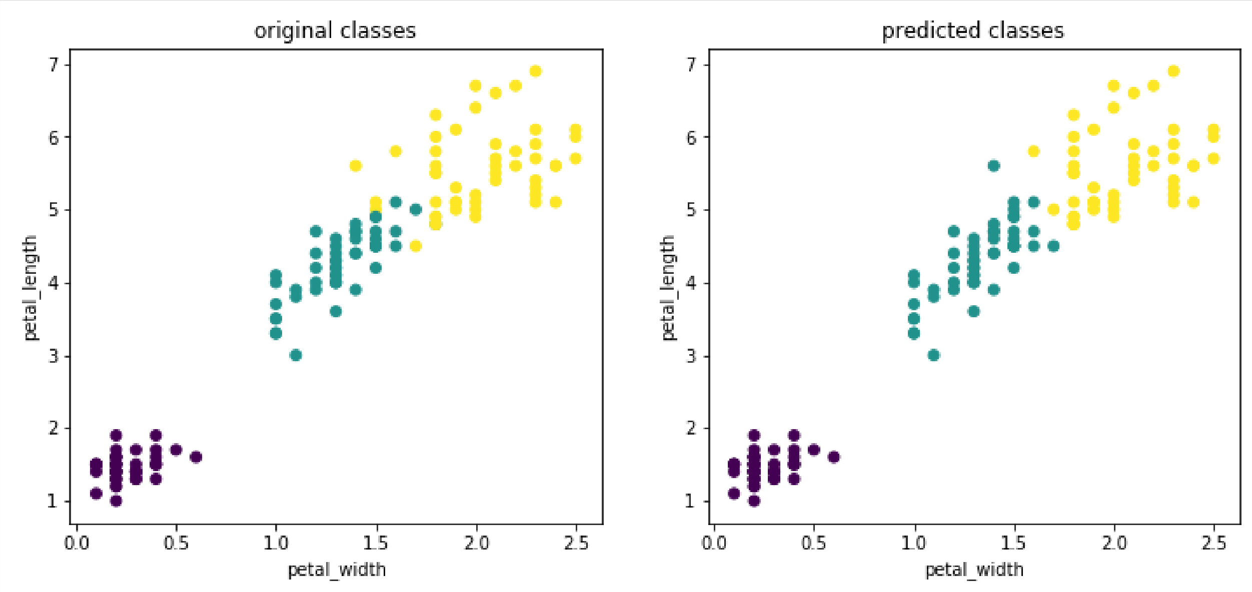
By making a simple scatter plot we can see the KMeans does a pretty good job.

[16]:
```
import matplotlib.pylab as plt
fig, ax = plt.subplots(1, 2, figsize=(12,5))

plt.sca(ax[0])
plt.title('original classes')
df.scatter(df.petal_width, df.petal_length, c_expr=df.class_)

plt.sca(ax[1])
plt.title('predicted classes')
df_predict.scatter(df_predict.petal_width, df_predict.petal_length, c_expr=df_predict.
↪prediction_kmeans)
```

[16]:
```
<matplotlib.collections.PathCollection at 0x12333eeb8>
```



7.2 KMeans benchmark

To demonstrate the performance and scaling of vaex, we continue with a special version of the iris dataset that has $\sim 10^7$ rows, by repeating the rows many times.

```
[28]: df = vaex.ml.datasets.load_iris_1e7()
```

We now use random initial conditions, and execute 10 runs in parallel (n_init), for a maximum of 5 iterations and benchmark it.

```
[29]: features = ['petal_width/2', 'petal_length/5']
kmeans = vaex.ml.cluster.KMeans(features=features, n_clusters=3, init='random',␣
↪random_state=1,
                                max_iter=5, verbose=True, n_init=10)
```

```
[30]: %%timeit -n1 -r1 -o
kmeans.fit(df)
```

```
Iteration    0, inertia  1784973.7999986452 |  1548329.799999016 |  354711.
↪39999875583 |  434173.39999885217 |  1005871.0000026902 |  1312114.6000003854 | ␣
↪1989377.3999927905 |  577104.4999989534 |  2747388.6000027955 |  628486.7999971791
Iteration    1, inertia  481645.0225601919 |  233311.807648651 |  214794.26525253727␣
↪|  175205.9965848818 |  490218.5413715277 |  816598.0811733825 |  285786.2566865457␣
↪|  456305.0601529535 |  1205488.9851008556 |  262443.28449456714
Iteration    2, inertia  458443.873920266 |  162015.13397359708 |  173081.69460305249␣
↪|  162580.06671935317 |  488402.97447322187 |  436698.8939923954 |  162626.
↪5498899455 |  394680.5108569788 |  850103.6561417003 |  198213.0961053151
Iteration    3, inertia  394680.5108569788 |  161882.05987810466 |  162580.0667193532␣
↪|  161882.05987810466 |  487435.98983613256 |  214098.28159484005 |  161882.
↪05987810466 |  275282.3731570135 |  594451.8937940609 |  169525.19719336918
Iteration    4, inertia  275282.3731570135 |  161882.05987810463 |  161882.
↪05987810463 |  161882.05987810463 |  486000.83124050766 |  169097.2713565477 | ␣
↪161882.05987810463 |  201144.2611065195 |  512055.1808623869 |  162023.37977993558
3.98 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[30]:
```
<TimeitResult : 3.98 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)>
```

[31]:
```
time_vaex = _
```

We now do the same using sklearn.

[32]:
```
from sklearn.cluster import KMeans
kmeans_sk = kmeans = KMeans(n_clusters=3, init='random', max_iter=5, verbose=True,␣
↪algorithm='full', n_jobs=-1,
                            precompute_distances=False, n_init=10)
# Doing an unfortunate memory copy
X = np.array(df[features])
```

[33]:
```
%%timeit -n1 -r1 -o
kmeans_sk.fit(X)
```

```
Initialization complete
Iteration  0, inertia 538264.600
Iteration  1, inertia 488488.457
Iteration  2, inertia 477825.973
Iteration  3, inertia 458443.874
Iteration  4, inertia 394680.511
Initialization complete
Iteration  0, inertia 1478542.600
Iteration  1, inertia 488488.457
Iteration  2, inertia 477825.973
Iteration  3, inertia 458443.874
Iteration  4, inertia 394680.511
Initialization complete
Iteration  0, inertia 422756.600
Iteration  1, inertia 182182.175
Iteration  2, inertia 164120.408
Iteration  3, inertia 162023.380
Iteration  4, inertia 161882.060
Converged at iteration 4: center shift 0.000000e+00 within tolerance 1.341649e-05
Initialization complete
Iteration  0, inertia 1873065.400
Iteration  1, inertia 260752.951
Iteration  2, inertia 161882.060
Converged at iteration 2: center shift 0.000000e+00 within tolerance 1.341649e-05
Initialization complete
Iteration  0, inertia 808489.000
Iteration  1, inertia 275282.373
Iteration  2, inertia 201144.261
Iteration  3, inertia 171177.750
Iteration  4, inertia 162580.067
Initialization complete
Iteration  0, inertia 3983719.500
Iteration  1, inertia 1112312.157
Iteration  2, inertia 550309.867
Iteration  3, inertia 261374.998
Iteration  4, inertia 178472.171
Initialization complete
Iteration  0, inertia 952003.000
Iteration  1, inertia 367032.453
Iteration  2, inertia 212341.557
Iteration  3, inertia 174578.392
```

(continues on next page)

(continued from previous page)

```
Iteration  4, inertia 165938.240
Initialization complete
Iteration  0, inertia 635682.600
Iteration  1, inertia 448595.010
Iteration  2, inertia 382619.025
Iteration  3, inertia 275282.373
Iteration  4, inertia 201144.261
Initialization complete
Iteration  0, inertia 950998.000
Iteration  1, inertia 490981.793
Iteration  2, inertia 490578.465
Iteration  3, inertia 489369.579
Iteration  4, inertia 488391.841
Initialization complete
Iteration  0, inertia 1329668.600
Iteration  1, inertia 353015.441
Iteration  2, inertia 168858.217
Iteration  3, inertia 162015.134
Iteration  4, inertia 161882.060
Converged at iteration 4: center shift 0.000000e+00 within tolerance 1.341649e-05
46.8 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[33]:
```
<TimeitResult : 46.8 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)>
```

[34]:
```
time_sklearn = _
```

We see that vaex is quite fast:

[35]:
```
print('vaex is approx', time_sklearn.best / time_vaex.best, 'times faster for KMeans')
```

```
vaex is approx 11.77461207454833 times faster for KMeans
```

But also, sklean will need to copy the data, while vaex will be very careful not to do unnecessary copies, and minimal amounts of passes of the data (Out-of-core). Therefore vaex will happily scale to massive datasets, while with sklearn you will be limited to the size of the RAM.

7.3 PCA Benchmark

We now continue with benchmarking a PCA on 4 features:

[36]:
```
features = [k.expression for k in [df.col.petal_width, df.col.petal_length, df.col.
↪sepal_width, df.col.sepal_length]]
pca = df.ml.pca(features=features)
```

[37]:
```
%%timeit -n1 -r3 -o
pca = df.ml.pca(features=features)
```

```
226 ms ± 30.6 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

[37]:
```
<TimeitResult : 226 ms ± 30.6 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)>
```

[38]:
```
time_vaex = _
```

Since sklearn takes too much memory with this dataset, we only use 10% for sklearn, and correct later.

[40]:
```
# on my laptop this takes too much memory with sklearn, use only a subset
factor = 0.1
df.set_active_fraction(factor)
len(df)
```

[40]:
```
1005000
```

[41]:
```
from sklearn.decomposition import PCA
pca_sk = PCA(n_components=2, random_state=33, svd_solver='full', whiten=False)
X = np.array(df.trim()[features])
```

[42]:
```
%%timeit -n1 -r3 -o
pca_sk.fit(X)
```

```
130 ms ± 25 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

[42]:
```
<TimeitResult : 130 ms ± 25 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)>
```

[43]:
```
time_sklearn = _
```

[44]:
```
print('vaex is approx', time_sklearn.best / time_vaex.best / factor, 'times faster␣
↪for a PCA')
```

```
vaex is approx 5.4043995278391295 times faster for a PCA
```

Again we see that vaex not only will outperform sklearn, but more importantly it will scale to much larger datasets.

7.4 A billion row PCA

We now run a PCA on **a billion rows**.

[51]:
```
df_big = vaex.ml.datasets.load_iris_1e9()
```

[52]:
```
%%timeit -n1 -r2 -o
pca = df_big.ml.pca(features=features)
```

```
3min 9s ± 20.5 s per loop (mean ± std. dev. of 2 runs, 1 loop each)
```

[52]:
```
<TimeitResult : 3min 9s ± 20.5 s per loop (mean ± std. dev. of 2 runs, 1 loop each)>
```

Note the although this dataset is $10\times$ larger, it takes more than $10\times$ to execute. This is because this dataset did not fit into memory this time, and is limited to the harddrive speed. But note that it *possible* to actually run it, instead of giving a MemoryError!

7.5 XGBoost

This example shows integration with xgboost, this is work in progress.

[2]:
```
import vaex.ml.xgboost
```

[3]:
```
df = vaex.ml.datasets.load_iris()
```

[4]:
```
features = [k.expression for k in [df.col.petal_width, df.col.petal_length, df.col.
↪sepal_width, df.col.sepal_length]]
```

[6]:
```
df_train, df_test = df.ml.train_test_split(verbose=False)
```

[7]:
```
params = {
    'max_depth': 3,  # the maximum depth of each tree
    'eta': 0.3,  # the training step for each iteration
    'silent': 1,  # logging mode - quiet
    'objective': 'multi:softmax',  # error evaluation for multiclass training
    'num_class': 3}  # the number of classes that exist in this datset
xgmodel = vaex.ml.xgboost.XGBoostModel(features=features, num_boost_round=10,␣
↪params=params)
```

[9]:
```
xgmodel.fit(df_train, df_train.class_)
```

[10]:
```
df_predict = xgmodel.transform(df_test)
df_predict
```

[10]:

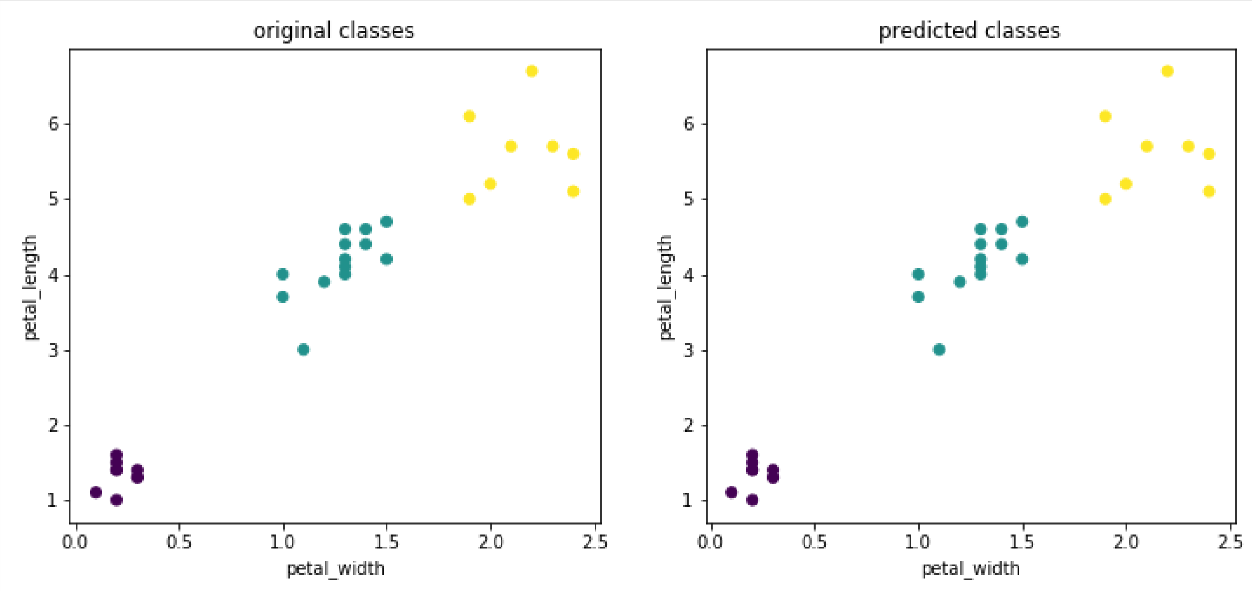
#	sepal_length	sepal_width	petal_length	petal_width	class_	xgboost_ ↪prediction
0	5.9	3.0	4.2	1.5	1	1.0
1	6.1	3.0	4.6	1.4	1	1.0
2	6.6	2.9	4.6	1.3	1	1.0
3	6.7	3.3	5.7	2.1	2	2.0
4	5.5	4.2	1.4	0.2	0	0.0
...	...	...	...	...	...	...
25	5.5	2.5	4.0	1.3	1	1.0
26	5.8	2.7	3.9	1.2	1	1.0
27	4.4	2.9	1.4	0.2	0	0.0
28	4.5	2.3	1.3	0.3	0	0.0
29	6.9	3.2	5.7	2.3	2	2.0

[11]:
```
import matplotlib.pylab as plt
fig, ax = plt.subplots(1, 2, figsize=(12,5))

plt.sca(ax[0])
plt.title('original classes')
df_predict.scatter(df_predict.petal_width, df_predict.petal_length, c_expr=df_predict.
↪class_)

plt.sca(ax[1])
plt.title('predicted classes')
df_predict.scatter(df_predict.petal_width, df_predict.petal_length, c_expr=df_predict.
↪xgboost_prediction)
```

[11]:
```
<matplotlib.collections.PathCollection at 0x7f81e829ea58>
```



7.6 One hot encoding

Shortly showing one hot encoding

[27]:
```
encoder = df.ml_one_hot_encoder([df.col.class_])
df_encoded = encoder.transform(df)
```

[28]:
```
df_encoded
```

[28]:
```
#     sepal_width     petal_length     sepal_length     petal_width     class_     random_
↪index     class__0      class__1     class__2
0     3.0             4.2              5.9              1.5             1          114      ␣
↪          0             1            0
1     3.0             4.6              6.1              1.4             1          74       ␣
↪          0             1            0
2     2.9             4.6              6.6              1.3             1          37       ␣
↪          0             1            0
3     3.3             5.7              6.7              2.1             2          116      ␣
↪          0             0            1
4     4.2             1.4              5.5              0.2             0          61       ␣
↪          1             0            0
...   ...             ...              ...              ...             ...        ...      ␣
↪          ...           ...          ...
145   3.4             1.4              5.2              0.2             0          119      ␣
↪          1             0            0
146   3.8             1.6              5.1              0.2             0          15       ␣
↪          1             0            0
147   2.6             4.0              5.8              1.2             1          22       ␣
↪          0             1            0
148   3.8             1.7              5.7              0.3             0          144      ␣
↪          1             0            0
149   2.9             4.3              6.2              1.3             1          102      ␣
↪          0             1            0
```

```
[ ]:
```

CHAPTER 8

Datasets to download

Here we list a few datasets, that might be interesting to explore with vaex

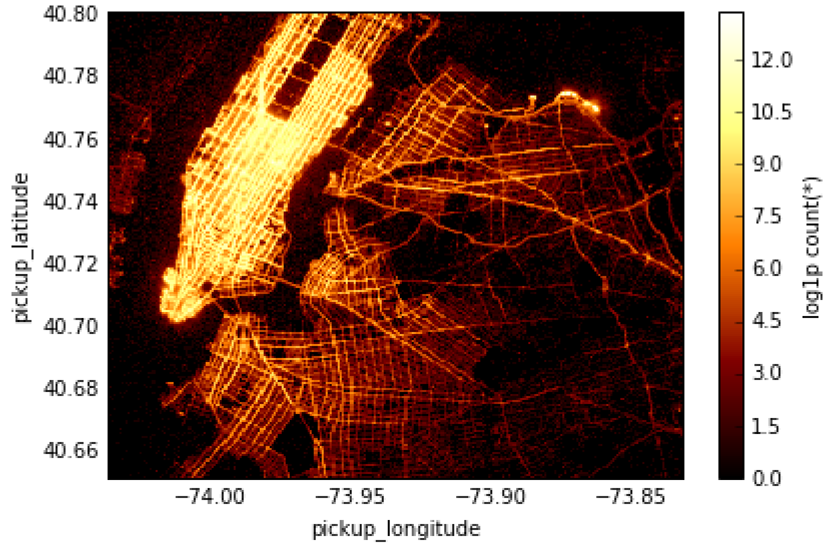
8.1 New york taxi dataset

See for instance Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance for some ideas.

- Year: 2015 - 146 million rows - 23GB
- Year 2009-2015 - 1 billion rows - 135GB

```
[2]: import vaex
```

```
[12]: df = vaex.open("/Users/users/breddels/.vaex/data/nyc_taxi/nyc_taxi2015.hdf5")
df.plot(df.col.pickup_longitude, df.col.pickup_latitude, f="log1p", show=True, limits=
↪"96%");
```



8.2 SDSS - dereddened

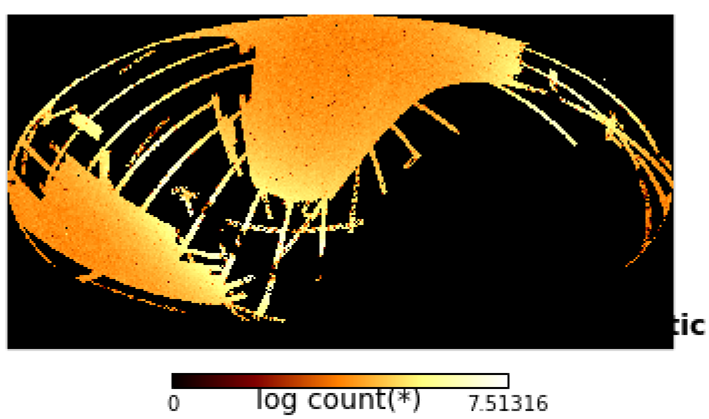
Only: `ra`, `dec`, `g`, `r`, `g_r` (deredenned using Schlegel maps).

The original query at SDSS archive was (although split in small parts):

```
SELECT ra, dec, g, r from PhotoObjAll WHERE type = 6 and  clean = 1 and r>=10.0 and r
↪<23.5;
```

- 162 million rows - 10GB

[22]:
```
sdss = vaex.open("/Users/maartenbreddels/vaex/data/sdss/sdss_dereddened.hdf5")
sdss.healpix_plot(sdss.col.healpix, show=True, f="log", healpix_max_level=9, healpix_
↪level=9,
                 healpix_input='galactic', healpix_output='galactic', rotation=(0,45)
                )
```

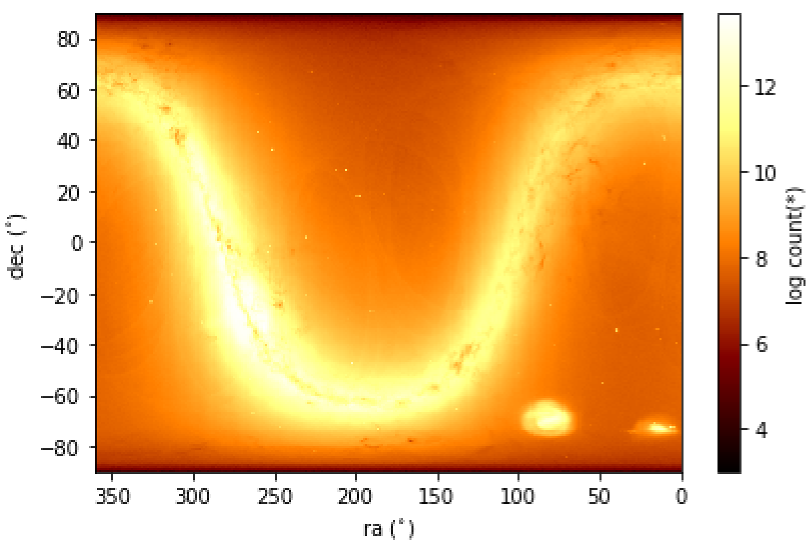


8.3 Gaia

See the Gaia Science Homepage for details, and you may want to try the Gaia Archive for ADQL (SQL like) queries.

- Gaia data release 2 (DR2)
  - Full Gaia DR2 - 1.7 billion rows 1.2TB
  - Split in two sets of columns:
  - All astrometry and errors (without covariances), radial velocity and basic photometry - 253 GB
  - Everything not contained in the above - 1 TB
  - Only with radial velocities - 7 million - 5.2GB
- Gaia data release 1 (DR1)
  - Full Gaia DR1 - 1 billion row - 351GB
  - A few columns of Gaia DR1 - 1 billion row - 88GB
  - 10% of Gaia DR1 - 1 billion row - 35GB
  - TGAS (subset of DR1 with proper motions) - 662MB

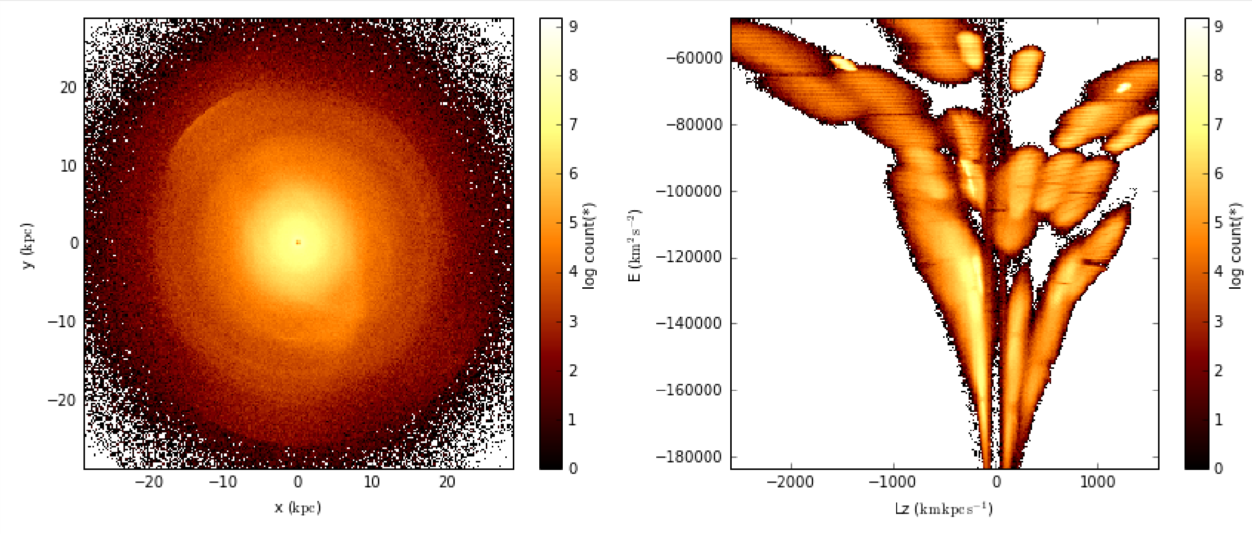
[3]:
```
gaia = vaex.open("/data/users/gaia/gaia-dr2/gaia-dr2-sort-by-source_id.hdf5")
gaia.plot("ra", "dec", f="log", limits=[[360, 0], [-90, 90]], show=True);
```



8.4 Helmi & de Zeeuw 2000

Result of an N-body simulation of the accretion of 33 satellite galaxies into a Milky Way dark matter halo * 3 million rows - 252MB

[26]:
```
hdz = vaex.datasets.helmi_de_zeeuw.fetch() # this will download it on the fly
hdz.plot([["x", "y"], ["Lz", "E"]], f="log", figsize=(12,5), show=True);
```



y (kpc)
x (kpc)
log count(*)
E (km²s⁻²)
Lz (kmkpcs⁻¹)
log count(*)

CHAPTER 9

What is Vaex?

Vaex is a python library for lazy **Out-of-Core DataFrames** (similar to Pandas), to visualize and explore big tabular datasets. It can calculate *statistics* such as mean, sum, count, standard deviation etc, on an *N-dimensional grid* up to **a billion** (10^9) objects/rows **per second**. Visualization is done using **histograms**, **density plots** and **3d volume rendering**, allowing interactive exploration of big data. Vaex uses memory mapping, a zero memory copy policy, and lazy computations for best performance (no memory wasted).

9.1 Why vaex

- **Performance:** works with huge tabular data, processes 10^9 rows/second
- **Lazy / Virtual columns:** compute on the fly, without wasting ram
- **Memory efficient** no memory copies when doing filtering/selections/subsets.
- **Visualization:** directly supported, a one-liner is often enough.
- **User friendly API:** you will only need to deal with the DataFrame object, and tab completion + docstring will help you out: `ds.mean<tab>`, feels very similar to Pandas.
- **Lean:** separated into multiple packages
  - `vaex-core`: DataFrame and core algorithms, takes numpy arrays as input columns.
  - `vaex-hdf5`: Provides memory mapped numpy arrays to a DataFrame.
  - `vaex-arrow`: Arrow support for cross language data sharing.
  - `vaex-viz`: Visualization based on matplotlib.
  - `vaex-jupyter`: Interactive visualization based on Jupyter widgets / ipywidgets, bqplot, ipyvolume and ipyleaflet.
  - `vaex-astro`: Astronomy related transformations and FITS file support.
  - `vaex-server`: Provides a server to access a DataFrame remotely.

- `vaex-distributed`: (Proof of concept) combined multiple servers / cluster into a single DataFrame for distributed computations.
- `vaex-qt`: Program written using Qt GUI.
- `vaex`: Meta package that installs all of the above.
- `vaex-ml`: *Machine learning*

- **Jupyter integration**: vaex-jupyter will give you interactive visualization and selection in the Jupyter notebook and Jupyter lab.

CHAPTER 10

Installation

Using conda:

- `conda install -c conda-forge vaex`

Using pip:

- `pip install --upgrade vaex`

Or read the *detailed instructions*

10.1 Getting started

We assume that you have installed vaex, and are running a Jupyter notebook server. We start by importing vaex and asking it to give us an example dataset.

[1]:
```
import vaex
df = vaex.example()  # open the example dataset provided with vaex
```

Instead, you can *download some larger datasets*, or *read in your csv file*.

[2]:
```
df  # will pretty print the DataFrame
```

[2]:
```
#        x              y              z              vx             vy             vz          ␣
↪    E                L                  Lz                  FeH
0        -0.777470767   2.10626292     1.93743467     53.276722      288.386047     -95.
↪2649078  -121238.171875   831.0799560546875   -336.426513671875    -2.
↪309227609164518
1        3.77427316     2.23387194     3.76209331     252.810791     -69.9498444    -56.
↪3121033  -100819.9140625  1435.1839599609375  -828.7567749023438   -1.
↪788735491591229
2        1.3757627      -6.3283844     2.63250017     96.276474      226.440201     -34.
↪7527161  -100559.9609375  1039.2989501953125  920.802490234375     -0.
↪7618109022478798
3        -7.06737804    1.31737781     -6.10543537    204.968842     -205.679016    -58.
↪9777031  -70174.8515625   2441.724853515625   1183.5899658203125   -1.
↪5208778422936413
```

(continues on next page)

(continued from previous page)

```
4        0.243441463   -0.822781682  -0.206593871  -311.742371  -238.41217   186.
↪824127   -144138.75       374.8164367675781   -314.5353088378906   -2.
↪655341358427361
...      ...           ...           ...           ...          ...          ...    ␣
↪    ...               ...                ...                ...
329,995  3.76883793    4.66251659    -4.42904139   107.432999   -2.13771296  17.
↪5130272   -119687.3203125  746.8833618164062   -508.96484375        -1.
↪6499842518381402
329,996  9.17409325    -8.87091351   -8.61707687   32.0         108.089264   179.
↪060638   -68933.8046875   2395.633056640625   1275.490234375       -1.
↪4336036247720836
329,997  -1.14041007   -8.4957695    2.25749826    8.46711349   -38.2765236  -127.
↪541473  -112580.359375   1182.436279296875  115.58557891845703   -1.
↪9306227597361942
329,998  -14.2985935   -5.51750422   -8.65472317   110.221558   -31.3925591  86.
↪2726822   -74862.90625     1324.5926513671875  1057.017333984375    -1.
↪225019818838568
329,999  10.5450506    -8.86106777   -4.65835428   -2.10541415  -27.6108856  3.
↪80799961   -95361.765625    351.0955505371094   -309.81439208984375  -2.
↪5689636894079477
```

Using **'square brackets[] <api.rst#vaex.dataframe.DataFrame.__getitem__>'__**, we can easily filter or get different views on the DataFrame.

[3]:
```
df_negative = df[df.x < 0]  # easily filter your DataFrame, without making a copy
df_negative[:5][['x', 'y']]  # take the first five rows, and only the 'x' and 'y'␣
↪column (no memory copy!)
```

[3]:

#	x	y
0	-0.777471	2.10626
1	-7.06738	1.31738
2	-5.17174	7.82915
3	-15.9539	5.77126
4	-12.3995	13.9182

When dealing with huge datasets, say a billion rows (10^9), computations with the data can waste memory, up to 8 GB for a new column. Instead, vaex uses lazy computation, storing only a representation of the computation, and computations are done on the fly when needed. You can just use many of the numpy functions, as if it was a normal array.

[4]:
```
import numpy as np
# creates an expression (nothing is computed)
some_expression = df.x + df.z
some_expression  # for convenience, we print out some values
```

[4]:
```
<vaex.expression.Expression(expressions='(x + z)')> instance at 0x118f71550 values=[1.
↪159963903, 7.53636647, 4.00826287, -13.17281341, 0.0368475919999999985 ... (total␣
↪330000 values) ... -0.66020346, 0.5570163800000003, 1.1170881900000003, -22.
↪95331667, 5.8866963199999995]
```

These expressions can be added to a DataFrame, creating what we call a *virtual column*. These virtual columns are similar to normal columns, except they do not waste memory.

[5]:
```
df['r'] = some_expression  # add a (virtual) column that will be computed on the fly
df.mean(df.x), df.mean(df.r)  # calculate statistics on normal and virtual columns
```

[5]:
```
(-0.06713149126400597, -0.0501732470530304)
```

One of the core features of vaex is its ability to calculate statistics on a regular (N-dimensional) grid. The dimensions of the grid are specified by the binby argument (analogous to SQL's grouby), and the shape and limits.

```
[6]: df.mean(df.r, binby=df.x, shape=32, limits=[-10, 10]) # create statistics on a␣
↪regular grid (1d)
```

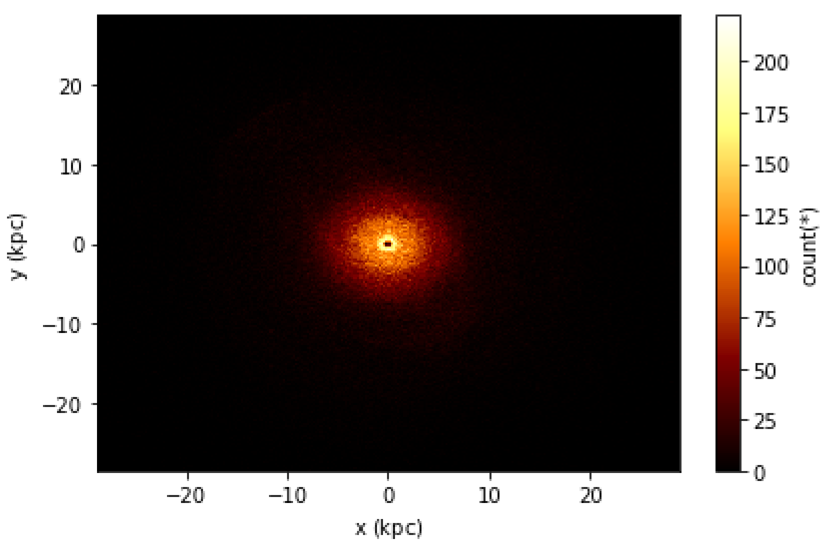
```
[6]: array([-9.67777315, -8.99466731, -8.17042477, -7.57122871, -6.98273954,
       -6.28362848, -5.70005784, -5.14022306, -4.52820368, -3.96953423,
       -3.3362477 , -2.7801045 , -2.20162243, -1.57910621, -0.92856689,
       -0.35964342,  0.30367721,  0.85684123,  1.53564551,  2.1274488 ,
        2.69235585,  3.37746363,  4.04648274,  4.59580105,  5.20540601,
        5.73475069,  6.28384101,  6.67880226,  7.46059303,  8.13480148,
        8.90738265,  9.6117928 ])
```

```
[7]: df.mean(df.r, binby=[df.x, df.y], shape=32, limits=[-10, 10]) # or 2d
df.count(df.r, binby=[df.x, df.y], shape=32, limits=[-10, 10]) # or 2d counts/
↪histogram
```

```
[7]: array([[22., 33., 37., ..., 58., 38., 45.],
       [37., 36., 47., ..., 52., 36., 53.],
       [34., 42., 47., ..., 59., 44., 56.],
       ...,
       [73., 73., 84., ..., 41., 40., 37.],
       [53., 58., 63., ..., 34., 35., 28.],
       [51., 32., 46., ..., 47., 33., 36.]])
```

These one and two dimensional grids can be visualized using any plotting library, such as matplotlib, but the setup can be tedious. For convenience we can use *plot1d*, *plot*, or see the *list of plotting commands*

```
[8]: df.plot(df.x, df.y, show=True);  # make a plot quickly
```



CHAPTER 11

Continue

Continue the tutorial here or check the *examples*

Python Module Index

V

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z