Virtual Internship Experience

Programming Language Scripting

Conditional Formatting
Iteration

Conditional Formatting

Pandas is an important data science library and everybody involved in data science uses it extensively. It presents the data in the form of a table similar to what we see in excel. If you have worked with excel, you must be aware that you can customize your sheets, add colors to the cells, and mark important figures that need extra attention.

While working with pandas, have you ever thought about how you can do the same styling to dataframes to make them more appealing and explainable? Generating reports out of the dataframes is a good option but what if you can do the styling in the dataframe using Pandas only?

That's where the Pandas Style API comes to the rescue. This detailed article will go through all the features of Pandas styling, various types of built-in functions, creating our custom functions, and some of its advanced usages.

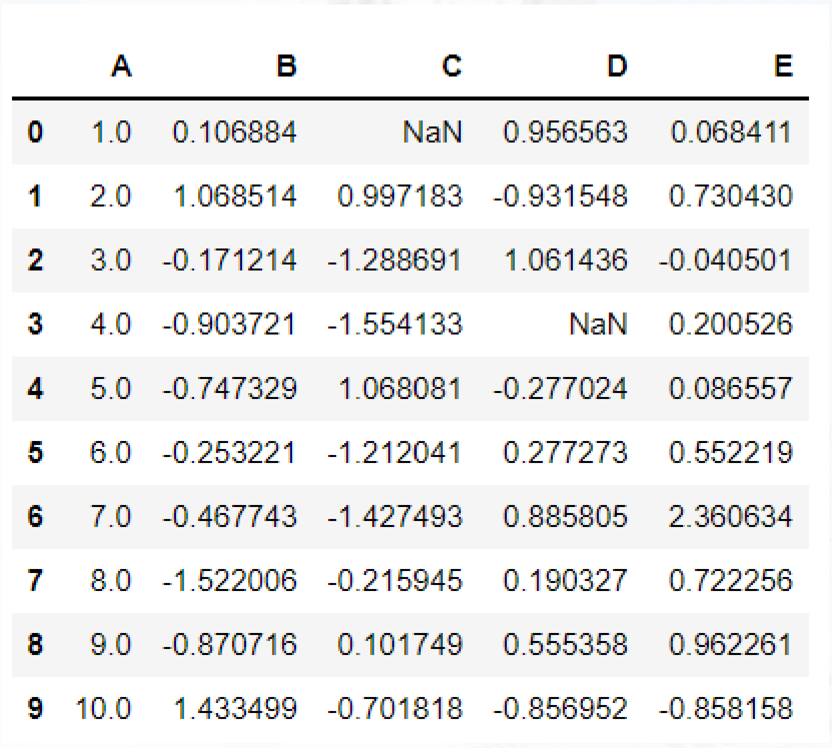
Introduction to Pandas Styling

A pandas dataframe is a tabular structure with rows and columns. One of the most popular environments for performing data-related tasks is Jupyter notebooks. These are web-based platform-independent IDEs. In Jupyter notebooks, the dataframe is rendered for display using HTML tags and CSS. This means that you can manipulate the styling of these web components.

We will see this in action in upcoming sections. For now, let's create a sample dataset and display the output dataframe.

```
import pandas as pd
import numpy as np
np.random.seed(88)
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
```

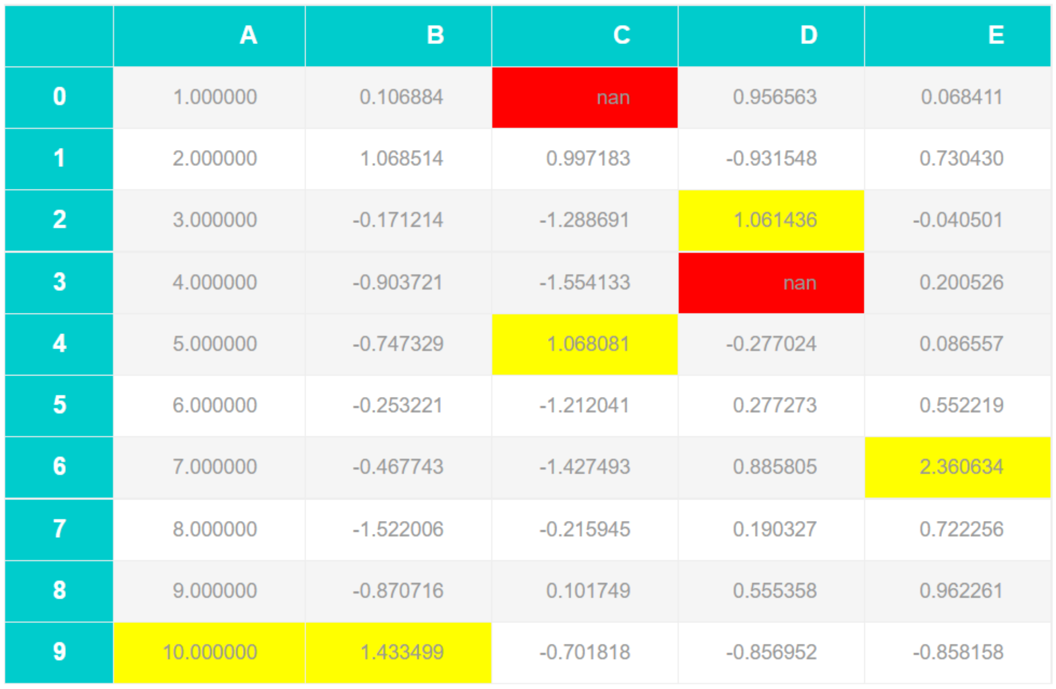
Output:



	A	B	C	D	E
0	1.0	0.106884	NaN	0.956563	0.068411
1	2.0	1.068514	0.997183	-0.931548	0.730430
2	3.0	-0.171214	-1.288691	1.061436	-0.040501
3	4.0	-0.903721	-1.554133	NaN	0.200526
4	5.0	-0.747329	1.068081	-0.277024	0.086557
5	6.0	-0.253221	-1.212041	0.277273	0.552219
6	7.0	-0.467743	-1.427493	0.885805	2.360634
7	8.0	-1.522006	-0.215945	0.190327	0.722256
8	9.0	-0.870716	0.101749	0.555358	0.962261
9	10.0	1.433499	-0.701818	-0.856952	-0.858158

Output Dataframe (Without Styling)

Doesn't this look boring to you? What if you transform this minimal table to this:



	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

Image by Author (Made in Pandas)

The transformed table above has:

1. Maximum values marked yellow for each column
2. Null values marked red for each column
3. More appealing table style, better fonts for header, and increased font size.

Now, we will be exploring all the possible ways of styling the dataframe and making it similar to what you saw above, so let's begin!

Styling the DataFrame

To access all the styling properties for the pandas dataframe, you need to use the accessor (Assume that dataframe object has been stored in variable "df"):

```
df.style
```

This accessor helps in the modification of the styler object (df.style), which controls the display of the dataframe on the web. Let's look at some of the methods to style the dataframe.

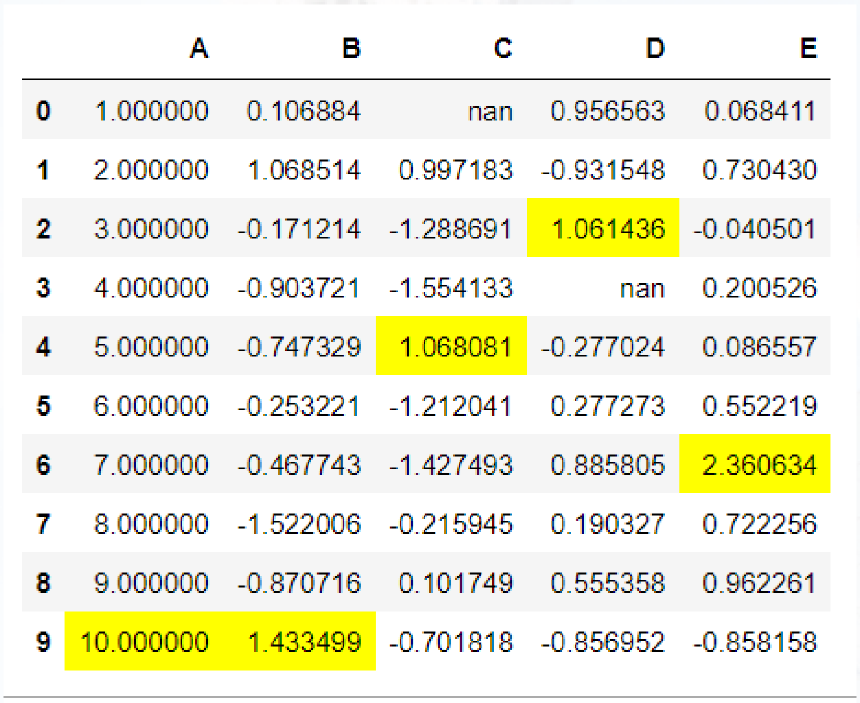
1. Highlight Min-Max values

   The dataframes can take a large number of values but when it is of a smaller size, then it makes sense to print out all the values of the dataframe. Now, you might be doing some type of analysis and you wanted to highlight the extreme values of the data. For this purpose, you can add style to your dataframe that highlights these extreme values.

   - For highlighting maximum values: Chain ".highlight_max()" function to the styler object. Additionally, you can also specify the axis for which you want to highlight the values. (axis=1: Rows, axis=0: Columns – default).]

     ```
     df.style.highlight_max()
     ```

     Output:

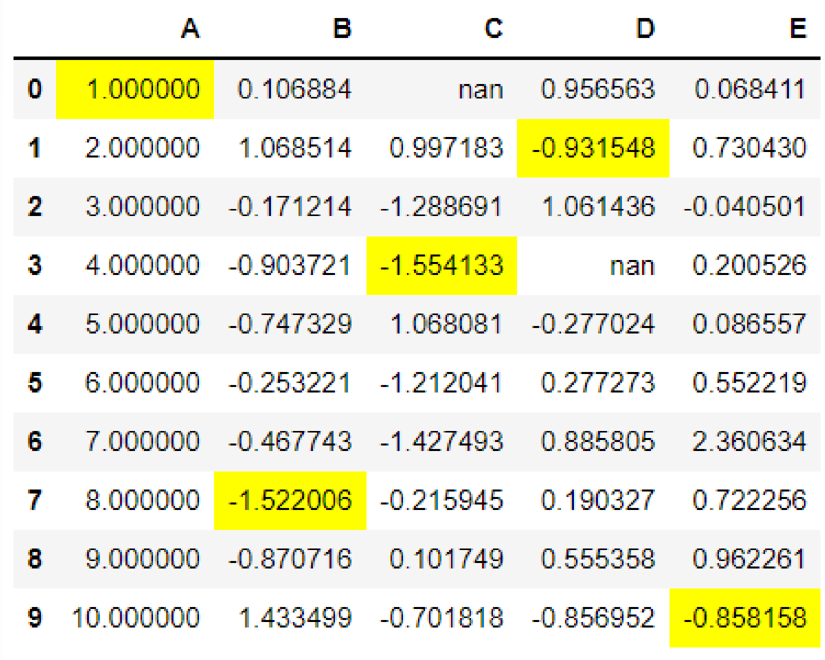


	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

- 1.2 For highlighting minimum values: Chain ".highlight_min()" function to the styler object. Here also, you can specify the axis at which these values will be highlighted.

```
df.style.highlight_min()
```

Output:



	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

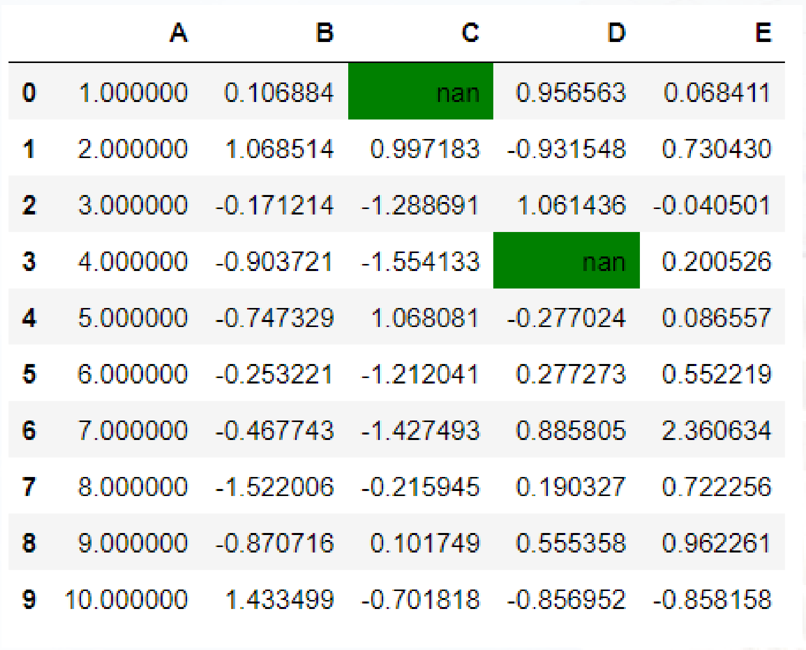
Both Min-Max highlight functions support the parameter "color" to change the highlight color from yellow.

2. Highlight Null values

Every dataset has some or the other null/missing values. These values should be either removed or handled in such a way that it doesn't introduce any biasness. To highlight such values, you can chain the ".highlight_null()" function to the styler object. This function doesn't support the axis parameter and the color control parameter here is "null_color" which takes the default value as "red".

```
df.style.highlight_null(null_color="green")
```

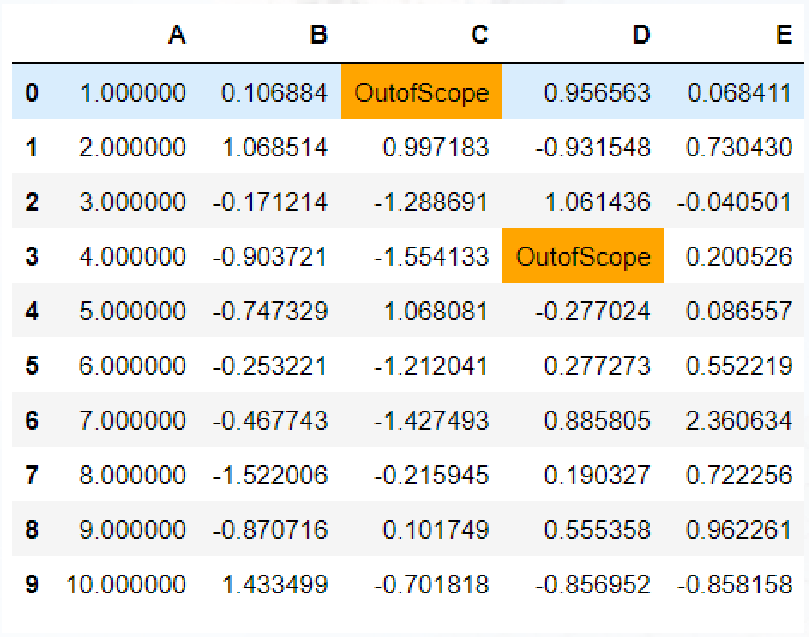
Output:



	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

set_na_rep(): Along with highlighting the missing values, they may be represented as "nan". You can change the representation of these missing values using the set_na_rep() function. This function can also be chained with any styler function but chaining it with highlight_null will provide more details.

```
df.style.set_na_rep("OutofScope").highlight_null(null_color="orange")
```

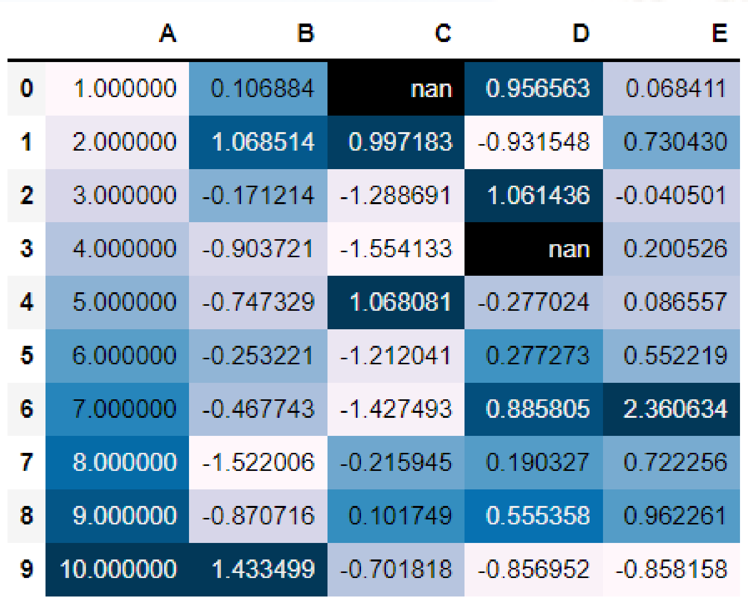


	A	B	C	D	E
0	1.000000	0.106884	OutofScope	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	OutofScope	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

3. Create Heatmap within dataframe
   Heatmaps are used to represent values with the color shades. The higher is the color shade, the larger is the value present. These color shades represent the intensity of values as compared to other values. To plot such a mapping in the dataframe itself, there is no direct function but the "styler.background_gradient()" workaround does the work.

```
df.style.background_gradient()
```

Output:



	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

There are few parameters you can pass to this function to further customize the output generated:

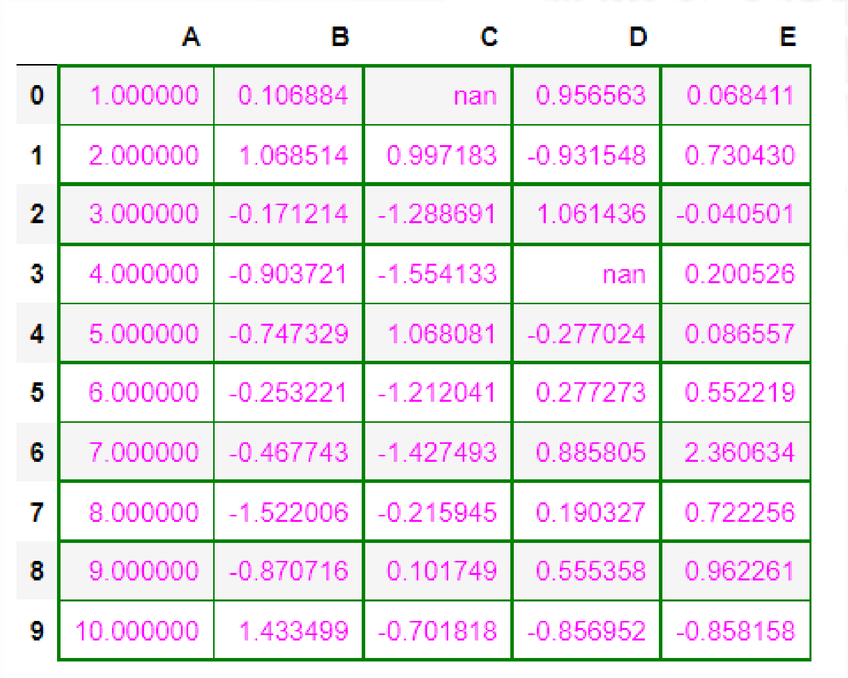
cmap: By default, the "PuBu" colormap is selected by pandas You can create a custom matplotlib colormap and pass it to the camp parameter.
axis: Generating heat plot via rows or columns criteria, by default: columns
text_color_threshold: Controls text visibility across varying background colors.

4. Table Properties
   As mentioned earlier also, the dataframe presented in the Jupyter notebooks is a table rendered using HTML and CSS. The table properties can be controlled using the "set_properties" method. This method is used to set one or more data-independent properties.
   This means that the modifications are done purely based on visual appearance and no significance as such. This method takes in the properties to be set as a dictionary.

   Example: Making table borders green with text color as purple.

```
df.style.set_properties(**{'border': '1.3px solid green',
                           'color': 'magenta'})
```

Output:



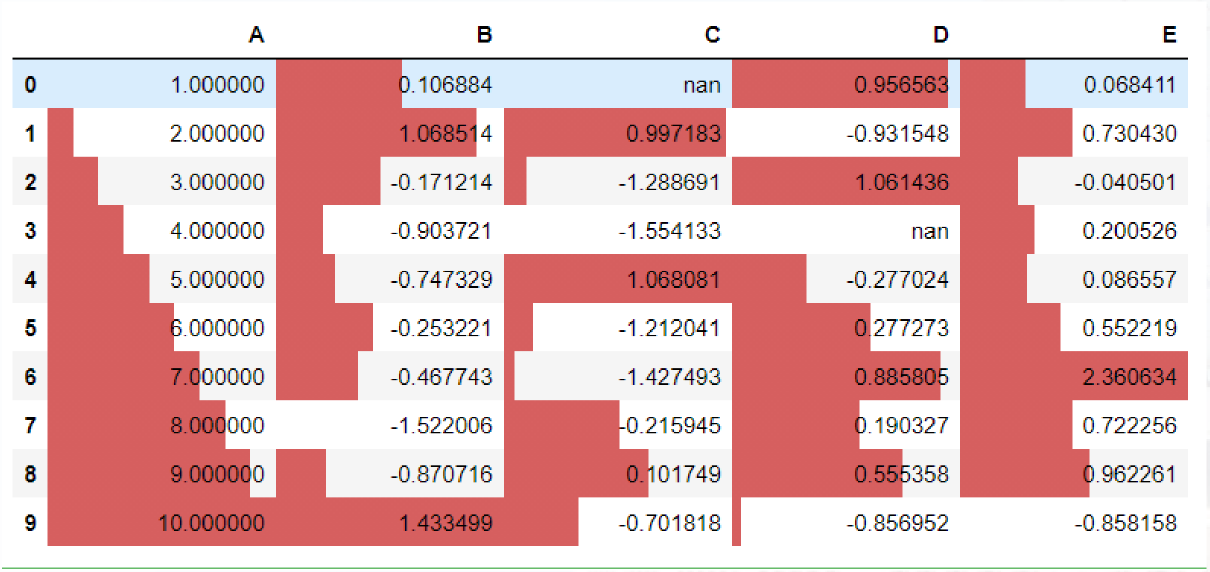
	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

5. Create Bar charts

Just as the heatmap, the bar charts can also be plotted within the dataframe itself. The bars are plotted in each cell depending upon the axis selected. By default, the axis=0 and the plot color are also fixed by pandas but it is configurable. To plot these bars, you simply need to chain the ".bar()" function to the styler object.

```
df.style.bar()
```

Output:

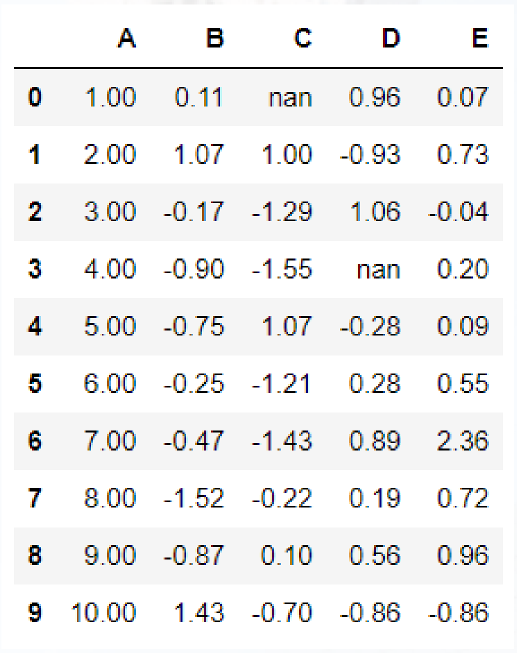


	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

6. Control precision
   The current values of the dataframe have float values and their decimals have no boundary condition. Even the column "A", which had to hold a single value is having too many decimal places. To control this behavior, you can use the ".set_precision()" function and pass the value for maximum decimals to be allowed.

```
df.style.set_precision(2)
```

Output:

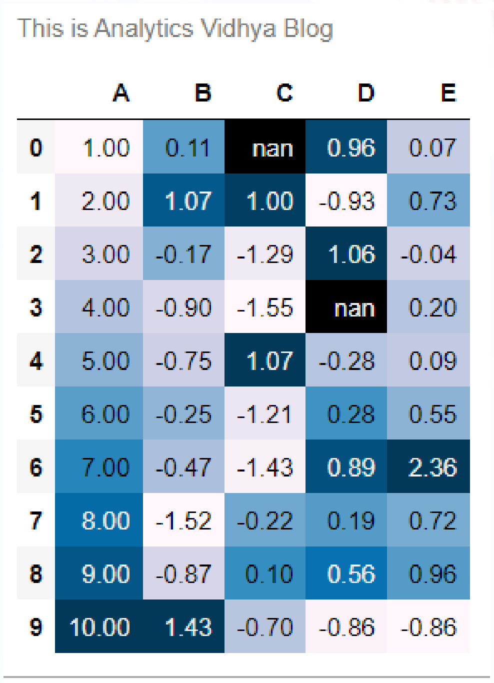


	A	B	C	D	E
0	1.00	0.11	nan	0.96	0.07
1	2.00	1.07	1.00	-0.93	0.73
2	3.00	-0.17	-1.29	1.06	-0.04
3	4.00	-0.90	-1.55	nan	0.20
4	5.00	-0.75	1.07	-0.28	0.09
5	6.00	-0.25	-1.21	0.28	0.55
6	7.00	-0.47	-1.43	0.89	2.36
7	8.00	-1.52	-0.22	0.19	0.72
8	9.00	-0.87	0.10	0.56	0.96
9	10.00	1.43	-0.70	-0.86	-0.86

7. Add Captions
   Like every image has a caption that defines the post text, you can add captions to your dataframes. This text will depict what the dataframe results talk about. They may be some sort of summary statistics like pivot tables.

```
df.style.set_caption("This is Analytics Vidhya Blog").set_precision(2).background_gradient()
```

Output:



This is Analytics Vidhya Blog

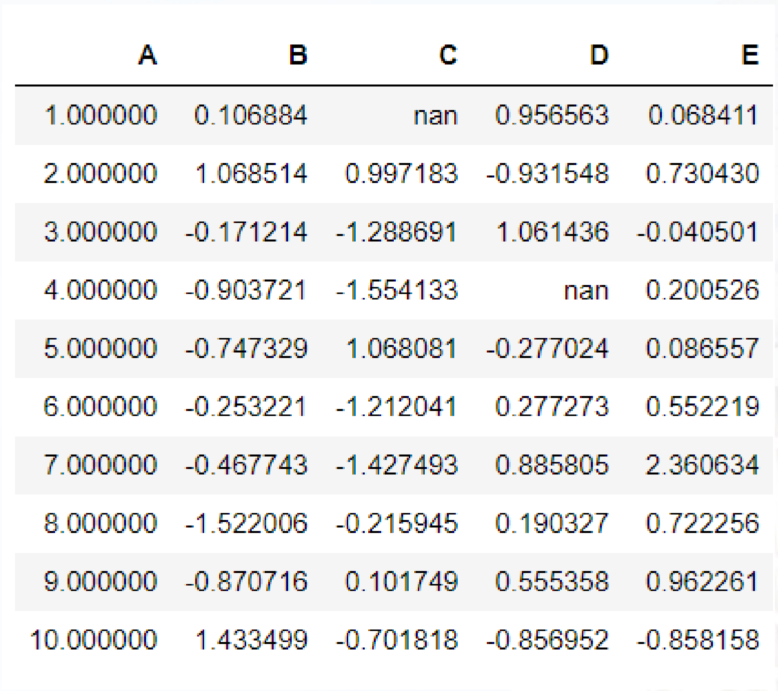
	A	B	C	D	E
0	1.00	0.11	nan	0.96	0.07
1	2.00	1.07	1.00	-0.93	0.73
2	3.00	-0.17	-1.29	1.06	-0.04
3	4.00	-0.90	-1.55	nan	0.20
4	5.00	-0.75	1.07	-0.28	0.09
5	6.00	-0.25	-1.21	0.28	0.55
6	7.00	-0.47	-1.43	0.89	2.36
7	8.00	-1.52	-0.22	0.19	0.72
8	9.00	-0.87	0.10	0.56	0.96
9	10.00	1.43	-0.70	-0.86	-0.86

8. Hiding Index or Column

   As the title suggests, you can hide the index or any particular column from the dataframe. Hiding index from the dataframe can be useful in cases when the index doesn't convey anything significant about the data. The column hiding depends on whether it is useful or not.

```
df.style.hide_index()
```

Output:



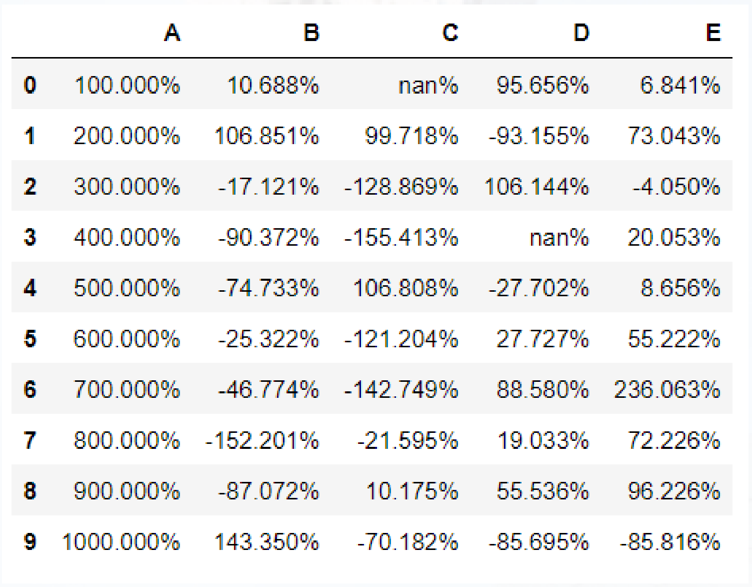
A	B	C	D	E
1.000000	0.106884	nan	0.956563	0.068411
2.000000	1.068514	0.997183	-0.931548	0.730430
3.000000	-0.171214	-1.288691	1.061436	-0.040501
4.000000	-0.903721	-1.554133	nan	0.200526
5.000000	-0.747329	1.068081	-0.277024	0.086557
6.000000	-0.253221	-1.212041	0.277273	0.552219
7.000000	-0.467743	-1.427493	0.885805	2.360634
8.000000	-1.522006	-0.215945	0.190327	0.722256
9.000000	-0.870716	0.101749	0.555358	0.962261
10.000000	1.433499	-0.701818	-0.856952	-0.858158

9. Control display values

   Using the styler object's ".format()" function, you can distinguish between the actual values held by the dataframe and the values you present. The "format" function takes in the format spec string that defines how individual values are presented. You can directly specify the specification which will apply to the whole dataset or you can pass the specific column on which you want to control the display values.

```
df.style.format("{:.3%}")
```

Output:

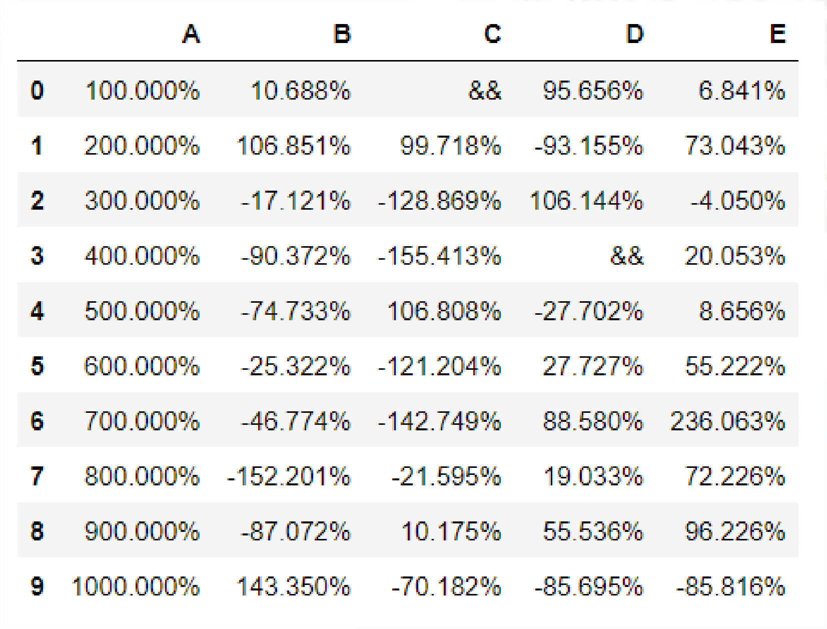


	A	B	C	D	E
0	100.000%	10.688%	nan%	95.656%	6.841%
1	200.000%	106.851%	99.718%	-93.155%	73.043%
2	300.000%	-17.121%	-128.869%	106.144%	-4.050%
3	400.000%	-90.372%	-155.413%	nan%	20.053%
4	500.000%	-74.733%	106.808%	-27.702%	8.656%
5	600.000%	-25.322%	-121.204%	27.727%	55.222%
6	700.000%	-46.774%	-142.749%	88.580%	236.063%
7	800.000%	-152.201%	-21.595%	19.033%	72.226%
8	900.000%	-87.072%	10.175%	55.536%	96.226%
9	1000.000%	143.350%	-70.182%	-85.695%	-85.816%

You may notice that the missing values have also been marked by the format function. This can be skipped and substituted with a different value using the "na_rep" (na replacement) parameter.

```
df.style.format("{:.3%}", na_rep="&&")
```

Output:



	A	B	C	D	E
0	100.000%	10.688%	&&	95.656%	6.841%
1	200.000%	106.851%	99.718%	-93.155%	73.043%
2	300.000%	-17.121%	-128.869%	106.144%	-4.050%
3	400.000%	-90.372%	-155.413%	&&	20.053%
4	500.000%	-74.733%	106.808%	-27.702%	8.656%
5	600.000%	-25.322%	-121.204%	27.727%	55.222%
6	700.000%	-46.774%	-142.749%	88.580%	236.063%
7	800.000%	-152.201%	-21.595%	19.033%	72.226%
8	900.000%	-87.072%	10.175%	55.536%	96.226%
9	1000.000%	143.350%	-70.182%	-85.695%	-85.816%

Create your Own Styling Method

Although you have many methods to style your dataframe, it might be the case that your requirements are different and you need a custom styling function for your analysis. You can create your function and use it with the styler object in two ways:

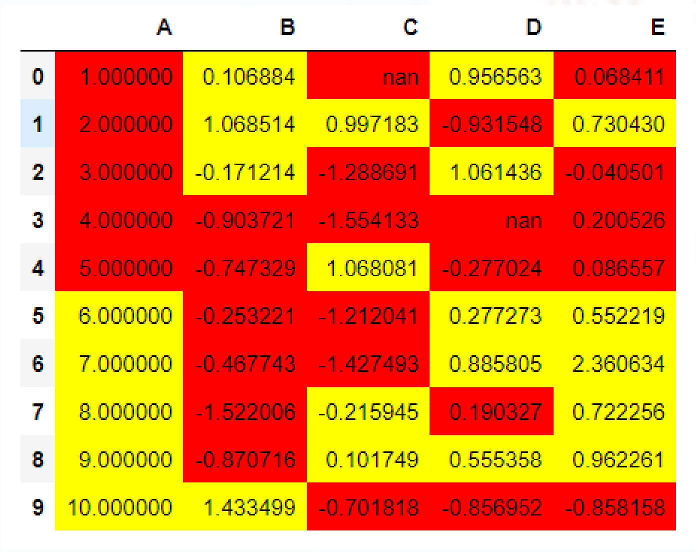
apply function: When you chain the "apply" function to the styler object, it sends out the entire row (series) or the dataframe depending upon the axis selected. Hence, if you make your function work with the "apply" function, it should return the series or dataframe with the same shape and CSS attribute-value pair.
apply map function: This function sends out scaler values (or element-wise) and therefore, your function should return a scaler only with CSS attribute-value pair.
Let's implement both types:

Target: apply function

```
def highlight_mean_greater(s):
    '''
    highlight yellow is value is greater than mean else red.
    '''
    is_max = s > s.mean()
    return ['background-color: yellow' if i else 'background-color: red' for i in is_max]
```

```
df.style.apply(highlight_mean_greater)
```

Output:



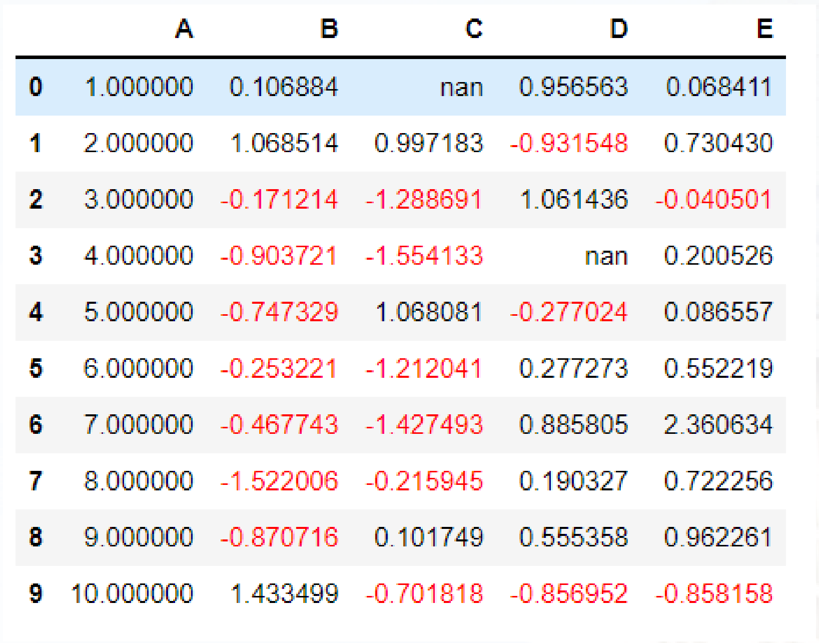
	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

Target: apply map function

```
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
```

```
df.style.apply(color_negative_red)
```

Output:



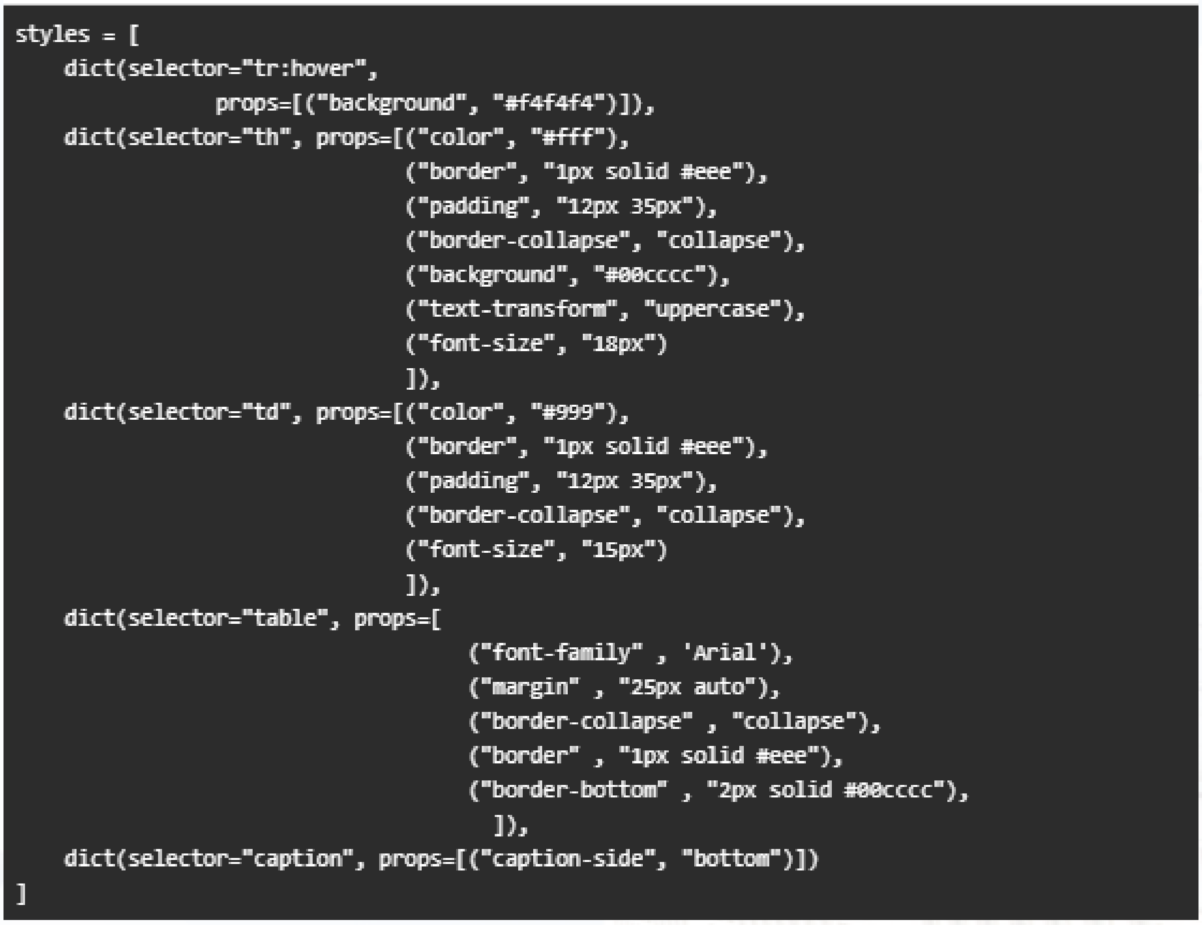
	A	B	C	D	E
0	1.000000	0.106884	nan	0.956563	0.068411
1	2.000000	1.068514	0.997183	-0.931548	0.730430
2	3.000000	-0.171214	-1.288691	1.061436	-0.040501
3	4.000000	-0.903721	-1.554133	nan	0.200526
4	5.000000	-0.747329	1.068081	-0.277024	0.086557
5	6.000000	-0.253221	-1.212041	0.277273	0.552219
6	7.000000	-0.467743	-1.427493	0.885805	2.360634
7	8.000000	-1.522006	-0.215945	0.190327	0.722256
8	9.000000	-0.870716	0.101749	0.555358	0.962261
9	10.000000	1.433499	-0.701818	-0.856952	-0.858158

Table Styles

These are styles that apply to the table as a whole, but don't look at the data. It is very similar to the set_properties function but here, in the table styles, you can customize all web elements more easily.

The function of concern here is the "set_table_styles" that takes in the list of dictionaries for defining the elements. The dictionary needs to have the selector (HTML tag or CSS class) and its corresponding props (attributes or properties of the element). The props need to be a list of tuples of properties for that selector.

The images shown in the beginning, the transformed table has the following style:



```
styles = [
    dict(selector="tr:hover",
                props=[("background", "#f4f4f4")]),
    dict(selector="th", props=[("color", "#fff"),
                               ("border", "1px solid #eee"),
                               ("padding", "12px 35px"),
                               ("border-collapse", "collapse"),
                               ("background", "#00cccc"),
                               ("text-transform", "uppercase"),
                               ("font-size", "18px")
                               ]),
    dict(selector="td", props=[("color", "#999"),
                               ("border", "1px solid #eee"),
                               ("padding", "12px 35px"),
                               ("border-collapse", "collapse"),
                               ("font-size", "15px")
                               ]),
    dict(selector="table", props=[
                                    ("font-family" , 'Arial'),
                                    ("margin" , "25px auto"),
                                    ("border-collapse" , "collapse"),
                                    ("border" , "1px solid #eee"),
                                    ("border-bottom" , "2px solid #00cccc"),
                                      ]),
    dict(selector="caption", props=[("caption-side", "bottom")])
]
```

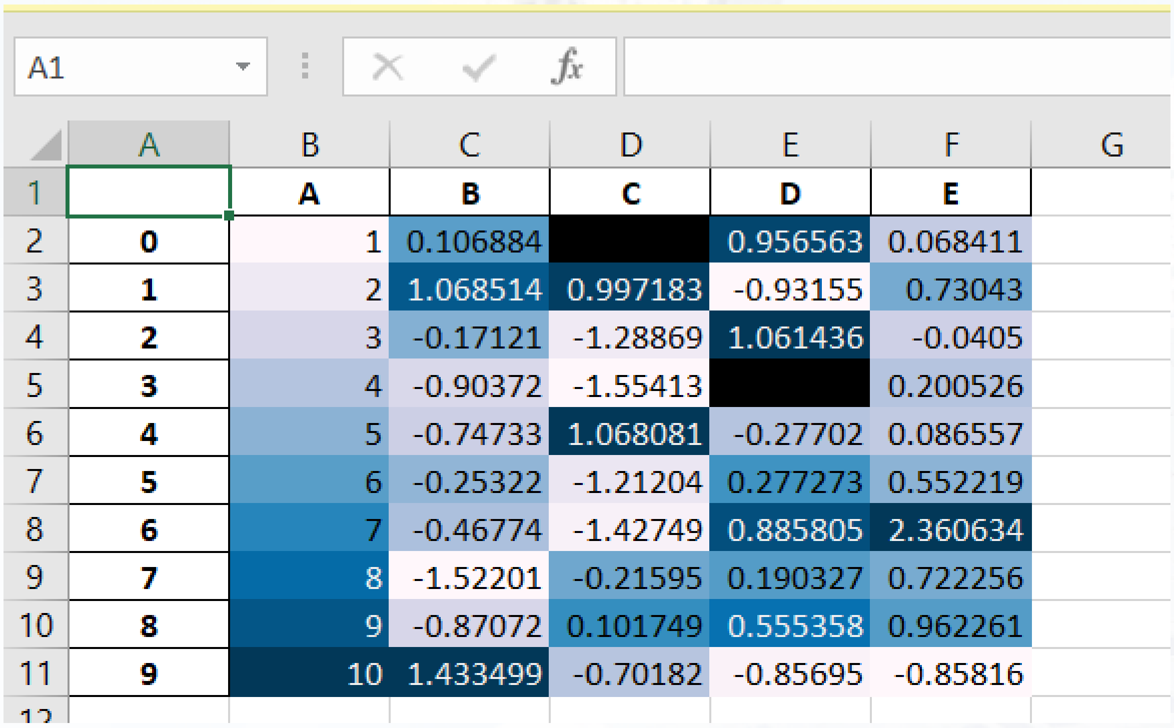
And the required methods which created the final table:

```
df.style.set_table_styles(styles).set_caption("Image by Author (Made in
Pandas)").highlight_max().highlight_null(null_color='red')
```

Export to Excel

You can store all the styling you have done on your dataframe in an excel file. The ".to_excel" function on the styler object makes it possible. The function needs two parameters: the name of the file to be saved (with extension XLSX) and the "engine" parameter should be "openpyxl".

```
df.style.set_precision(2).background_gradient().hide_index().to_exce
l('styled.xlsx', engine='openpyxl')
```



	A	B	C	D	E
0	1	0.106884		0.956563	0.068411
1	2	1.068514	0.997183	-0.93155	0.73043
2	3	-0.17121	-1.28869	1.061436	-0.0405
3	4	-0.90372	-1.55413		0.200526
4	5	-0.74733	1.068081	-0.27702	0.086557
5	6	-0.25322	-1.21204	0.277273	0.552219
6	7	-0.46774	-1.42749	0.885805	2.360634
7	8	-1.52201	-0.21595	0.190327	0.722256
8	9	-0.87072	0.101749	0.555358	0.962261
9	10	1.433499	-0.70182	-0.85695	-0.85816

Iteration

Definite iteration loops are frequently referred to as for loops because for is the keyword that is used to introduce them in nearly all programming languages, including Python.

Historically, programming languages have offered a few assorted flavors of for loop. These are briefly described in the following sections.

Numeric Range Loop

The most basic for loop is a simple numeric range statement with start and end values. The exact format varies depending on the language but typically looks something like this:

BASIC

```
for i = 1 to 10
    <loop body>
```

Here, the body of the loop is executed ten times. The variable i assumes the value 1 on the first iteration, 2 on the second, and so on. This sort of for loop is used in the languages BASIC, Algol, and Pascal.

Three-Expression Loop

Another form of for loop popularized by the C programming language contains three parts:

- An initialization
- An expression specifying an ending condition
- An action to be performed at the end of each iteration.

This type of loop has the following form:

C

```
for (i = 1; i <= 10; i++)
    <loop body>
```

This loop is interpreted as follows:

- Initialize i to 1.
- Continue looping as long as i <= 10.

- Increment i by 1 after each loop iteration.

Three-expression for loops are popular because the expressions specified for the three parts can be nearly anything, so this has quite a bit more flexibility than the simpler numeric range form shown above. These for loops are also featured in the C++, Java, PHP, and Perl languages.

Collection-Based or Iterator-Based Loop

This type of loop iterates over a collection of objects, rather than specifying numeric values or conditions:

Python

```
for i in <collection>
    <loop body>
```

Each time through the loop, the variable i takes on the value of the next object in <collection>. This type of for loop is arguably the most generalized and abstract. Perl and PHP also support this type of loop, but it is introduced by the keyword foreach instead of for.

The Python for Loop

Of the loop types listed above, Python only implements the last: collection-based iteration. At first blush, that may seem like a raw deal, but rest assured that Python's implementation of definite iteration is so versatile that you won't end up feeling cheated!

Shortly, you'll dig into the guts of Python's for loop in detail. But for now, let's start with a quick prototype and example, just to get acquainted.

Python's for loop looks like this:

Python

```
for <var> in <iterable>:
    <statement(s)>
```

<iterable> is a collection of objects—for example, a list or tuple. The <statement(s)> in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in <iterable>. The loop variable <var> takes on the value of the next element in <iterable> each time through the loop.

Here is a representative example:

```
Python

>>> a = ['foo', 'bar', 'baz']
>>> for i in a:
...     print(i)
...
foo
bar
baz
```

In this example, <iterable> is the list a, and <var> is the variable i. Each time through the loop, i takes on a successive item in a, so print() displays the values 'foo', 'bar', and 'baz', respectively. A for loop like this is the Pythonic way to process the items in an iterable.

But what exactly is an iterable? Before examining for loops further, it will be beneficial to delve more deeply into what iterables are in Python.

Iterables

In Python, iterable means an object can be used in iteration. The term is used as:

- An adjective: An object may be described as iterable.
- A noun: An object may be characterized as an iterable.

If an object is iterable, it can be passed to the built-in Python function iter(), which returns something called an iterator. Yes, the terminology gets a bit repetitive. Hang in there. It all works out in the end.

Each of the objects in the following example is an iterable and returns some type of iterator when passed to iter():

```
Python
>>> iter('foobar')                              # String
<str_iterator object at 0x036E2750>

>>> iter(['foo', 'bar', 'baz'])                 # List
<list_iterator object at 0x036E27D0>

>>> iter(('foo', 'bar', 'baz'))                 # Tuple
<tuple_iterator object at 0x036E27F0>

>>> iter({'foo', 'bar', 'baz'})                 # Set
<set_iterator object at 0x036DEA08>

>>> iter({'foo': 1, 'bar': 2, 'baz': 3})        # Dict
<dict_keyiterator object at 0x036DD990>
```

These object types, on the other hand, aren't iterable:

```
Python
>>> iter(42)                                    # Integer
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    iter(42)
TypeError: 'int' object is not iterable

>>> iter(3.1)                                   # Float
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    iter(3.1)
TypeError: 'float' object is not iterable

>>> iter(len)                                   # Built-in function
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    iter(len)
TypeError: 'builtin_function_or_method' object is not iterable
```

All the data types you have encountered so far that are collection or container types are iterable. These include the string, list, tuple, dict, set, and frozenset types.

But these are by no means the only types that you can iterate over. Many objects that are built into Python or defined in modules are designed to be iterable. For example, open files in Python are iterable. As you will see soon in the tutorial on file I/O, iterating over an open file object reads data from the file.

In fact, almost any object in Python can be made iterable. Even user-defined objects can be designed in such a way that they can be iterated over. (You will find out how that is done in the upcoming article on object-oriented programming.).

Iterators

Okay, now you know what it means for an object to be iterable, and you know how to use iter() to obtain an iterator from it. Once you've got an iterator, what can you do with it?

An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function next() is used to obtain the next value from in iterator.

Here is an example using the same list as above:

Python

```
>>> a = ['foo', 'bar', 'baz']

>>> itr = iter(a)
>>> itr
<list_iterator object at 0x031EFD10>

>>> next(itr)
'foo'
>>> next(itr)
'bar'
>>> next(itr)
'baz'
```

In this example, a is an iterable list and itr is the associated iterator, obtained with iter(). Each next(itr) call obtains the next value from itr.

Notice how an iterator retains its state internally. It knows which values have been obtained already, so when you call next(), it knows what value to return next. What happens when the iterator runs out of values? Let's make one more next() call on the iterator above:

Python

```
>>> next(itr)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(itr)
StopIteration
```

If all the values from an iterator have been returned already, a subsequent next() call raises a StopIteration exception. Any further attempts to obtain values from the iterator will fail.

You can only obtain values from an iterator in one direction. You can't go backward. There is no prev() function. But you can define two independent iterators on the same iterable object:

Python

```
>>> a
['foo', 'bar', 'baz']

>>> itr1 = iter(a)
>>> itr2 = iter(a)

>>> next(itr1)
'foo'
>>> next(itr1)
'bar'
>>> next(itr1)
'baz'

>>> next(itr2)
'foo'
```

The Guts of the Python for Loop

You now have been introduced to all the concepts you need to fully understand how Python's for loop works. Before proceeding, let's review the relevant terms:

Term	Meaning
Iteration	The process of looping through the objects or items in a collection
Iterable	An object (or the adjective used to describe an object) that can be iterated over
Iterator	The object that produces successive items or values from its associated iterable
`iter()`	The built-in function used to obtain an iterator from an iterable

Reference :

1. https://www.analyticsvidhya.com/blog/2021/06/style-your-pandas-dataframe-and-make-it-stunning/
2. https://realpython.com/python-for-loop/