Elements can be set to NaT using np.nan analogously to datetimes:

```
In [40]: y[1] = np.nan

In [41]: y
Out[41]:
0       NaT
1       NaT
2    1 days
dtype: timedelta64[ns]
```

Operands can also appear in a reversed order (a singular object operated with a Series):

```
In [42]: s.max() - s
Out[42]:
0    2 days
1    1 days
2    0 days
dtype: timedelta64[ns]

In [43]: datetime.datetime(2011, 1, 1, 3, 5) - s
Out[43]:
0   -365 days +03:05:00
1   -366 days +03:05:00
2   -367 days +03:05:00
dtype: timedelta64[ns]

In [44]: datetime.timedelta(minutes=5) + s
Out[44]:
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

min, max and the corresponding idxmin, idxmax operations are supported on frames:

```
In [45]: A = s - pd.Timestamp('20120101') - pd.Timedelta('00:05:05')

In [46]: B = s - pd.Series(pd.date_range('2012-1-2', periods=3, freq='D'))

In [47]: df = pd.DataFrame({'A': A, 'B': B})

In [48]: df
Out[48]:
                  A       B
0 -1 days +23:54:55 -1 days
1   0 days 23:54:55 -1 days
2   1 days 23:54:55 -1 days

In [49]: df.min()
Out[49]:
A   -1 days +23:54:55
B   -1 days +00:00:00
dtype: timedelta64[ns]

In [50]: df.min(axis=1)
Out[50]:
0   -1 days
```

(continues on next page)

(continued from previous page)

```
1   -1 days
2   -1 days
dtype: timedelta64[ns]

In [51]: df.idxmin()
Out[51]:
A    0
B    0
dtype: int64

In [52]: df.idxmax()
Out[52]:
A    2
B    0
dtype: int64
```

min, max, idxmin, idxmax operations are supported on Series as well. A scalar result will be a Timedelta.

```
In [53]: df.min().max()
Out[53]: Timedelta('-1 days +23:54:55')

In [54]: df.min(axis=1).min()
Out[54]: Timedelta('-1 days +00:00:00')

In [55]: df.min().idxmax()
Out[55]: 'A'

In [56]: df.min(axis=1).idxmin()
Out[56]: 0
```

You can fillna on timedeltas, passing a timedelta to get a particular value.

```
In [57]: y.fillna(pd.Timedelta(0))
Out[57]:
0   0 days
1   0 days
2   1 days
dtype: timedelta64[ns]

In [58]: y.fillna(pd.Timedelta(10, unit='s'))
Out[58]:
0   0 days 00:00:10
1   0 days 00:00:10
2   1 days 00:00:00
dtype: timedelta64[ns]

In [59]: y.fillna(pd.Timedelta('-1 days, 00:00:05'))
Out[59]:
0   -1 days +00:00:05
1   -1 days +00:00:05
2     1 days 00:00:00
dtype: timedelta64[ns]
```

You can also negate, multiply and use abs on Timedeltas:

```
In [60]: td1 = pd.Timedelta('-1 days 2 hours 3 seconds')
```

(continues on next page)

(continued from previous page)

```
In [61]: td1
Out[61]: Timedelta('-2 days +21:59:57')

In [62]: -1 * td1
Out[62]: Timedelta('1 days 02:00:03')

In [63]: - td1
Out[63]: Timedelta('1 days 02:00:03')

In [64]: abs(td1)
Out[64]: Timedelta('1 days 02:00:03')
```

2.15.3 Reductions

Numeric reduction operation for `timedelta64[ns]` will return `Timedelta` objects. As usual `NaT` are skipped during evaluation.

```
In [65]: y2 = pd.Series(pd.to_timedelta(['-1 days +00:00:05', 'nat',
   ....:                                 '-1 days +00:00:05', '1 days']))
   ....:

In [66]: y2
Out[66]:
0   -1 days +00:00:05
1                 NaT
2   -1 days +00:00:05
3     1 days 00:00:00
dtype: timedelta64[ns]

In [67]: y2.mean()
Out[67]: Timedelta('-1 days +16:00:03.333333')

In [68]: y2.median()
Out[68]: Timedelta('-1 days +00:00:05')

In [69]: y2.quantile(.1)
Out[69]: Timedelta('-1 days +00:00:05')

In [70]: y2.sum()
Out[70]: Timedelta('-1 days +00:00:10')
```

2.15.4 Frequency conversion

Timedelta Series, `TimedeltaIndex`, and `Timedelta` scalars can be converted to other 'frequencies' by dividing by another timedelta, or by astyping to a specific timedelta type. These operations yield Series and propagate `NaT` -> `nan`. Note that division by the NumPy scalar is true division, while astyping is equivalent of floor division.

```
In [71]: december = pd.Series(pd.date_range('20121201', periods=4))

In [72]: january = pd.Series(pd.date_range('20130101', periods=4))

In [73]: td = january - december
```

(continues on next page)

(continued from previous page)

```
In [74]: td[2] += datetime.timedelta(minutes=5, seconds=3)

In [75]: td[3] = np.nan

In [76]: td
Out[76]:
0   31 days 00:00:00
1   31 days 00:00:00
2   31 days 00:05:03
3                NaT
dtype: timedelta64[ns]

# to days
In [77]: td / np.timedelta64(1, 'D')
Out[77]:
0    31.000000
1    31.000000
2    31.003507
3          NaN
dtype: float64

In [78]: td.astype('timedelta64[D]')
Out[78]:
0    31.0
1    31.0
2    31.0
3     NaN
dtype: float64

# to seconds
In [79]: td / np.timedelta64(1, 's')
Out[79]:
0    2678400.0
1    2678400.0
2    2678703.0
3          NaN
dtype: float64

In [80]: td.astype('timedelta64[s]')
Out[80]:
0    2678400.0
1    2678400.0
2    2678703.0
3          NaN
dtype: float64

# to months (these are constant months)
In [81]: td / np.timedelta64(1, 'M')
Out[81]:
0    1.018501
1    1.018501
2    1.018617
3         NaN
dtype: float64
```

Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series yields another `timedelta64[ns]` dtypes Series.

```
In [82]: td * -1
Out[82]:
0   -31 days +00:00:00
1   -31 days +00:00:00
2   -32 days +23:54:57
3                  NaT
dtype: timedelta64[ns]

In [83]: td * pd.Series([1, 2, 3, 4])
Out[83]:
0    31 days 00:00:00
1    62 days 00:00:00
2    93 days 00:15:09
3                 NaT
dtype: timedelta64[ns]
```

Rounded division (floor-division) of a `timedelta64[ns]` Series by a scalar `Timedelta` gives a series of integers.

```
In [84]: td // pd.Timedelta(days=3, hours=4)
Out[84]:
0     9.0
1     9.0
2     9.0
3     NaN
dtype: float64

In [85]: pd.Timedelta(days=3, hours=4) // td
Out[85]:
0     0.0
1     0.0
2     0.0
3     NaN
dtype: float64
```

The mod (%) and divmod operations are defined for `Timedelta` when operating with another timedelta-like or with a numeric argument.

```
In [86]: pd.Timedelta(hours=37) % datetime.timedelta(hours=2)
Out[86]: Timedelta('0 days 01:00:00')

# divmod against a timedelta-like returns a pair (int, Timedelta)
In [87]: divmod(datetime.timedelta(hours=2), pd.Timedelta(minutes=11))
Out[87]: (10, Timedelta('0 days 00:10:00'))

# divmod against a numeric returns a pair (Timedelta, Timedelta)
In [88]: divmod(pd.Timedelta(hours=25), 86400000000000)
Out[88]: (Timedelta('0 days 00:00:00.000000'), Timedelta('0 days 01:00:00'))
```

2.15.5 Attributes

You can access various components of the Timedelta or TimedeltaIndex directly using the attributes days,seconds,microseconds,nanoseconds. These are identical to the values returned by datetime.timedelta, in that, for example, the .seconds attribute represents the number of seconds >= 0 and < 1 day. These are signed according to whether the Timedelta is signed.

These operations can also be directly accessed via the .dt property of the Series as well.

Note: Note that the attributes are NOT the displayed values of the Timedelta. Use .components to retrieve the displayed values.

For a Series:

```
In [89]: td.dt.days
Out[89]:
0    31.0
1    31.0
2    31.0
3     NaN
dtype: float64

In [90]: td.dt.seconds
Out[90]:
0      0.0
1      0.0
2    303.0
3      NaN
dtype: float64
```

You can access the value of the fields for a scalar Timedelta directly.

```
In [91]: tds = pd.Timedelta('31 days 5 min 3 sec')

In [92]: tds.days
Out[92]: 31

In [93]: tds.seconds
Out[93]: 303

In [94]: (-tds).seconds
Out[94]: 86097
```

You can use the .components property to access a reduced form of the timedelta. This returns a DataFrame indexed similarly to the Series. These are the *displayed* values of the Timedelta.

```
In [95]: td.dt.components
Out[95]:
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0  31.0    0.0      0.0      0.0           0.0           0.0          0.0
1  31.0    0.0      0.0      0.0           0.0           0.0          0.0
2  31.0    0.0      5.0      3.0           0.0           0.0          0.0
3   NaN    NaN      NaN      NaN           NaN           NaN          NaN

In [96]: td.dt.components.seconds
Out[96]:
```

(continues on next page)

(continued from previous page)

```
0    0.0
1    0.0
2    3.0
3    NaN
Name: seconds, dtype: float64
```

You can convert a `Timedelta` to an ISO 8601 Duration string with the `.isoformat` method

```
In [97]: pd.Timedelta(days=6, minutes=50, seconds=3,
   ....:              milliseconds=10, microseconds=10,
   ....:              nanoseconds=12).isoformat()
   ....:
Out[97]: 'P6DT0H50M3.010010012S'
```

2.15.6 TimedeltaIndex

To generate an index with time delta, you can use either the *`TimedeltaIndex`* or the *`timedelta_range()`* constructor.

Using `TimedeltaIndex` you can pass string-like, `Timedelta`, `timedelta`, or `np.timedelta64` objects. Passing `np.nan/pd.NaT/nat` will represent missing values.

```
In [98]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05', np.timedelta64(2, 'D'),
   ....:                    datetime.timedelta(days=2, seconds=2)])
   ....:
Out[98]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
                '2 days 00:00:02'],
               dtype='timedelta64[ns]', freq=None)
```

The string 'infer' can be passed in order to set the frequency of the index as the inferred frequency upon creation:

```
In [99]: pd.TimedeltaIndex(['0 days', '10 days', '20 days'], freq='infer')
Out[99]: TimedeltaIndex(['0 days', '10 days', '20 days'], dtype='timedelta64[ns]',␣
↪freq='10D')
```

Generating ranges of time deltas

Similar to *`date_range()`*, you can construct regular ranges of a `TimedeltaIndex` using *`timedelta_range()`*. The default frequency for `timedelta_range` is calendar day:

```
In [100]: pd.timedelta_range(start='1 days', periods=5)
Out[100]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↪'timedelta64[ns]', freq='D')
```

Various combinations of `start`, `end`, and `periods` can be used with `timedelta_range`:

```
In [101]: pd.timedelta_range(start='1 days', end='5 days')
Out[101]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↪'timedelta64[ns]', freq='D')

In [102]: pd.timedelta_range(end='10 days', periods=4)
Out[102]: TimedeltaIndex(['7 days', '8 days', '9 days', '10 days'], dtype=
↪'timedelta64[ns]', freq='D')
```

The freq parameter can passed a variety of *frequency aliases*:

```
In [103]: pd.timedelta_range(start='1 days', end='2 days', freq='30T')
Out[103]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
                '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                '1 days 10:30:00', '1 days 11:00:00', '1 days 11:30:00',
                '1 days 12:00:00', '1 days 12:30:00', '1 days 13:00:00',
                '1 days 13:30:00', '1 days 14:00:00', '1 days 14:30:00',
                '1 days 15:00:00', '1 days 15:30:00', '1 days 16:00:00',
                '1 days 16:30:00', '1 days 17:00:00', '1 days 17:30:00',
                '1 days 18:00:00', '1 days 18:30:00', '1 days 19:00:00',
                '1 days 19:30:00', '1 days 20:00:00', '1 days 20:30:00',
                '1 days 21:00:00', '1 days 21:30:00', '1 days 22:00:00',
                '1 days 22:30:00', '1 days 23:00:00', '1 days 23:30:00',
                '2 days 00:00:00'],
               dtype='timedelta64[ns]', freq='30T')

In [104]: pd.timedelta_range(start='1 days', periods=5, freq='2D5H')
Out[104]:
TimedeltaIndex(['1 days 00:00:00', '3 days 05:00:00', '5 days 10:00:00',
                '7 days 15:00:00', '9 days 20:00:00'],
               dtype='timedelta64[ns]', freq='53H')
```

New in version 0.23.0.

Specifying start, end, and periods will generate a range of evenly spaced timedeltas from start to end inclusively, with periods number of elements in the resulting TimedeltaIndex:

```
In [105]: pd.timedelta_range('0 days', '4 days', periods=5)
Out[105]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
↪'timedelta64[ns]', freq=None)

In [106]: pd.timedelta_range('0 days', '4 days', periods=10)
Out[106]:
TimedeltaIndex(['0 days 00:00:00', '0 days 10:40:00', '0 days 21:20:00',
                '1 days 08:00:00', '1 days 18:40:00', '2 days 05:20:00',
                '2 days 16:00:00', '3 days 02:40:00', '3 days 13:20:00',
                '4 days 00:00:00'],
               dtype='timedelta64[ns]', freq=None)
```

Using the TimedeltaIndex

Similarly to other of the datetime-like indices, DatetimeIndex and PeriodIndex, you can use TimedeltaIndex as the index of pandas objects.

```
In [107]: s = pd.Series(np.arange(100),
   .....:               index=pd.timedelta_range('1 days', periods=100, freq='h'))
   .....:

In [108]: s
Out[108]:
```

(continues on next page)

(continued from previous page)

```
1 days 00:00:00     0
1 days 01:00:00     1
1 days 02:00:00     2
1 days 03:00:00     3
1 days 04:00:00     4
                   ..
4 days 23:00:00    95
5 days 00:00:00    96
5 days 01:00:00    97
5 days 02:00:00    98
5 days 03:00:00    99
Freq: H, Length: 100, dtype: int64
```

Selections work similarly, with coercion on string-likes and slices:

```
In [109]: s['1 day':'2 day']
Out[109]:
1 days 00:00:00     0
1 days 01:00:00     1
1 days 02:00:00     2
1 days 03:00:00     3
1 days 04:00:00     4
                   ..
2 days 19:00:00    43
2 days 20:00:00    44
2 days 21:00:00    45
2 days 22:00:00    46
2 days 23:00:00    47
Freq: H, Length: 48, dtype: int64

In [110]: s['1 day 01:00:00']
Out[110]: 1

In [111]: s[pd.Timedelta('1 day 1h')]
Out[111]: 1
```

Furthermore you can use partial string selection and the range will be inferred:

```
In [112]: s['1 day':'1 day 5 hours']
Out[112]:
1 days 00:00:00    0
1 days 01:00:00    1
1 days 02:00:00    2
1 days 03:00:00    3
1 days 04:00:00    4
1 days 05:00:00    5
Freq: H, dtype: int64
```

Operations

Finally, the combination of `TimedeltaIndex` with `DatetimeIndex` allow certain combination operations that are NaT preserving:

```
In [113]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [114]: tdi.to_list()
Out[114]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [115]: dti = pd.date_range('20130101', periods=3)

In [116]: dti.to_list()
Out[116]:
[Timestamp('2013-01-01 00:00:00', freq='D'),
 Timestamp('2013-01-02 00:00:00', freq='D'),
 Timestamp('2013-01-03 00:00:00', freq='D')]

In [117]: (dti + tdi).to_list()
Out[117]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [118]: (dti - tdi).to_list()
Out[118]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]
```

Conversions

Similarly to frequency conversion on a `Series` above, you can convert these indices to yield another Index.

```
In [119]: tdi / np.timedelta64(1, 's')
Out[119]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

In [120]: tdi.astype('timedelta64[s]')
Out[120]: Float64Index([86400.0, nan, 172800.0], dtype='float64')
```

Scalars type ops work as well. These can potentially return a *different* type of index.

```
# adding or timedelta and date -> datelike
In [121]: tdi + pd.Timestamp('20130101')
Out[121]: DatetimeIndex(['2013-01-02', 'NaT', '2013-01-03'], dtype='datetime64[ns]',␣
↪freq=None)

# subtraction of a date and a timedelta -> datelike
# note that trying to subtract a date from a Timedelta will raise an exception
In [122]: (pd.Timestamp('20130101') - tdi).to_list()
Out[122]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2012-12-30 00:00:00')]

# timedelta + timedelta -> timedelta
In [123]: tdi + pd.Timedelta('10 days')
Out[123]: TimedeltaIndex(['11 days', NaT, '12 days'], dtype='timedelta64[ns]',␣
↪freq=None)

# division can result in a Timedelta if the divisor is an integer
In [124]: tdi / 2
Out[124]: TimedeltaIndex(['0 days 12:00:00', NaT, '1 days 00:00:00'], dtype=
↪'timedelta64[ns]', freq=None)
```

(continues on next page)

(continued from previous page)

```
# or a Float64Index if the divisor is a Timedelta
In [125]: tdi / tdi[0]
Out[125]: Float64Index([1.0, nan, 2.0], dtype='float64')
```

2.15.7 Resampling

Similar to *timeseries resampling*, we can resample with a `TimedeltaIndex`.

```
In [126]: s.resample('D').mean()
Out[126]:
1 days    11.5
2 days    35.5
3 days    59.5
4 days    83.5
5 days    97.5
Freq: D, dtype: float64
```

2.16 Styling

This document is written as a Jupyter Notebook, and can be viewed or downloaded here.

You can apply **conditional formatting**, the visual styling of a DataFrame depending on the data within, by using the `DataFrame.style` property. This is a property that returns a `Styler` object, which has useful methods for formatting and displaying DataFrames.

The styling is accomplished using CSS. You write "style functions" that take scalars, `DataFrames` or `Series`, and return *like-indexed* DataFrames or Series with CSS `"attribute: value"` pairs for the values. These functions can be incrementally passed to the `Styler` which collects the styles before rendering.

2.16.1 Building styles

Pass your style functions into one of the following methods:

- `Styler.applymap`: elementwise
- `Styler.apply`: column-/row-/table-wise

Both of those methods take a function (and some other keyword arguments) and applies your function to the DataFrame in a certain way. `Styler.applymap` works through the DataFrame elementwise. `Styler.apply` passes each column or row into your DataFrame one-at-a-time or the entire table at once, depending on the `axis` keyword argument. For columnwise use `axis=0`, rowwise use `axis=1`, and for the entire table at once use `axis=None`.

For `Styler.applymap` your function should take a scalar and return a single string with the CSS attribute-value pair.

For `Styler.apply` your function should take a Series or DataFrame (depending on the axis parameter), and return a Series or DataFrame with an identical shape where each value is a string with a CSS attribute-value pair.

Let's see some examples.

[2]:
```
import pandas as pd
import numpy as np
```

(continues on next page)

(continued from previous page)

```
np.random.seed(24)
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
               axis=1)
df.iloc[3, 3] = np.nan
df.iloc[0, 2] = np.nan
```

Here's a boring example of rendering a DataFrame, without any (visible) styles:

```
[3]: df.style
```

```
[3]: <pandas.io.formats.style.Styler at 0x7f10074d6f10>
```

Note: The `DataFrame.style` attribute is a property that returns a `Styler` object. `Styler` has a `_repr_html_` method defined on it so they are rendered automatically. If you want the actual HTML back for further processing or for writing to file call the `.render()` method which returns a string.

The above output looks very similar to the standard DataFrame HTML representation. But we've done some work behind the scenes to attach CSS classes to each cell. We can view these by calling the `.render` method.

```
[4]: df.style.highlight_null().render().split('\n')[:10]
```

```
[4]: ['<style  type="text/css" >',
 '    #T_37521038_b0c3_11ea_808b_0242ac110002row0_col2 {',
 '            background-color:  red;',
 '        }    #T_37521038_b0c3_11ea_808b_0242ac110002row3_col3 {',
 '            background-color:  red;',
 '        }</style><table id="T_37521038_b0c3_11ea_808b_0242ac110002" ><thead>    <tr>
↪        <th class="blank level0" ></th>        <th class="col_heading level0 col0" >
↪A</th>        <th class="col_heading level0 col1" >B</th>        <th class="col_
↪heading level0 col2" >C</th>        <th class="col_heading level0 col3" >D</th>    ␣
↪    <th class="col_heading level0 col4" >E</th>    </tr></thead><tbody>',
 '                <tr>',
 '                        <th id="T_37521038_b0c3_11ea_808b_0242ac110002level0_row0"␣
↪class="row_heading level0 row0" >0</th>',
 '                        <td id="T_37521038_b0c3_11ea_808b_0242ac110002row0_col0"␣
↪class="data row0 col0" >1.000000</td>',
 '                        <td id="T_37521038_b0c3_11ea_808b_0242ac110002row0_col1"␣
↪class="data row0 col1" >1.329212</td>']
```

The `row0_col2` is the identifier for that particular cell. We've also prepended each row/column identifier with a UUID unique to each DataFrame so that the style from one doesn't collide with the styling from another within the same notebook or page (you can set the `uuid` if you'd like to tie together the styling of two DataFrames).

When writing style functions, you take care of producing the CSS attribute / value pairs you want. Pandas matches those up with the CSS classes that identify each cell.

Let's write a simple style function that will color negative numbers red and positive numbers black.

```
[5]: def color_negative_red(val):
    """
    Takes a scalar and returns a string with
    the css property `'color: red'` for negative
    strings, black otherwise.
    """
    color = 'red' if val < 0 else 'black'
    return 'color: %s' % color
```

In this case, the cell's style depends only on it's own value. That means we should use the `Styler.applymap` method which works elementwise.

[6]:
```
s = df.style.applymap(color_negative_red)
s
```

[6]:
```
<pandas.io.formats.style.Styler at 0x7f0fe5ac68d0>
```

Notice the similarity with the standard `df.applymap`, which operates on DataFrames elementwise. We want you to be able to reuse your existing knowledge of how to interact with DataFrames.

Notice also that our function returned a string containing the CSS attribute and value, separated by a colon just like in a `<style>` tag. This will be a common theme.

Finally, the input shapes matched. `Styler.applymap` calls the function on each scalar input, and the function returns a scalar output.

Now suppose you wanted to highlight the maximum value in each column. We can't use `.applymap` anymore since that operated elementwise. Instead, we'll turn to `.apply` which operates columnwise (or rowwise using the `axis` keyword). Later on we'll see that something like `highlight_max` is already defined on `Styler` so you wouldn't need to write this yourself.

[7]:
```
def highlight_max(s):
    '''
    highlight the maximum in a Series yellow.
    '''
    is_max = s == s.max()
    return ['background-color: yellow' if v else '' for v in is_max]
```

[8]:
```
df.style.apply(highlight_max)
```

[8]:
```
<pandas.io.formats.style.Styler at 0x7f0fe5ac0710>
```

In this case the input is a `Series`, one column at a time. Notice that the output shape of `highlight_max` matches the input shape, an array with `len(s)` items.

We encourage you to use method chains to build up a style piecewise, before finally rending at the end of the chain.

[9]:
```
df.style.\
    applymap(color_negative_red).\
    apply(highlight_max)
```

[9]:
```
<pandas.io.formats.style.Styler at 0x7f0fe5b25b10>
```

Above we used `Styler.apply` to pass in each column one at a time.

Debugging Tip: If you're having trouble writing your style function, try just passing it into DataFrame.apply. Internally, Styler.apply uses DataFrame.apply so the result should be the same.

What if you wanted to highlight just the maximum value in the entire table? Use `.apply(function, axis=None)` to indicate that your function wants the entire table, not one column or row at a time. Let's try that next.

We'll rewrite our `highlight-max` to handle either Series (from `.apply(axis=0 or 1)`) or DataFrames (from `.apply(axis=None)`). We'll also allow the color to be adjustable, to demonstrate that `.apply`, and `.applymap` pass along keyword arguments.

[10]:
```
def highlight_max(data, color='yellow'):
    '''
    highlight the maximum in a Series or DataFrame
```

(continues on next page)

(continued from previous page)

```
    '''
    attr = 'background-color: {}'.format(color)
    if data.ndim == 1:  # Series from .apply(axis=0) or axis=1
        is_max = data == data.max()
        return [attr if v else '' for v in is_max]
    else:  # from .apply(axis=None)
        is_max = data == data.max().max()
        return pd.DataFrame(np.where(is_max, attr, ''),
                            index=data.index, columns=data.columns)
```

When using `Styler.apply(func, axis=None)`, the function must return a DataFrame with the same index and column labels.

```
[11]: df.style.apply(highlight_max, color='darkorange', axis=None)
```

```
[11]: <pandas.io.formats.style.Styler at 0x7f0fe5acc850>
```

Building Styles Summary

Style functions should return strings with one or more CSS `attribute: value` delimited by semicolons. Use

- `Styler.applymap(func)` for elementwise styles
- `Styler.apply(func, axis=0)` for columnwise styles
- `Styler.apply(func, axis=1)` for rowwise styles
- `Styler.apply(func, axis=None)` for tablewise styles

And crucially the input and output shapes of `func` must match. If `x` is the input then `func(x).shape == x.shape`.

2.16.2 Finer control: slicing

Both `Styler.apply`, and `Styler.applymap` accept a `subset` keyword. This allows you to apply styles to specific rows or columns, without having to code that logic into your `style` function.

The value passed to `subset` behaves similar to slicing a DataFrame.

- A scalar is treated as a column label
- A list (or series or numpy array)
- A tuple is treated as `(row_indexer, column_indexer)`

Consider using `pd.IndexSlice` to construct the tuple for the last one.

```
[12]: df.style.apply(highlight_max, subset=['B', 'C', 'D'])
```

```
[12]: <pandas.io.formats.style.Styler at 0x7f0fe5aaca90>
```

For row and column slicing, any valid indexer to `.loc` will work.

```
[13]: df.style.applymap(color_negative_red,
                        subset=pd.IndexSlice[2:5, ['B', 'D']])
```

```
[13]: <pandas.io.formats.style.Styler at 0x7f0fe5a06c90>
```

Only label-based slicing is supported right now, not positional.

If your style function uses a `subset` or `axis` keyword argument, consider wrapping your function in a `functools.partial`, partialing out that keyword.

```
my_func2 = functools.partial(my_func, subset=42)
```

2.16.3 Finer Control: Display Values

We distinguish the *display* value from the *actual* value in `Styler`. To control the display value, the text is printed in each cell, use `Styler.format`. Cells can be formatted according to a format spec string or a callable that takes a single value and returns a string.

```
[14]: df.style.format("{:.2%}")
```

```
[14]: <pandas.io.formats.style.Styler at 0x7f0fe5a83150>
```

Use a dictionary to format specific columns.

```
[15]: df.style.format({'B': "{:0<4.0f}", 'D': '{:+.2f}'})
```

```
[15]: <pandas.io.formats.style.Styler at 0x7f0fe5a06e50>
```

Or pass in a callable (or dictionary of callables) for more flexible handling.

```
[16]: df.style.format({"B": lambda x: "±{:.2f}".format(abs(x))})
```

```
[16]: <pandas.io.formats.style.Styler at 0x7f0fe5a1b0d0>
```

You can format the text displayed for missing values by `na_rep`.

```
[17]: df.style.format("{:.2%}", na_rep="-")
```

```
[17]: <pandas.io.formats.style.Styler at 0x7f0fe5a83710>
```

These formatting techniques can be used in combination with styling.

```
[18]: df.style.highlight_max().format(None, na_rep="-")
```

```
[18]: <pandas.io.formats.style.Styler at 0x7f0fe59fc390>
```

2.16.4 Builtin styles

Finally, we expect certain styling functions to be common enough that we've included a few "built-in" to the `Styler`, so you don't have to write them yourself.

```
[19]: df.style.highlight_null(null_color='red')
```

```
[19]: <pandas.io.formats.style.Styler at 0x7f0fe5a06690>
```

You can create "heatmaps" with the `background_gradient` method. These require matplotlib, and we'll use Seaborn to get a nice colormap.

```
[20]: import seaborn as sns

cm = sns.light_palette("green", as_cmap=True)
```

(continues on next page)

(continued from previous page)

```
s = df.style.background_gradient(cmap=cm)
s
```

[20]: <pandas.io.formats.style.Styler at 0x7f0fe5acccd0>

`Styler.background_gradient` takes the keyword arguments `low` and `high`. Roughly speaking these extend the range of your data by `low` and `high` percent so that when we convert the colors, the colormap's entire range isn't used. This is useful so that you can actually read the text still.

```
[21]: # Uses the full color range
df.loc[:4].style.background_gradient(cmap='viridis')
```

[21]: <pandas.io.formats.style.Styler at 0x7f0fe2933dd0>

```
[22]: # Compress the color range
(df.loc[:4]
    .style
    .background_gradient(cmap='viridis', low=.5, high=0)
    .highlight_null('red'))
```

[22]: <pandas.io.formats.style.Styler at 0x7f0fe293cc90>

There's also `.highlight_min` and `.highlight_max`.

```
[23]: df.style.highlight_max(axis=0)
```

[23]: <pandas.io.formats.style.Styler at 0x7f0fe2939f90>

Use `Styler.set_properties` when the style doesn't actually depend on the values.

```
[24]: df.style.set_properties(**{'background-color': 'black',
                           'color': 'lawngreen',
                           'border-color': 'white'})
```

[24]: <pandas.io.formats.style.Styler at 0x7f0fe2939510>

Bar charts

You can include "bar charts" in your DataFrame.

```
[25]: df.style.bar(subset=['A', 'B'], color='#d65f5f')
```

[25]: <pandas.io.formats.style.Styler at 0x7f0fe2954d50>

New in version 0.20.0 is the ability to customize further the bar chart: You can now have the `df.style.bar` be centered on zero or midpoint value (in addition to the already existing way of having the min value at the left side of the cell), and you can pass a list of `[color_negative, color_positive]`.

Here's how you can change the above with the new `align='mid'` option:

```
[26]: df.style.bar(subset=['A', 'B'], align='mid', color=['#d65f5f', '#5fba7d'])
```

[26]: <pandas.io.formats.style.Styler at 0x7f0fe29390d0>

The following example aims to give a highlight of the behavior of the new align options:

[27]:
```
import pandas as pd
from IPython.display import HTML

# Test series
test1 = pd.Series([-100,-60,-30,-20], name='All Negative')
test2 = pd.Series([10,20,50,100], name='All Positive')
test3 = pd.Series([-10,-5,0,90], name='Both Pos and Neg')

head = """
<table>
    <thead>
        <th>Align</th>
        <th>All Negative</th>
        <th>All Positive</th>
        <th>Both Neg and Pos</th>
    </thead>
    </tbody>

"""

aligns = ['left','zero','mid']
for align in aligns:
    row = "<tr><th>{}</th>".format(align)
    for serie in [test1,test2,test3]:
        s = serie.copy()
        s.name=''
        row += "<td>{}</td>".format(s.to_frame().style.bar(align=align,
                                                           color=['#d65f5f', '#5fba7d
↪'],
                                                           width=100).render())
↪#testn['width']
    row += '</tr>'
    head += row

head+= """
</tbody>
</table>"""


HTML(head)
```

[27]:
```
<IPython.core.display.HTML object>
```

2.16.5 Sharing styles

Say you have a lovely style built up for a DataFrame, and now you want to apply the same style to a second DataFrame. Export the style with `df1.style.export`, and import it on the second DataFrame with `df1.style.set`

[28]:
```
df2 = -df
style1 = df.style.applymap(color_negative_red)
style1
```

[28]:
```
<pandas.io.formats.style.Styler at 0x7f0fe2905250>
```

[29]:
```
style2 = df2.style
style2.use(style1.export())
style2
```

[29]:
```
<pandas.io.formats.style.Styler at 0x7f0fe28f85d0>
```

Notice that you're able to share the styles even though they're data aware. The styles are re-evaluated on the new DataFrame they've been `used` upon.

2.16.6 Other Options

You've seen a few methods for data-driven styling. `Styler` also provides a few other options for styles that don't depend on the data.

- precision
- captions
- table-wide styles
- missing values representation
- hiding the index or columns

Each of these can be specified in two ways:

- A keyword argument to `Styler.__init__`
- A call to one of the `.set_` or `.hide_` methods, e.g. `.set_caption` or `.hide_columns`

The best method to use depends on the context. Use the `Styler` constructor when building many styled DataFrames that should all share the same properties. For interactive use, the `.set_` and `.hide_` methods are more convenient.

Precision

You can control the precision of floats using pandas' regular `display.precision` option.

[30]:
```
with pd.option_context('display.precision', 2):
    html = (df.style
              .applymap(color_negative_red)
              .apply(highlight_max))
html
```

[30]:
```
<pandas.io.formats.style.Styler at 0x7f0fe28f7410>
```

Or through a `set_precision` method.

[31]:
```
df.style\
  .applymap(color_negative_red)\
  .apply(highlight_max)\
  .set_precision(2)
```

[31]:
```
<pandas.io.formats.style.Styler at 0x7f0fe5a1b410>
```

Setting the precision only affects the printed number; the full-precision values are always passed to your style functions. You can always use `df.round(2).style` if you'd prefer to round from the start.

Captions

Regular table captions can be added in a few ways.

[32]:
```
df.style.set_caption('Colormaps, with a caption.')\
    .background_gradient(cmap=cm)
```

[32]:
```
<pandas.io.formats.style.Styler at 0x7f0fe2911110>
```

Table styles

The next option you have are "table styles". These are styles that apply to the table as a whole, but don't look at the data. Certain sytlings, including pseudo-selectors like `:hover` can only be used this way.

[33]:
```
from IPython.display import HTML

def hover(hover_color="#ffff99"):
    return dict(selector="tr:hover",
                props=[("background-color", "%s" % hover_color)])

styles = [
    hover(),
    dict(selector="th", props=[("font-size", "150%"),
                               ("text-align", "center")]),
    dict(selector="caption", props=[("caption-side", "bottom")])
]
html = (df.style.set_table_styles(styles)
          .set_caption("Hover to highlight."))
html
```

[33]:
```
<pandas.io.formats.style.Styler at 0x7f0fe2913310>
```

`table_styles` should be a list of dictionaries. Each dictionary should have the `selector` and `props` keys. The value for `selector` should be a valid CSS selector. Recall that all the styles are already attached to an `id`, unique to each `Styler`. This selector is in addition to that `id`. The value for `props` should be a list of tuples of `('attribute', 'value')`.

`table_styles` are extremely flexible, but not as fun to type out by hand. We hope to collect some useful ones either in pandas, or preferable in a new package that *builds on top* the tools here.

Missing values

You can control the default missing values representation for the entire table through `set_na_rep` method.

[34]:
```
(df.style
   .set_na_rep("FAIL")
   .format(None, na_rep="PASS", subset=["D"])
   .highlight_null("yellow"))
```

[34]:
```
<pandas.io.formats.style.Styler at 0x7f0fe2911fd0>
```

Hiding the Index or Columns

The index can be hidden from rendering by calling `Styler.hide_index`. Columns can be hidden from rendering by calling `Styler.hide_columns` and passing in the name of a column, or a slice of columns.

```
[35]: df.style.hide_index()
```

```
[35]: <pandas.io.formats.style.Styler at 0x7f0fe2905950>
```

```
[36]: df.style.hide_columns(['C','D'])
```

```
[36]: <pandas.io.formats.style.Styler at 0x7f0fe2913e90>
```

CSS classes

Certain CSS classes are attached to cells.

- Index and Column names include `index_name` and `level<k>` where `k` is its level in a MultiIndex
- Index label cells include
  - `row_heading`
  - `row<n>` where `n` is the numeric position of the row
  - `level<k>` where `k` is the level in a MultiIndex
- Column label cells include
  - `col_heading`
  - `col<n>` where `n` is the numeric position of the column
  - `level<k>` where `k` is the level in a MultiIndex
- Blank cells include `blank`
- Data cells include `data`

Limitations

- DataFrame only `(use Series.to_frame().style)`
- The index and columns must be unique
- No large repr, and performance isn't great; this is intended for summary DataFrames
- You can only style the *values*, not the index or columns
- You can only apply styles, you can't insert new HTML entities

Some of these will be addressed in the future.

Terms

- Style function: a function that's passed into `Styler.apply` or `Styler.applymap` and returns values like `'css attribute: value'`
- Builtin style functions: style functions that are methods on `Styler`
- table style: a dictionary with the two keys `selector` and `props`. `selector` is the CSS selector that `props` will apply to. `props` is a list of `(attribute, value)` tuples. A list of table styles passed into `Styler`.

2.16.7 Fun stuff

Here are a few interesting examples.

`Styler` interacts pretty well with widgets. If you're viewing this online instead of running the notebook yourself, you're missing out on interactively adjusting the color palette.

[37]:
```
from IPython.html import widgets
@widgets.interact
def f(h_neg=(0, 359, 1), h_pos=(0, 359), s=(0., 99.9), l=(0., 99.9)):
    return df.style.background_gradient(
        cmap=sns.palettes.diverging_palette(h_neg=h_neg, h_pos=h_pos, s=s, l=l,
                                            as_cmap=True)
    )
```

```
<pandas.io.formats.style.Styler at 0x7f0fe2905850>
```

[38]:
```
def magnify():
    return [dict(selector="th",
                 props=[("font-size", "4pt")]),
            dict(selector="td",
                 props=[('padding', "0em 0em")]),
            dict(selector="th:hover",
                 props=[("font-size", "12pt")]),
            dict(selector="tr:hover td:hover",
                 props=[('max-width', '200px'),
                        ('font-size', '12pt')])
]
```

[39]:
```
np.random.seed(25)
cmap = cmap=sns.diverging_palette(5, 250, as_cmap=True)
bigdf = pd.DataFrame(np.random.randn(20, 25)).cumsum()

bigdf.style.background_gradient(cmap, axis=1)\
    .set_properties(**{'max-width': '80px', 'font-size': '1pt'})\
    .set_caption("Hover to magnify")\
    .set_precision(2)\
    .set_table_styles(magnify())
```

[39]:
```
<pandas.io.formats.style.Styler at 0x7f0fe2939310>
```

2.16.8 Export to Excel

New in version 0.20.0

Experimental: This is a new feature and still under development. We'll be adding features and possibly making breaking changes in future releases. We'd love to hear your feedback.

Some support is available for exporting styled `DataFrames` to Excel worksheets using the `OpenPyXL` or `XlsxWriter` engines. CSS2.2 properties handled include:

- `background-color`
- `border-style`, `border-width`, `border-color` and their {`top`, `right`, `bottom`, `left` variants}
- `color`
- `font-family`
- `font-style`
- `font-weight`
- `text-align`
- `text-decoration`
- `vertical-align`
- `white-space: nowrap`
- Only CSS2 named colors and hex colors of the form `#rgb` or `#rrggbb` are currently supported.
- The following pseudo CSS properties are also available to set excel specific style properties:
  - `number-format`

[40]:
```
df.style.\
    applymap(color_negative_red).\
    apply(highlight_max).\
    to_excel('styled.xlsx', engine='openpyxl')
```

A screenshot of the output:

	A	B	C	D	E	F
1		A	B	C	D	E
2	0	1	1.329212		-0.31628	-0.99081
3	1	2	-1.070816	-1.438713	0.564417	0.295722
4	2	3	-1.626404	0.219565	0.678805	1.889273
5	3	4	0.961538	0.104011	-0.481165	0.850229
6	4	5	1.453425	1.057737	0.165562	0.515018
7	5	6	-1.336936	0.562861	1.392855	-0.063328
8	6	7	0.121668	1.207603	-0.00204	1.627796
9	7	8	0.354493	1.037528	-0.385684	0.519818
10	8	9	1.686583	-1.325963	1.428984	-2.089354
11	9	10	-0.12982	0.631523	-0.586538	0.29072

2.16.9 Extensibility

The core of pandas is, and will remain, its "high-performance, easy-to-use data structures". With that in mind, we hope that `DataFrame.style` accomplishes two goals

- Provide an API that is pleasing to use interactively and is "good enough" for many tasks
- Provide the foundations for dedicated libraries to build on

If you build a great library on top of this, let us know and we'll link to it.

Subclassing

If the default template doesn't quite suit your needs, you can subclass Styler and extend or override the template. We'll show an example of extending the default template to insert a custom header before each table.

[41]:
```
from jinja2 import Environment, ChoiceLoader, FileSystemLoader
from IPython.display import HTML
from pandas.io.formats.style import Styler
```

We'll use the following template:

[42]:
```
with open("templates/myhtml.tpl") as f:
    print(f.read())
```

```
{% extends "html.tpl" %}
{% block table %}
<h1>{{ table_title|default("My Table") }}</h1>
{{ super() }}
{% endblock table %}
```

Now that we've created a template, we need to set up a subclass of `Styler` that knows about it.

[43]:
```
class MyStyler(Styler):
    env = Environment(
        loader=ChoiceLoader([
            FileSystemLoader("templates"),  # contains ours
            Styler.loader,  # the default
        ])
    )
    template = env.get_template("myhtml.tpl")
```

Notice that we include the original loader in our environment's loader. That's because we extend the original template, so the Jinja environment needs to be able to find it.

Now we can use that custom styler. It's `__init__` takes a DataFrame.

[44]:
```
MyStyler(df)
```

[44]:
```
<__main__.MyStyler at 0x7f0fe0029090>
```

Our custom template accepts a `table_title` keyword. We can provide the value in the `.render` method.

[45]:
```
HTML(MyStyler(df).render(table_title="Extending Example"))
```

[45]:
```
<IPython.core.display.HTML object>
```

For convenience, we provide the `Styler.from_custom_template` method that does the same as the custom subclass.

```
[46]: EasyStyler = Styler.from_custom_template("templates", "myhtml.tpl")
      EasyStyler(df)
```

```
[46]: <pandas.io.formats.style.Styler.from_custom_template.<locals>.MyStyler at␣
      ↪0x7f0fe00384d0>
```

Here's the template structure:

```
[47]: with open("templates/template_structure.html") as f:
          structure = f.read()

      HTML(structure)
```

```
[47]: <IPython.core.display.HTML object>
```

See the template in the GitHub repo for more details.

2.17 Options and settings

2.17.1 Overview

pandas has an options system that lets you customize some aspects of its behaviour, display-related options being those the user is most likely to adjust.

Options have a full "dotted-style", case-insensitive name (e.g. `display.max_rows`). You can get/set options directly as attributes of the top-level `options` attribute:

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

In [2]: pd.options.display.max_rows
Out[2]: 15

In [3]: pd.options.display.max_rows = 999

In [4]: pd.options.display.max_rows
Out[4]: 999
```

The API is composed of 5 relevant functions, available directly from the `pandas` namespace:

- *`get_option()`* / *`set_option()`* - get/set the value of a single option.
- *`reset_option()`* - reset one or more options to their default value.
- *`describe_option()`* - print the descriptions of one or more options.
- *`option_context()`* - execute a codeblock with a set of options that revert to prior settings after execution.

Note: Developers can check out pandas/core/config.py for more information.

All of the functions above accept a regexp pattern (`re.search` style) as an argument, and so passing in a substring will work - as long as it is unambiguous:

```
In [5]: pd.get_option("display.max_rows")
Out[5]: 999

In [6]: pd.set_option("display.max_rows", 101)

In [7]: pd.get_option("display.max_rows")
```

(continues on next page)

(continued from previous page)

```
Out[7]: 101

In [8]: pd.set_option("max_r", 102)

In [9]: pd.get_option("display.max_rows")
Out[9]: 102
```

The following will **not work** because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```
In [10]: try:
   ....:     pd.get_option("column")
   ....: except KeyError as e:
   ....:     print(e)
   ....:
'Pattern matched multiple keys'
```

Note: Using this form of shorthand may cause your code to break if new options with similar names are added in future versions.

You can get a list of available options and their descriptions with `describe_option`. When called with no argument `describe_option` will print out the descriptions for all available options.

2.17.2 Getting and setting options

As described above, *get_option()* and *set_option()* are available from the pandas namespace. To change an option, call `set_option('option regex', new_value)`.

```
In [11]: pd.get_option('mode.sim_interactive')
Out[11]: False

In [12]: pd.set_option('mode.sim_interactive', True)

In [13]: pd.get_option('mode.sim_interactive')
Out[13]: True
```

Note: The option 'mode.sim_interactive' is mostly used for debugging purposes.

All options also have a default value, and you can use `reset_option` to do just that:

```
In [14]: pd.get_option("display.max_rows")
Out[14]: 60

In [15]: pd.set_option("display.max_rows", 999)

In [16]: pd.get_option("display.max_rows")
Out[16]: 999

In [17]: pd.reset_option("display.max_rows")

In [18]: pd.get_option("display.max_rows")
Out[18]: 60
```

It's also possible to reset multiple options at once (using a regex):