```
In [19]: pd.reset_option("^display")
```

`option_context` context manager has been exposed through the top-level API, allowing you to execute code with given option values. Option values are restored automatically when you exit the *with* block:

```
In [20]: with pd.option_context("display.max_rows", 10, "display.max_columns", 5):
   ....:     print(pd.get_option("display.max_rows"))
   ....:     print(pd.get_option("display.max_columns"))
   ....:
10
5

In [21]: print(pd.get_option("display.max_rows"))
60

In [22]: print(pd.get_option("display.max_columns"))
0
```

2.17.3 Setting startup options in Python/IPython environment

Using startup scripts for the Python/IPython environment to import pandas and set options makes working with pandas more efficient. To do this, create a .py or .ipy script in the startup directory of the desired profile. An example where the startup folder is in a default ipython profile can be found at:

```
$IPYTHONDIR/profile_default/startup
```

More information can be found in the ipython documentation. An example startup script for pandas is displayed below:

```
import pandas as pd
pd.set_option('display.max_rows', 999)
pd.set_option('precision', 5)
```

2.17.4 Frequently Used Options

The following is a walk-through of the more frequently used display options.

`display.max_rows` and `display.max_columns` sets the maximum number of rows and columns displayed when a frame is pretty-printed. Truncated lines are replaced by an ellipsis.

```
In [23]: df = pd.DataFrame(np.random.randn(7, 2))

In [24]: pd.set_option('max_rows', 7)

In [25]: df
Out[25]:
          0         1
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
3  0.119209 -1.044236
4 -0.861849 -2.104569
5 -0.494929  1.071804
6  0.721555 -0.706771
```

(continues on next page)

(continued from previous page)

```
In [26]: pd.set_option('max_rows', 5)

In [27]: df
Out[27]:
          0         1
0  0.469112 -0.282863
1 -1.509059 -1.135632
..      ...       ...
5 -0.494929  1.071804
6  0.721555 -0.706771

[7 rows x 2 columns]

In [28]: pd.reset_option('max_rows')
```

Once the `display.max_rows` is exceeded, the `display.min_rows` options determines how many rows are shown in the truncated repr.

```
In [29]: pd.set_option('max_rows', 8)

In [30]: pd.set_option('min_rows', 4)

# below max_rows -> all rows shown
In [31]: df = pd.DataFrame(np.random.randn(7, 2))

In [32]: df
Out[32]:
          0         1
0 -1.039575  0.271860
1 -0.424972  0.567020
2  0.276232 -1.087401
3 -0.673690  0.113648
4 -1.478427  0.524988
5  0.404705  0.577046
6 -1.715002 -1.039268

# above max_rows -> only min_rows (4) rows shown
In [33]: df = pd.DataFrame(np.random.randn(9, 2))

In [34]: df
Out[34]:
          0         1
0 -0.370647 -1.157892
1 -1.344312  0.844885
..      ...       ...
7  0.276662 -0.472035
8 -0.013960 -0.362543

[9 rows x 2 columns]

In [35]: pd.reset_option('max_rows')

In [36]: pd.reset_option('min_rows')
```

`display.expand_frame_repr` allows for the representation of dataframes to stretch across pages, wrapped over the full column vs row-wise.

```
In [37]: df = pd.DataFrame(np.random.randn(5, 10))

In [38]: pd.set_option('expand_frame_repr', True)

In [39]: df
Out[39]:
          0         1         2         3         4         5         6         7  ␣
↪        8         9
0 -0.006154 -0.923061  0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309 -1.
↪170299 -0.226169
1  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127 -1.436737 -1.
↪413681  1.607920
2  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747 -0.410001 -0.078638  0.
↪545952 -1.219217
3 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883  0.695775  0.341734  0.
↪959726 -1.110336
4 -0.619976  0.149748 -0.732339  0.687738  0.176444  0.403310 -0.154951  0.301624 -2.
↪179861 -1.369849

In [40]: pd.set_option('expand_frame_repr', False)

In [41]: df
Out[41]:
          0         1         2         3         4         5         6         7  ␣
↪        8         9
0 -0.006154 -0.923061  0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309 -1.
↪170299 -0.226169
1  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127 -1.436737 -1.
↪413681  1.607920
2  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747 -0.410001 -0.078638  0.
↪545952 -1.219217
3 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883  0.695775  0.341734  0.
↪959726 -1.110336
4 -0.619976  0.149748 -0.732339  0.687738  0.176444  0.403310 -0.154951  0.301624 -2.
↪179861 -1.369849

In [42]: pd.reset_option('expand_frame_repr')
```

`display.large_repr` lets you select whether to display dataframes that exceed `max_columns` or `max_rows` as a truncated frame, or as a summary.

```
In [43]: df = pd.DataFrame(np.random.randn(10, 10))

In [44]: pd.set_option('max_rows', 5)

In [45]: pd.set_option('large_repr', 'truncate')

In [46]: df
Out[46]:
           0         1         2         3         4         5         6         7  ␣
↪         8         9
0  -0.954208  1.462696 -1.743161 -0.826591 -0.345352  1.314232  0.690579  0.995761  2.
↪396780  0.014871
1   3.357427 -0.317441 -1.236269  0.896171 -0.487602 -0.082240 -2.182937  0.380396  0.
↪084844  0.432390
..       ...       ...       ...       ...       ...       ...       ...       ...     ␣
↪   ...       ...
```

(continues on next page)

(continued from previous page)

```
8  -0.303421 -0.858447  0.306996 -0.028665  0.384316  1.574159  1.588931  0.476720  0.
↪473424 -0.242861
9  -0.014805 -0.284319  0.650776 -1.461665 -1.137707 -0.891060 -0.693921  1.613616  0.
↪464000  0.227371

[10 rows x 10 columns]

In [47]: pd.set_option('large_repr', 'info')

In [48]: df
Out[48]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   0       10 non-null     float64
 1   1       10 non-null     float64
 2   2       10 non-null     float64
 3   3       10 non-null     float64
 4   4       10 non-null     float64
 5   5       10 non-null     float64
 6   6       10 non-null     float64
 7   7       10 non-null     float64
 8   8       10 non-null     float64
 9   9       10 non-null     float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [49]: pd.reset_option('large_repr')

In [50]: pd.reset_option('max_rows')
```

`display.max_colwidth` sets the maximum width of columns. Cells of this length or longer will be truncated with an ellipsis.

```
In [51]: df = pd.DataFrame(np.array([['foo', 'bar', 'bim', 'uncomfortably long string
↪'],
   ....:                                ['horse', 'cow', 'banana', 'apple']]))
   ....:

In [52]: pd.set_option('max_colwidth', 40)

In [53]: df
Out[53]:
       0    1       2                          3
0    foo  bar     bim  uncomfortably long string
1  horse  cow  banana                      apple

In [54]: pd.set_option('max_colwidth', 6)

In [55]: df
Out[55]:
       0    1      2      3
0    foo  bar    bim  un...
1  horse  cow  ba...  apple
```

(continues on next page)

(continued from previous page)

```
In [56]: pd.reset_option('max_colwidth')
```

`display.max_info_columns` sets a threshold for when by-column info will be given.

```
In [57]: df = pd.DataFrame(np.random.randn(10, 10))

In [58]: pd.set_option('max_info_columns', 11)

In [59]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       10 non-null     float64
 1   1       10 non-null     float64
 2   2       10 non-null     float64
 3   3       10 non-null     float64
 4   4       10 non-null     float64
 5   5       10 non-null     float64
 6   6       10 non-null     float64
 7   7       10 non-null     float64
 8   8       10 non-null     float64
 9   9       10 non-null     float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [60]: pd.set_option('max_info_columns', 5)

In [61]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Columns: 10 entries, 0 to 9
dtypes: float64(10)
memory usage: 928.0 bytes

In [62]: pd.reset_option('max_info_columns')
```

`display.max_info_rows`: `df.info()` will usually show null-counts for each column. For large frames this can be quite slow. `max_info_rows` and `max_info_cols` limit this null check only to frames with smaller dimensions then specified. Note that you can specify the option `df.info(null_counts=True)` to override on showing a particular frame.

```
In [63]: df = pd.DataFrame(np.random.choice([0, 1, np.nan], size=(10, 10)))

In [64]: df
Out[64]: 
     0    1    2    3    4    5    6    7    8    9
0  0.0  NaN  1.0  NaN  NaN  0.0  NaN  0.0  NaN  1.0
1  1.0  NaN  1.0  1.0  1.0  1.0  NaN  0.0  0.0  NaN
2  0.0  NaN  1.0  0.0  0.0  NaN  NaN  NaN  NaN  0.0
3  NaN  NaN  NaN  0.0  1.0  1.0  NaN  1.0  NaN  1.0
4  0.0  NaN  NaN  NaN  0.0  NaN  NaN  NaN  1.0  0.0
5  0.0  1.0  1.0  1.0  1.0  0.0  NaN  NaN  1.0  0.0
6  1.0  1.0  1.0  NaN  1.0  NaN  1.0  0.0  NaN  NaN
```

(continues on next page)

(continued from previous page)

```
7  0.0  0.0  1.0  0.0  1.0  0.0  1.0  1.0  0.0  NaN
8  NaN  NaN  NaN  0.0  NaN  NaN  NaN  NaN  1.0  NaN
9  0.0  NaN  0.0  NaN  NaN  0.0  NaN  1.0  1.0  0.0

In [65]: pd.set_option('max_info_rows', 11)

In [66]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   0       8 non-null      float64
 1   1       3 non-null      float64
 2   2       7 non-null      float64
 3   3       6 non-null      float64
 4   4       7 non-null      float64
 5   5       6 non-null      float64
 6   6       2 non-null      float64
 7   7       6 non-null      float64
 8   8       6 non-null      float64
 9   9       6 non-null      float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [67]: pd.set_option('max_info_rows', 5)

In [68]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
 #   Column  Dtype
---  ------  -----
 0   0       float64
 1   1       float64
 2   2       float64
 3   3       float64
 4   4       float64
 5   5       float64
 6   6       float64
 7   7       float64
 8   8       float64
 9   9       float64
dtypes: float64(10)
memory usage: 928.0 bytes

In [69]: pd.reset_option('max_info_rows')
```

`display.precision` sets the output display precision in terms of decimal places. This is only a suggestion.

```
In [70]: df = pd.DataFrame(np.random.randn(5, 5))

In [71]: pd.set_option('precision', 7)

In [72]: df
Out[72]:
           0          1          2          3          4
```

(continues on next page)

(continued from previous page)

```
0 -1.1506406 -0.7983341 -0.5576966  0.3813531  1.3371217
1 -1.5310949  1.3314582 -0.5713290 -0.0266708 -1.0856630
2 -1.1147378 -0.0582158 -0.4867681  1.6851483  0.1125723
3 -1.4953086  0.8984347 -0.1482168 -1.5960698  0.1596530
4  0.2621358  0.0362196  0.1847350 -0.2550694 -0.2710197

In [73]: pd.set_option('precision', 4)

In [74]: df
Out[74]:
        0       1       2       3       4
0 -1.1506 -0.7983 -0.5577  0.3814  1.3371
1 -1.5311  1.3315 -0.5713 -0.0267 -1.0857
2 -1.1147 -0.0582 -0.4868  1.6851  0.1126
3 -1.4953  0.8984 -0.1482 -1.5961  0.1597
4  0.2621  0.0362  0.1847 -0.2551 -0.2710
```

`display.chop_threshold` sets at what level pandas rounds to zero when it displays a Series of DataFrame. This setting does not change the precision at which the number is stored.

```
In [75]: df = pd.DataFrame(np.random.randn(6, 6))

In [76]: pd.set_option('chop_threshold', 0)

In [77]: df
Out[77]:
        0       1       2       3       4       5
0  1.2884  0.2946 -1.1658  0.8470 -0.6856  0.6091
1 -0.3040  0.6256 -0.0593  0.2497  1.1039 -1.0875
2  1.9980 -0.2445  0.1362  0.8863 -1.3507 -0.8863
3 -1.0133  1.9209 -0.3882 -2.3144  0.6655  0.4026
4  0.3996 -1.7660  0.8504  0.3881  0.9923  0.7441
5 -0.7398 -1.0549 -0.1796  0.6396  1.5850  1.9067

In [78]: pd.set_option('chop_threshold', .5)

In [79]: df
Out[79]:
        0       1       2       3       4       5
0  1.2884  0.0000 -1.1658  0.8470 -0.6856  0.6091
1  0.0000  0.6256  0.0000  0.0000  1.1039 -1.0875
2  1.9980  0.0000  0.0000  0.8863 -1.3507 -0.8863
3 -1.0133  1.9209  0.0000 -2.3144  0.6655  0.0000
4  0.0000 -1.7660  0.8504  0.0000  0.9923  0.7441
5 -0.7398 -1.0549  0.0000  0.6396  1.5850  1.9067

In [80]: pd.reset_option('chop_threshold')
```

`display.colheader_justify` controls the justification of the headers. The options are 'right', and 'left'.

```
In [81]: df = pd.DataFrame(np.array([np.random.randn(6),
   ....:                             np.random.randint(1, 9, 6) * .1,
   ....:                             np.zeros(6)]).T,
   ....:                   columns=['A', 'B', 'C'], dtype='float')
   ....:
```

(continues on next page)

(continued from previous page)

```
In [82]: pd.set_option('colheader_justify', 'right')

In [83]: df
Out[83]:
        A    B    C
0  0.1040  0.1  0.0
1  0.1741  0.5  0.0
2 -0.4395  0.4  0.0
3 -0.7413  0.8  0.0
4 -0.0797  0.4  0.0
5 -0.9229  0.3  0.0

In [84]: pd.set_option('colheader_justify', 'left')

In [85]: df
Out[85]:
   A        B    C
0  0.1040  0.1  0.0
1  0.1741  0.5  0.0
2 -0.4395  0.4  0.0
3 -0.7413  0.8  0.0
4 -0.0797  0.4  0.0
5 -0.9229  0.3  0.0

In [86]: pd.reset_option('colheader_justify')
```

2.17.5 Available options

Option	Default	Function
display.chop_threshold	None	If set to a float value, all float values smaller then the given threshold will be dis
display.colheader_justify	right	Controls the justification of column headers. used by DataFrameFormatter.
display.column_space	12	No description available.
display.date_dayfirst	False	When True, prints and parses dates with the day first, eg 20/01/2005
display.date_yearfirst	False	When True, prints and parses dates with the year first, eg 2005/01/20
display.encoding	UTF-8	Defaults to the detected encoding of the console. Specifies the encoding to be u
display.expand_frame_repr	True	Whether to print out the full DataFrame repr for wide DataFrames across multi
display.float_format	None	The callable should accept a floating point number and return a string with the d
display.large_repr	truncate	For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can
display.latex.repr	False	Whether to produce a latex DataFrame representation for jupyter frontends that
display.latex.escape	True	Escapes special characters in DataFrames, when using the to_latex method.
display.latex.longtable	False	Specifies if the to_latex method of a DataFrame uses the longtable format.
display.latex.multicolumn	True	Combines columns when using a MultiIndex
display.latex.multicolumn_format	'l'	Alignment of multicolumn labels
display.latex.multirow	False	Combines rows when using a MultiIndex. Centered instead of top-aligned, sepa
display.max_columns	0 or 20	max_rows and max_columns are used in __repr__() methods to decide if to_str
display.max_colwidth	50	The maximum width in characters of a column in the repr of a pandas data struc
display.max_info_columns	100	max_info_columns is used in DataFrame.info method to decide if per column in
display.max_info_rows	1690785	df.info() will usually show null-counts for each column. For large frames this c
display.max_rows	60	This sets the maximum number of rows pandas should output when printing ou
display.min_rows	10	The numbers of rows to show in a truncated repr (when *max_rows* is exceeded)
display.max_seq_items	100	when pretty-printing a long sequence, no more then *max_seq_items* will be prin

Option	Default	Function
display.memory_usage	True	This specifies if the memory usage of a DataFrame should be displayed when th
display.multi_sparse	True	"Sparsify" MultiIndex display (don't display repeated elements in outer levels w
display.notebook_repr_html	True	When True, IPython notebook will use html representation for pandas objects (
display.pprint_nest_depth	3	Controls the number of nested levels to process when pretty-printing
display.precision	6	Floating point output precision in terms of number of places after the decimal, f
display.show_dimensions	truncate	Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is sp
display.width	80	Width of the display in characters. In case python/IPython is running in a termi
display.html.table_schema	False	Whether to publish a Table Schema representation for frontends that support it.
display.html.border	1	A `border=value` attribute is inserted in the `<table>` tag for the DataFrame
display.html.use_mathjax	True	When True, Jupyter notebook will process table contents using MathJax, render
io.excel.xls.writer	xlwt	The default Excel writer engine for 'xls' files.
io.excel.xlsm.writer	openpyxl	The default Excel writer engine for 'xlsm' files. Available options: 'openpyxl' (
io.excel.xlsx.writer	openpyxl	The default Excel writer engine for 'xlsx' files.
io.hdf.default_format	None	default format writing format, if None, then put will default to 'fixed' and appen
io.hdf.dropna_table	True	drop ALL nan rows when appending to a table
io.parquet.engine	None	The engine to use as a default for parquet reading and writing. If None then try
mode.chained_assignment	warn	Controls `SettingWithCopyWarning`: 'raise', 'warn', or None. Raise an ex
mode.sim_interactive	False	Whether to simulate interactive mode for purposes of testing.
mode.use_inf_as_na	False	True means treat None, NaN, -INF, INF as NA (old way), False means None an
compute.use_bottleneck	True	Use the bottleneck library to accelerate computation if it is installed.
compute.use_numexpr	True	Use the numexpr library to accelerate computation if it is installed.
plotting.backend	matplotlib	Change the plotting backend to a different backend than the current matplotlib o
plotting.matplotlib.register_converters	True	Register custom converters with matplotlib. Set to False to de-register.

2.17.6 Number formatting

pandas also allows you to set how numbers are displayed in the console. This option is not set through the `set_options` API.

Use the `set_eng_float_format` function to alter the floating-point formatting of pandas objects to produce a particular format.

For instance:

```
In [87]: import numpy as np

In [88]: pd.set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [89]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [90]: s / 1.e3
Out[90]:
a    303.638u
b   -721.084u
c   -622.696u
d    648.250u
e     -1.945m
dtype: float64

In [91]: s / 1.e6
Out[91]:
a    303.638n
```

(continues on next page)

(continued from previous page)

```
b   -721.084n
c   -622.696n
d    648.250n
e     -1.945u
dtype: float64
```

To round floats on a case-by-case basis, you can also use `round()` and `round()`.

2.17.7 Unicode formatting

> **Warning:** Enabling this option will affect the performance for printing of DataFrame and Series (about 2 times slower). Use only when it is actually required.

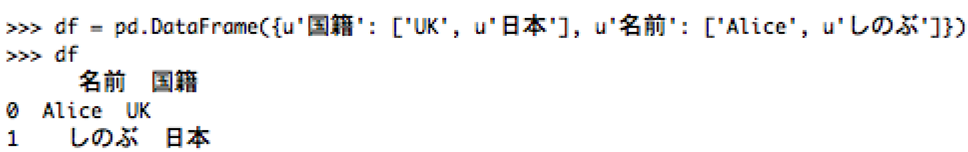
Some East Asian countries use Unicode characters whose width corresponds to two Latin characters. If a DataFrame or Series contains these characters, the default output mode may not align them properly.

Note: Screen captures are attached for each output to show the actual results.

```
In [92]: df = pd.DataFrame({'': ['UK', ''], '': ['Alice', '']})

In [93]: df
Out[93]:

0  UK  Alice
1
```

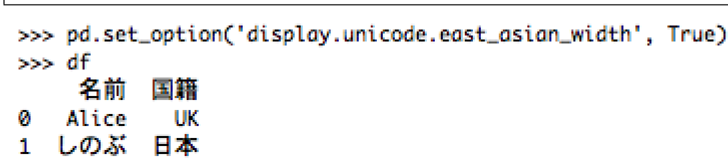


Enabling `display.unicode.east_asian_width` allows pandas to check each character's "East Asian Width" property. These characters can be aligned properly by setting this option to `True`. However, this will result in longer render times than the standard `len` function.

```
In [94]: pd.set_option('display.unicode.east_asian_width', True)

In [95]: df
Out[95]:

0    UK   Alice
1
```



In addition, Unicode characters whose width is "Ambiguous" can either be 1 or 2 characters wide depending on the terminal setting or encoding. The option `display.unicode.ambiguous_as_wide` can be used to handle the ambiguity.

By default, an "Ambiguous" character's width, such as "¡" (inverted exclamation) in the example below, is taken to be 1.

```
In [96]: df = pd.DataFrame({'a': ['xxx', '¡¡'], 'b': ['yyy', '¡¡']})

In [97]: df
Out[97]:
     a    b
0  xxx  yyy
1   ¡¡   ¡¡
```

```
>>> df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})
>>> df
     a    b
0  xxx  yyy
1   ¡¡   ¡¡
```

Enabling `display.unicode.ambiguous_as_wide` makes pandas interpret these characters' widths to be 2. (Note that this option will only be effective when `display.unicode.east_asian_width` is enabled.)

However, setting this option incorrectly for your terminal will cause these characters to be aligned incorrectly:

```
In [98]: pd.set_option('display.unicode.ambiguous_as_wide', True)

In [99]: df
Out[99]:
      a     b
0   xxx   yyy
1  ¡¡  ¡¡
```

```
>>> pd.set_option('display.unicode.ambiguous_as_wide', True)
>>> df
      a     b
0   xxx   yyy
1  ¡¡  ¡¡
```

2.17.8 Table schema display

`DataFrame` and `Series` will publish a Table Schema representation by default. False by default, this can be enabled globally with the `display.html.table_schema` option:

```
In [100]: pd.set_option('display.html.table_schema', True)
```

Only `'display.max_rows'` are serialized and published.

2.18 Enhancing performance

In this part of the tutorial, we will investigate how to speed up certain functions operating on pandas `DataFrames` using three different techniques: Cython, Numba and *pandas.eval()*. We will see a speed improvement of ~200 when we use Cython and Numba on a test function operating row-wise on the `DataFrame`. Using *pandas.eval()* we will speed up a sum by an order of ~2.

2.18.1 Cython (writing C extensions for pandas)

For many use cases writing pandas in pure Python and NumPy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in Python, for example by trying to remove for-loops and making use of NumPy vectorization. It's always worth optimising in Python first.

This tutorial walks through a "typical" process of cythonizing a slow computation. We use an example from the Cython documentation but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure Python solution.

Pure Python

We have a `DataFrame` to which we want to apply a function row-wise.

```
In [1]: df = pd.DataFrame({'a': np.random.randn(1000),
   ...:                    'b': np.random.randn(1000),
   ...:                    'N': np.random.randint(100, 1000, (1000)),
   ...:                    'x': 'x'})
   ...:

In [2]: df
Out[2]:
            a         b    N  x
0    0.469112 -0.218470  585  x
1   -0.282863 -0.061645  841  x
2   -1.509059 -0.723780  251  x
3   -1.135632  0.551225  972  x
4    1.212112 -0.497767  181  x
..        ...       ...  ... ..
995 -1.512743  0.874737  374  x
996  0.933753  1.120790  246  x
997 -0.308013  0.198768  157  x
998 -0.079915  1.757555  977  x
999 -1.010589 -1.115680  770  x

[1000 rows x 4 columns]
```

Here's the function in pure Python:

```
In [3]: def f(x):
   ...:     return x * (x - 1)
   ...:

In [4]: def integrate_f(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f(a + i * dx)
   ...:     return s * dx
   ...:
```

We achieve our result by using `apply` (row-wise):

```
In [7]: %timeit df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 174 ms per loop
```

But clearly this isn't fast enough for us. Let's take a look and see where the time is spent during this operation (limited to the most time consuming four calls) using the prun ipython magic function:

```
In [5]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)  #␣
↪noqa E999
         701222 function calls (698196 primitive calls) in 0.496 seconds

   Ordered by: internal time
   List reduced from 227 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.226    0.000    0.337    0.000 <ipython-input-4-c2a74e076cf0>:
↪1(integrate_f)
   552423    0.111    0.000    0.111    0.000 <ipython-input-3-c138bdd570e3>:1(f)
     3000    0.015    0.000    0.125    0.000 base.py:4373(get_value)
    18363    0.012    0.000    0.022    0.000 {built-in method builtins.isinstance}
```

By far the majority of time is spend inside either `integrate_f` or `f`, hence we'll concentrate our efforts cythonizing these two functions.

Note: In Python 2 replacing the `range` with its generator counterpart (`xrange`) would mean the `range` line would vanish. In Python 3 `range` is already a generator.

Plain Cython

First we're going to need to import the Cython magic function to ipython:

```
In [6]: %load_ext Cython
```

Now, let's simply copy our functions over to Cython as is (the suffix is here to distinguish between function versions):

```
In [7]: %%cython
   ...: def f_plain(x):
   ...:     return x * (x - 1)
   ...: def integrate_f_plain(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_plain(a + i * dx)
   ...:     return s * dx
   ...:
```

Note: If you're having trouble pasting the above into your ipython, you may need to be using bleeding edge ipython for paste to play well with cell magics.

```
In [4]: %timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 85.5 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

Adding type

We get another huge improvement simply by providing type information:

```
In [8]: %%cython
   ...: cdef double f_typed(double x) except? -2:
   ...:     return x * (x - 1)
   ...: cpdef double integrate_f_typed(double a, double b, int N):
   ...:     cdef int i
   ...:     cdef double s, dx
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_typed(a + i * dx)
   ...:     return s * dx
   ...:
```

```
In [4]: %timeit df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 20.3 ms per loop
```

Now, we're talking! It's now over ten times faster than the original python implementation, and we haven't *really* modified the code. Let's have another look at what's eating up time:

```
In [9]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']),␣
↪axis=1)
         148795 function calls (145769 primitive calls) in 0.122 seconds

   Ordered by: internal time
   List reduced from 222 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3000    0.012    0.000    0.098    0.000 base.py:4373(get_value)
    18363    0.009    0.000    0.018    0.000 {built-in method builtins.isinstance}
     3000    0.007    0.000    0.108    0.000 series.py:868(__getitem__)
     3006    0.007    0.000    0.043    0.000 construction.py:337(extract_array)
```

Using ndarray

It's calling series… a lot! It's creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in Python, so maybe we could minimize these by cythonizing the apply part.

Note: We are now passing ndarrays into the Cython function, fortunately Cython plays very nicely with NumPy.

```
In [10]: %%cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
```

(continues on next page)

(continued from previous page)

```
   ....:     for i in range(N):
   ....:         s += f_typed(a + i * dx)
   ....:     return s * dx
   ....: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_
↪b,
   ....:                                            np.ndarray col_N):
   ....:     assert (col_a.dtype == np.float
   ....:             and col_b.dtype == np.float and col_N.dtype == np.int)
   ....:     cdef Py_ssize_t i, n = len(col_N)
   ....:     assert (len(col_a) == len(col_b) == n)
   ....:     cdef np.ndarray[double] res = np.empty(n)
   ....:     for i in range(len(col_a)):
   ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
   ....:     return res
   ....:
```

The implementation is simple, it creates an array of zeros and loops over the rows, applying our `integrate_f_typed`, and putting this in the zeros array.

Warning: You can **not pass** a `Series` directly as a `ndarray` typed parameter to a Cython function. Instead pass the actual `ndarray` using the *Series.to_numpy()*. The reason is that the Cython definition is specific to an ndarray and not the passed `Series`.

So, do not do this:

```
apply_integrate_f(df['a'], df['b'], df['N'])
```

But rather, use *Series.to_numpy()* to get the underlying `ndarray`:

```
apply_integrate_f(df['a'].to_numpy(),
                  df['b'].to_numpy(),
                  df['N'].to_numpy())
```

Note: Loops like this would be *extremely* slow in Python, but in Cython looping over NumPy arrays is *fast*.

```
In [4]: %timeit apply_integrate_f(df['a'].to_numpy(),
                                  df['b'].to_numpy(),
                                  df['N'].to_numpy())
1000 loops, best of 3: 1.25 ms per loop
```

We've gotten another big improvement. Let's check again where the time is spent:

```
In [11]: %%prun -l 4 apply_integrate_f(df['a'].to_numpy(),
   ....:                               df['b'].to_numpy(),
   ....:                               df['N'].to_numpy())
   ....:
         260 function calls in 0.002 seconds

   Ordered by: internal time
   List reduced from 65 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002    0.002    0.002 {built-in method _cython_magic_
↪bcb0b6f1e59fec3c1d5a6272f215f062.apply_integrate_f}
```

(continues on next page)

(continued from previous page)

```
        3    0.000    0.000    0.001    0.000 frame.py:2767(__getitem__)
        1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
        3    0.000    0.000    0.000    0.000 managers.py:979(iget)
```

As one might expect, the majority of the time is now spent in `apply_integrate_f`, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

More advanced techniques

There is still hope for improvement. Here's an example of using some more advanced Cython techniques:

```
In [12]: %%cython
   ....: cimport cython
   ....: cimport numpy as np
   ....: import numpy as np
   ....: cdef double f_typed(double x) except? -2:
   ....:     return x * (x - 1)
   ....: cpdef double integrate_f_typed(double a, double b, int N):
   ....:     cdef int i
   ....:     cdef double s, dx
   ....:     s = 0
   ....:     dx = (b - a) / N
   ....:     for i in range(N):
   ....:         s += f_typed(a + i * dx)
   ....:     return s * dx
   ....: @cython.boundscheck(False)
   ....: @cython.wraparound(False)
   ....: cpdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a,
   ....:                                                 np.ndarray[double] col_b,
   ....:                                                 np.ndarray[int] col_N):
   ....:     cdef int i, n = len(col_N)
   ....:     assert len(col_a) == len(col_b) == n
   ....:     cdef np.ndarray[double] res = np.empty(n)
   ....:     for i in range(n):
   ....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
   ....:     return res
   ....:
```

```
In [4]: %timeit apply_integrate_f_wrap(df['a'].to_numpy(),
                                       df['b'].to_numpy(),
                                       df['N'].to_numpy())
1000 loops, best of 3: 987 us per loop
```

Even faster, with the caveat that a bug in our Cython code (an off-by-one error, for example) might cause a segfault because memory access isn't checked. For more about `boundscheck` and `wraparound`, see the Cython docs on compiler directives.

2.18.2 Using Numba

A recent alternative to statically compiling Cython code, is to use a *dynamic jit-compiler*, Numba.

Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack.

Note: You will need to install Numba. This is easy with `conda`, by using: `conda install numba`, see *installing using miniconda*.

Note: As of Numba version 0.20, pandas objects cannot be passed directly to Numba-compiled functions. Instead, one must pass the NumPy array underlying the pandas object to the Numba-compiled function as demonstrated below.

Jit

We demonstrate how to use Numba to just-in-time compile our code. We simply take the plain Python code from above and annotate with the `@jit` decorator.

```
import numba


@numba.jit
def f_plain(x):
    return x * (x - 1)


@numba.jit
def integrate_f_numba(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_plain(a + i * dx)
    return s * dx


@numba.jit
def apply_integrate_f_numba(col_a, col_b, col_N):
    n = len(col_N)
    result = np.empty(n, dtype='float64')
    assert len(col_a) == len(col_b) == n
    for i in range(n):
        result[i] = integrate_f_numba(col_a[i], col_b[i], col_N[i])
    return result


def compute_numba(df):
    result = apply_integrate_f_numba(df['a'].to_numpy(),
                                     df['b'].to_numpy(),
```

(continues on next page)

(continued from previous page)

```
                                    df['N'].to_numpy())
    return pd.Series(result, index=df.index, name='result')
```

Note that we directly pass NumPy arrays to the Numba function. `compute_numba` is just a wrapper that provides a nicer interface by passing/returning pandas objects.

```
In [4]: %timeit compute_numba(df)
1000 loops, best of 3: 798 us per loop
```

In this example, using Numba was faster than Cython.

Vectorize

Numba can also be used to write vectorized functions that do not require the user to explicitly loop over the observations of a vector; a vectorized function will be applied to each row automatically. Consider the following toy example of doubling each observation:

```
import numba


def double_every_value_nonumba(x):
    return x * 2


@numba.vectorize
def double_every_value_withnumba(x):  # noqa E501
    return x * 2
```

```
# Custom function without numba
In [5]: %timeit df['col1_doubled'] = df['a'].apply(double_every_value_nonumba)  #␣
↪noqa E501
1000 loops, best of 3: 797 us per loop

# Standard implementation (faster than a custom function)
In [6]: %timeit df['col1_doubled'] = df['a'] * 2
1000 loops, best of 3: 233 us per loop

# Custom function with numba
In [7]: %timeit (df['col1_doubled'] = double_every_value_withnumba(df['a'].to_numpy())
1000 loops, best of 3: 145 us per loop
```

Caveats

Note: Numba will execute on any function, but can only accelerate certain classes of functions.

Numba is best at accelerating functions that apply numerical functions to NumPy arrays. When passed a function that only uses operations it knows how to accelerate, it will execute in `nopython` mode.

If Numba is passed a function that includes something it doesn't know how to work with – a category that currently includes sets, lists, dictionaries, or string functions – it will revert to `object mode`. In `object mode`, Numba will execute but your code will not speed up significantly. If you would prefer that Numba throw an error if it cannot compile a function in a way that speeds up your code, pass Numba the argument `nopython=True` (e.g. `@numba.jit(nopython=True)`). For more on troubleshooting Numba modes, see the Numba troubleshooting page.

Read more in the Numba docs.

2.18.3 Expression evaluation via `eval()`

The top-level function `pandas.eval()` implements expression evaluation of `Series` and `DataFrame` objects.

Note: To benefit from using `eval()` you need to install `numexpr`. See the *recommended dependencies section* for more details.

The point of using `eval()` for expression evaluation rather than plain Python is two-fold: 1) large `DataFrame` objects are evaluated more efficiently and 2) large arithmetic and boolean expressions are evaluated all at once by the underlying engine (by default `numexpr` is used for evaluation).

Note: You should not use `eval()` for simple expressions or for expressions involving small DataFrames. In fact, `eval()` is many orders of magnitude slower for smaller expressions/objects than plain ol' Python. A good rule of thumb is to only use `eval()` when you have a `DataFrame` with more than 10,000 rows.

`eval()` supports all arithmetic expressions supported by the engine in addition to some extensions available only in pandas.

Note: The larger the frame and the larger the expression the more speedup you will see from using `eval()`.

Supported syntax

These operations are supported by `pandas.eval()`:

- Arithmetic operations except for the left shift (<<) and right shift (>>) operators, e.g., `df + 2 * pi / s ** 4 % 42 - the_golden_ratio`
- Comparison operations, including chained comparisons, e.g., `2 < df < df2`
- Boolean operations, e.g., `df < df2 and df3 < df4 or not df_bool`
- `list` and `tuple` literals, e.g., `[1, 2]` or `(1, 2)`
- Attribute access, e.g., `df.a`
- Subscript expressions, e.g., `df[0]`
- Simple variable evaluation, e.g., `pd.eval('df')` (this is not very useful)
- Math functions: *sin*, *cos*, *exp*, *log*, *expm1*, *log1p*, *sqrt*, *sinh*, *cosh*, *tanh*, *arcsin*, *arccos*, *arctan*, *arccosh*, *arcsinh*, *arctanh*, *abs*, *arctan2* and *log10*.

This Python syntax is **not** allowed:

- Expressions
  - Function calls other than math functions.
  - `is`/`is not` operations
  - `if` expressions

- `lambda` expressions
- `list/set/dict` comprehensions
- Literal `dict` and `set` expressions
- `yield` expressions
- Generator expressions
- Boolean expressions consisting of only scalar values

- Statements
  - Neither simple nor compound statements are allowed. This includes things like `for`, `while`, and `if`.

`eval()` examples

`pandas.eval()` works well with expressions containing large arrays.

First let's create a few decent-sized arrays to play with:

```
In [13]: nrows, ncols = 20000, 100

In [14]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols)) for _ in␣
↪range(4)]
```

Now let's compare adding them together using plain ol' Python versus `eval()`:

```
In [15]: %timeit df1 + df2 + df3 + df4
39.2 ms +- 4.8 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [16]: %timeit pd.eval('df1 + df2 + df3 + df4')
17.1 ms +- 1.25 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Now let's do the same thing but with comparisons:

```
In [17]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
202 ms +- 22.4 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [18]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
26.8 ms +- 3.43 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

`eval()` also works with unaligned pandas objects:

```
In [19]: s = pd.Series(np.random.randn(50))

In [20]: %timeit df1 + df2 + df3 + df4 + s
170 ms +- 15.9 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [21]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
23.9 ms +- 3.89 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Note: Operations such as

```
1 and 2  # would parse to 1 & 2, but should evaluate to 2
3 or 4  # would parse to 3 | 4, but should evaluate to 3
~1  # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type `bool` or `np.bool_`. Again, you should perform these kinds of operations in plain Python.

The `DataFrame.eval` method

In addition to the top level *pandas.eval()* function you can also evaluate an expression in the "context" of a *DataFrame*.

```
In [22]: df = pd.DataFrame(np.random.randn(5, 2), columns=['a', 'b'])

In [23]: df.eval('a + b')
Out[23]:
0   -0.246747
1    0.867786
2   -1.626063
3   -1.134978
4   -1.027798
dtype: float64
```

Any expression that is a valid *pandas.eval()* expression is also a valid *DataFrame.eval()* expression, with the added benefit that you don't have to prefix the name of the *DataFrame* to the column(s) you're interested in evaluating.

In addition, you can perform assignment of columns within an expression. This allows for *formulaic evaluation*. The assignment target can be a new column name or an existing column name, and it must be a valid Python identifier.

The `inplace` keyword determines whether this assignment will performed on the original `DataFrame` or return a copy with the new column.

Warning: For backwards compatibility, `inplace` defaults to `True` if not specified. This will change in a future version of pandas - if your code depends on an inplace assignment you should update to explicitly set `inplace=True`.

```
In [24]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [25]: df.eval('c = a + b', inplace=True)

In [26]: df.eval('d = a + b + c', inplace=True)

In [27]: df.eval('a = 1', inplace=True)

In [28]: df
Out[28]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

When `inplace` is set to `False`, a copy of the `DataFrame` with the new or modified columns is returned and the original frame is unchanged.

```
In [29]: df
Out[29]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26

In [30]: df.eval('e = a - c', inplace=False)
Out[30]:
   a  b   c   d   e
0  1  5   5  10  -4
1  1  6   7  14  -6
2  1  7   9  18  -8
3  1  8  11  22 -10
4  1  9  13  26 -12

In [31]: df
Out[31]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

As a convenience, multiple assignments can be performed by using a multi-line string.

```
In [32]: df.eval("""
   ....: c = a + b
   ....: d = a + b + c
   ....: a = 1""", inplace=False)
   ....:
Out[32]:
   a  b   c   d
0  1  5   6  12
1  1  6   7  14
2  1  7   8  16
3  1  8   9  18
4  1  9  10  20
```

The equivalent in standard Python would be

```
In [33]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [34]: df['c'] = df['a'] + df['b']

In [35]: df['d'] = df['a'] + df['b'] + df['c']

In [36]: df['a'] = 1

In [37]: df
Out[37]:
   a  b   c   d
0  1  5   5  10
1  1  6   7  14
```

(continues on next page)

(continued from previous page)

```
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

The `query` method has a `inplace` keyword which determines whether the query modifies the original frame.

```
In [38]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [39]: df.query('a > 2')
Out[39]:
   a  b
3  3  8
4  4  9

In [40]: df.query('a > 2', inplace=True)

In [41]: df
Out[41]:
   a  b
3  3  8
4  4  9
```

Warning: Unlike with `eval`, the default value for `inplace` for `query` is `False`. This is consistent with prior versions of pandas.

Local variables

You must *explicitly reference* any local variable that you want to use in an expression by placing the `@` character in front of the name. For example,

```
In [42]: df = pd.DataFrame(np.random.randn(5, 2), columns=list('ab'))

In [43]: newcol = np.random.randn(len(df))

In [44]: df.eval('b + @newcol')
Out[44]:
0   -0.173926
1    2.493083
2   -0.881831
3   -0.691045
4    1.334703
dtype: float64

In [45]: df.query('b < @newcol')
Out[45]:
          a         b
0  0.863987 -0.115998
2 -2.621419 -1.297879
```

If you don't prefix the local variable with `@`, pandas will raise an exception telling you the variable is undefined.

When using `DataFrame.eval()` and `DataFrame.query()`, this allows you to have a local variable and a `DataFrame` column with the same name in an expression.

```
In [46]: a = np.random.randn()

In [47]: df.query('@a < a')
Out[47]:
          a         b
0  0.863987 -0.115998

In [48]: df.loc[a < df['a']]  # same as the previous expression
Out[48]:
          a         b
0  0.863987 -0.115998
```

With *pandas.eval()* you cannot use the `@` prefix *at all*, because it isn't defined in that context. `pandas` will let you know this if you try to use `@` in a top-level call to *pandas.eval()*. For example,

```
In [49]: a, b = 1, 2

In [50]: pd.eval('@a + b')
Traceback (most recent call last):

  File "/opt/conda/envs/pandas/lib/python3.7/site-packages/IPython/core/
↪interactiveshell.py", line 3331, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-50-af17947a194f>", line 1, in <module>
    pd.eval('@a + b')

  File "/pandas-release/pandas/pandas/core/computation/eval.py", line 321, in eval
    _check_for_locals(expr, level, parser)

  File "/pandas-release/pandas/pandas/core/computation/eval.py", line 167, in _check_
↪for_locals
    raise SyntaxError(msg)

  File "<string>", line unknown
SyntaxError: The '@' prefix is not allowed in top-level eval calls,
please refer to your variables by name without the '@' prefix
```

In this case, you should simply refer to the variables like you would in standard Python.

```
In [51]: pd.eval('a + b')
Out[51]: 3
```

`pandas.eval()` parsers

There are two different parsers and two different engines you can use as the backend.

The default `'pandas'` parser allows a more intuitive syntax for expressing query-like operations (comparisons, conjunctions and disjunctions). In particular, the precedence of the `&` and `|` operators is made equal to the precedence of the corresponding boolean operations `and` and `or`.

For example, the above conjunction can be written without parentheses. Alternatively, you can use the `'python'` parser to enforce strict Python semantics.

```
In [52]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'
```

(continues on next page)

(continued from previous page)

```
In [53]: x = pd.eval(expr, parser='python')

In [54]: expr_no_parens = 'df1 > 0 & df2 > 0 & df3 > 0 & df4 > 0'

In [55]: y = pd.eval(expr_no_parens, parser='pandas')

In [56]: np.all(x == y)
Out[56]: True
```

The same expression can be "anded" together with the word `and` as well:

```
In [57]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)'

In [58]: x = pd.eval(expr, parser='python')

In [59]: expr_with_ands = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'

In [60]: y = pd.eval(expr_with_ands, parser='pandas')

In [61]: np.all(x == y)
Out[61]: True
```

The `and` and `or` operators here have the same precedence that they would in vanilla Python.

`pandas.eval()` backends

There's also the option to make `eval()` operate identical to plain ol' Python.

Note: Using the `'python'` engine is generally *not* useful, except for testing other evaluation engines against it. You will achieve **no** performance benefits using `eval()` with `engine='python'` and in fact may incur a performance hit.

You can see this by using `pandas.eval()` with the `'python'` engine. It is a bit slower (not by much) than evaluating the same expression in Python

```
In [62]: %timeit df1 + df2 + df3 + df4
44.7 ms +- 6.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [63]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
40.3 ms +- 4.66 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

`pandas.eval()` performance

`eval()` is intended to speed up certain kinds of operations. In particular, those operations involving complex expressions with large `DataFrame`/`Series` objects should see a significant performance benefit. Here is a plot showing the running time of `pandas.eval()` as function of the size of the frame involved in the computation. The two lines are two different engines.