Style your Data Analysis

Building styles

Let's see some examples.

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

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:

In [2]:
```
df.style
```

Out[2]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

In [3]:
```
df.style.highlight_null().render().split('\n')[:10]
```

Out[3]:
```
['<style  type="text/css" >',
 '#T_dc243_row0_col2,#T_dc243_row3_col3{',
 '            background-color:  red;',
 '        }</style><table id="T_dc243_" ><thead>    <tr>        <th class="blank level0" ></th>        <th class="col_heading level0 col0" >A</th>        <th class="col_hea
ding 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_dc243_level0_row0" class="row_heading level0 row0" >0</th>',
 '                        <td id="T_dc243_row0_col0" class="data row0 col0" >1.000000</td>',
 '                        <td id="T_dc243_row0_col1" class="data row0 col1" >1.329212</td>',
 '                        <td id="T_dc243_row0_col2" class="data row0 col2" >nan</td>',
 '                        <td id="T_dc243_row0_col3" class="data row0 col3" >-0.316280</td>']
```

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

In [4]:
```
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 its own value. That means we should use the Styler.applymap method which works elementwise.

In [5]:
```
s = df.style.applymap(color_negative_red)
s
```

Out[5]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354

	A	B	C	D	E
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

In [6]:
```
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]
```

In [7]:
```
df.style.apply(highlight_max)
```

Out[7]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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.

In [8]:
```
df.style.\
    applymap(color_negative_red).\
    apply(highlight_max)
```

Out[8]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722

	A	B	C	D	E
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

In [9]:

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

In [10]:

```
df.style.apply(highlight_max, color='darkorange', axis=None)
```

Out[10]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796

	A	B	C	D	E
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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***.

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, columnindexer)*

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

In [11]:
```
df.style.apply(highlight_max, subset=['B', 'C', 'D'])
```

Out[11]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722

	A	B	C	D	E
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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

In [12]:
```
df.style.applymap(color_negative_red, subset=pd.IndexSlice[2:5, ['B', 'D']])
```

Out[12]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

Finer Control: Display Values

In [13]:
```
df.style.format("{:.2%}")
```

Out[13]:

	A	B	C	D	E

	A	B	C	D	E
0	100.00%	132.92%	nan%	-31.63%	-99.08%
1	200.00%	-107.08%	-143.87%	56.44%	29.57%
2	300.00%	-162.64%	21.96%	67.88%	188.93%
3	400.00%	96.15%	10.40%	nan%	85.02%
4	500.00%	145.34%	105.77%	16.56%	51.50%
5	600.00%	-133.69%	56.29%	139.29%	-6.33%
6	700.00%	12.17%	120.76%	-0.20%	162.78%
7	800.00%	35.45%	103.75%	-38.57%	51.98%
8	900.00%	168.66%	-132.60%	142.90%	-208.94%
9	1000.00%	-12.98%	63.15%	-58.65%	29.07%

Use a dictionary to format specific columns.

In [14]:
```
df.style.format({'B': "{:0<4.0f}", 'D': '{:+.2f}'})
```

Out[14]:

	A	B	C	D	E
0	1.000000	1000	nan	-0.32	-0.990810
1	2.000000	-100	-1.438713	+0.56	0.295722
2	3.000000	-200	0.219565	+0.68	1.889273
3	4.000000	1000	0.104011	+nan	0.850229
4	5.000000	1000	1.057737	+0.17	0.515018
5	6.000000	-100	0.562861	+1.39	-0.063328
6	7.000000	0000	1.207603	-0.00	1.627796
7	8.000000	0000	1.037528	-0.39	0.519818
8	9.000000	2000	-1.325963	+1.43	-2.089354
9	10.000000	-000	0.631523	-0.59	0.290720

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

In [15]:
```
df.style.format({"B": lambda x: "±{:.2f}".format(abs(x))})
```

Out[15]:

	A	B	C	D	E
0	1.000000	±1.33	nan	-0.316280	-0.990810
1	2.000000	±1.07	-1.438713	0.564417	0.295722
2	3.000000	±1.63	0.219565	0.678805	1.889273
3	4.000000	±0.96	0.104011	nan	0.850229
4	5.000000	±1.45	1.057737	0.165562	0.515018
5	6.000000	±1.34	0.562861	1.392855	-0.063328
6	7.000000	±0.12	1.207603	-0.002040	1.627796
7	8.000000	±0.35	1.037528	-0.385684	0.519818
8	9.000000	±1.69	-1.325963	1.428984	-2.089354
9	10.000000	±0.13	0.631523	-0.586538	0.290720

You can format the text displayed for missing values by *_narep*.

In [16]:
```
df.style.format("{:.2%}", na_rep="-")
```

Out[16]:

	A	B	C	D	E
0	100.00%	132.92%	-	-31.63%	-99.08%
1	200.00%	-107.08%	-143.87%	56.44%	29.57%
2	300.00%	-162.64%	21.96%	67.88%	188.93%
3	400.00%	96.15%	10.40%	-	85.02%
4	500.00%	145.34%	105.77%	16.56%	51.50%
5	600.00%	-133.69%	56.29%	139.29%	-6.33%
6	700.00%	12.17%	120.76%	-0.20%	162.78%
7	800.00%	35.45%	103.75%	-38.57%	51.98%
8	900.00%	168.66%	-132.60%	142.90%	-208.94%
9	1000.00%	-12.98%	63.15%	-58.65%	29.07%

These formatting techniques can be used in combination with styling.

In [17]:
```
df.style.highlight_max().format(None, na_rep="-")
```

Out[17]:

	A	B	C	D	E
0	1.000000	1.329212	-	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	-	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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.

In [18]:
```
df.style.highlight_null(null_color='red')
```

Out[18]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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

In [19]:
```
import seaborn as sns

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

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

Out[19]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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.

In [20]:
```
# Uses the full color range
df.loc[:4].style.background_gradient(cmap='viridis')
```

Out[20]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018

In [21]:
```
# Compress the color range
(df.loc[:4]
```

```
    .style
    .background_gradient(cmap='viridis', low=.5, high=0)
    .highlight_null('red'))
```

Out[21]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018

There's also .highlight_min and .highlight_max.

In [22]:

```
df.style.highlight_max(axis=0)
```

Out[22]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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

In [23]:

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

Out[23]:

	A	B	C	D	E

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

Bar charts

You can include “bar charts” in your DataFrame.

In [24]:
```
df.style.bar(subset=['A', 'B'], color='#d65f5f')
```

Out[24]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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

In [25]:
```
df.style.bar(subset=['A', 'B'], align='mid', color=['#d65f5f', '#5fba7d'])
```

Out[25]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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

In [26]:
```
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 series in [test1,test2,test3]:
```

```
        s = series.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)
```

Out[26]:

Align	All Negative	All Positive	Both Neg and Pos
left	0 -100 1 -60 2 -30 3 -20	0 10 1 20 2 50 3 100	0 -10 1 -5 2 0 3 90
zero	0 -100 1 -60 2 -30 3 -20	0 10 1 20 2 50 3 100	0 -10 1 -5 2 0 3 90
mid	0 -100 1 -60 2 -30 3 -20	0 10 1 20 2 50 3 100	0 -10 1 -5 2 0 3 90

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*

In [27]:
```
df2 = -df
style1 = df.style.applymap(color_negative_red)
style1
```

Out[27]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

In [28]:
```
style2 = df2.style
style2.use(style1.export())
style2
```

Out[28]:

	A	B	C	D	E
0	-1.000000	-1.329212	nan	0.316280	0.990810
1	-2.000000	1.070816	1.438713	-0.564417	-0.295722
2	-3.000000	1.626404	-0.219565	-0.678805	-1.889273
3	-4.000000	-0.961538	-0.104011	nan	-0.850229
4	-5.000000	-1.453425	-1.057737	-0.165562	-0.515018
5	-6.000000	1.336936	-0.562861	-1.392855	0.063328
6	-7.000000	-0.121668	-1.207603	0.002040	-1.627796
7	-8.000000	-0.354493	-1.037528	0.385684	-0.519818
8	-9.000000	-1.686583	1.325963	-1.428984	2.089354

	A	B	C	D	E
9	-10.000000	0.129820	-0.631523	0.586538	-0.290720

Precision

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

In [29]:
```
with pd.option_context('display.precision', 2):
    html = (df.style
              .applymap(color_negative_red)
              .apply(highlight_max))
html
```

Out[29]:

	A	B	C	D	E
0	1.00	1.33	nan	-0.32	-0.99
1	2.00	-1.07	-1.44	0.56	0.30
2	3.00	-1.63	0.22	0.68	1.89
3	4.00	0.96	0.10	nan	0.85
4	5.00	1.45	1.06	0.17	0.52
5	6.00	-1.34	0.56	1.39	-0.06
6	7.00	0.12	1.21	-0.00	1.63
7	8.00	0.35	1.04	-0.39	0.52
8	9.00	1.69	-1.33	1.43	-2.09
9	10.00	-0.13	0.63	-0.59	0.29

Or through a set_precision method.

In [30]:
```
df.style\
  .applymap(color_negative_red)\
  .apply(highlight_max)\
  .set_precision(2)
```

Out[30]:

	A	B	C	D	E
0	1.00	1.33	nan	-0.32	-0.99

	A	B	C	D	E
1	2.00	-1.07	-1.44	0.56	0.30
2	3.00	-1.63	0.22	0.68	1.89
3	4.00	0.96	0.10	nan	0.85
4	5.00	1.45	1.06	0.17	0.52
5	6.00	-1.34	0.56	1.39	-0.06
6	7.00	0.12	1.21	-0.00	1.63
7	8.00	0.35	1.04	-0.39	0.52
8	9.00	1.69	-1.33	1.43	-2.09
9	10.00	-0.13	0.63	-0.59	0.29

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.

In [31]:
```
df.style.set_caption('Colormaps, with a caption.')\
    .background_gradient(cmap=cm)
```

Out[31]:

Colormaps, with a caption.

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354

	A	B	C	D	E
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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 stylings, including pseudo-selectors like :hover can only be used this way. These can also be used to set specific row or column based class selectors, as will be shown.

In [32]:
```
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
```

Out[32]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796

	A	B	C	D	E
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

Hover to highlight.

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.

table_styles can be used to add column and row based class descriptors. For large tables this can increase performance by avoiding repetitive individual css for each cell, and it can also simplify style construction in some cases. If table_styles is given as a dictionary each key should be a specified column or index value and this will map to specific class CSS selectors of the given column or row.

Note that Styler.set_table_styles will overwrite existing styles but can be chained by setting the overwrite argument to False.

In [33]:
```
html = html.set_table_styles({
    'B': [dict(selector='', props=[('color', 'green')])],
    'C': [dict(selector='td', props=[('color', 'red')])],
    }, overwrite=False)
html
```

Out[33]:

	A	B	C	D	E
0	1.000000	1.329212	nan	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	nan	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018

	A	B	C	D	E
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

Hover to highlight.

Missing values

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

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

Out[34]:

	A	B	C	D	E
0	1.000000	1.329212	FAIL	-0.316280	-0.990810
1	2.000000	-1.070816	-1.438713	0.564417	0.295722
2	3.000000	-1.626404	0.219565	0.678805	1.889273
3	4.000000	0.961538	0.104011	PASS	0.850229
4	5.000000	1.453425	1.057737	0.165562	0.515018
5	6.000000	-1.336936	0.562861	1.392855	-0.063328
6	7.000000	0.121668	1.207603	-0.002040	1.627796
7	8.000000	0.354493	1.037528	-0.385684	0.519818
8	9.000000	1.686583	-1.325963	1.428984	-2.089354
9	10.000000	-0.129820	0.631523	-0.586538	0.290720

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.

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

Out[35]:

A	B	C	D	E
1.000000	1.329212	nan	-0.316280	-0.990810
2.000000	-1.070816	-1.438713	0.564417	0.295722
3.000000	-1.626404	0.219565	0.678805	1.889273
4.000000	0.961538	0.104011	nan	0.850229
5.000000	1.453425	1.057737	0.165562	0.515018
6.000000	-1.336936	0.562861	1.392855	-0.063328
7.000000	0.121668	1.207603	-0.002040	1.627796
8.000000	0.354493	1.037528	-0.385684	0.519818
9.000000	1.686583	-1.325963	1.428984	-2.089354
10.000000	-0.129820	0.631523	-0.586538	0.290720

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

Out[36]:

	A	B	E
0	1.000000	1.329212	-0.990810
1	2.000000	-1.070816	0.295722
2	3.000000	-1.626404	1.889273
3	4.000000	0.961538	0.850229
4	5.000000	1.453425	0.515018
5	6.000000	-1.336936	-0.063328
6	7.000000	0.121668	1.627796
7	8.000000	0.354493	0.519818
8	9.000000	1.686583	-2.089354

	A	B	E
9	10.000000	-0.129820	0.290720

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.

In [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)
    )
```

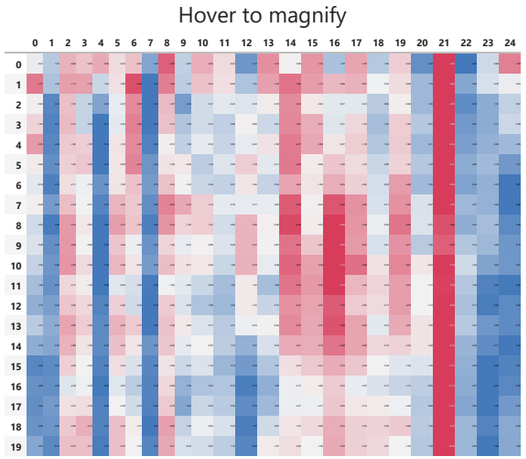
```
c:\users\trentino\appdata\local\programs\python\python39\lib\site-packages\IPython\html.py:12: ShimWarning: The `IPython.html` package has been deprecated since IPython 4.
0. You should import from `notebook` instead. `IPython.html.widgets` has moved to `ipywidgets`.
  warn("The `IPython.html` package has been deprecated since IPython 4.0. "
```

In [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')])
]
```

In [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())
```

Out[39]:


Hover to magnify
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19