Table Visualization

Using the Styler class we can be able to visualize the tabular data as per our requirements.

Styler Object and HTML

`DataFrame.style` creates a html <table> for the data frame and thereafter we can apply CSS to the table as per our requirements to manipulate many parameters including colors, fonts, borders, background, etc.

We get a HTML table representation of a DataFrame as an Output which is similar to original DataFrame. But even though we hadn't created any styles, it has already attached with some CSS classes to each cell which we can see the raw HTML as string by calling the method .to_html().

Formatting the Display

We can also use methods in styler to distinguish the display values from the actual values, and also headers of index and columns.

Formatting Values:

Using .format() method we can format the text display value of cells. This can be done for the whole table or index, or for individual columns, or multiindex levels as we want to display.
And .format_index() we can format the text display value of index labels or column headers.

Hiding Data:

We can Hide the entire index / column headers, or specific rows / columns from display using .hide() method.

For hiding the index .hide() can be used without any arguments which is useful if the index if of integer based and we dont want to display it in the table. And columns can be hidden by passing argument axis as .hide(axis="columns").

Particular rows or columns can also be hidden passing row/column labels or a slice of those labels which we like to hide to the subset argument.

Doing all these doesn't effect the integer arrangement of CSS classes.

Methods to Add Styles

Mainly there are 3 methods of adding custom CSS styles to Styler:

1. .set_table_styles() function can be used to style the entire table, columns, rows or specific HTML selectors.

2. .set_td_classes() function can be used to link external or internal CSS classes to our data cells which are created by .set_table_styles().

3. We can also add direct internal CSS to specific data cells by using .apply() and .applymap() functions.

   Both of these functions can be used for different purposes:
   .apply() - Apply a CSS-styling function column-wise, row-wise, or table-wise.
   .applymap() - Apply a CSS-styling function elementwise.

Table Styles

For controlling different individual parts of the table, including column headers and indexes table styles are very much flexible enough. we can apply all these of any feature at once to the entire table by creating a generic hover functionality.

Styles are passed in a list of dicts to .set_table_styles()

Usually, we do chaining the methods of styles so we need to explicitly instruct the method not to overwrite the existing styles.

Setting Classes and Linking to External CSS

Instead of duplicating all the CSS in python, we can use the external CSS files that controls the styling of tale and cell objects which are created earlier while designing some website.

Table Attributes:

Class can be added very easily to the main table using .set_table_attributes()

Data Cell CSS Classes:

Using .set_td_classes() method DataFrame containing strings that will be translated to CSS classes, mapped by identical column and index key values that must exist on the underlying Styler data.

Styler Functions

Acting on Data:

.applymap() (elementwise) - returns a string with a CSS attribute value pair by accepting a function which takes a single value.
.apply() (column-/row-/table-wise) - it accepts a function which takes and returns Series, DataFrame, or numpy array with same shape in which each ele is a string with a CSS attribute-value pair.

Acting on the Index and Column Headers:

.applymap_index() - Apply a CSS-styling function to the index or column headers, elementwise.
.apply_index() - Apply a CSS-styling function to the index or column headers, level-wise.

Tooltips and Captions

.set_caption() - by using this method we can add some table captions
.set_tooltips() - we can add some comments to each cell which we can able to see if we hovering on the respective cell. This can be done by passing a DataFrame (with tooltips, index and column values) as an argument.
we need to make sure that we add the previous classes because Setting new classes always overwrites.

Finer Control with Slicing

To apply styles to the specific rows or columns we use a subset argument in .apply() and .applymap() functions. It is very much similar to slicing a DataFrame
These functions combined with pd.IndexSlice which constructs a tuple can index across both dimensions with greater flexibility.
We can also put some conditions based on which filtering can be done which is conditional filtering.

Builtin Styles

There are already some built in styles exist in Styler which we can apply them directly without creating by our own and use them.

- .highlight_null: to identify missing data
- .highlight_min and .highlight_max: to identify extremities in data.
- .highlight_between to identify the values between specific ranges provided as float or Numpy or Series provided the indexes match.
- .highlight_quantile: to identify highest and lowest percentile values
- .background_gradient: highlets cells based on their values on a numeric scale.
- .text_gradient: highlets text based on their values on a numeric scale.
- .bar: display's mini bar charts within each cell backgrounds.

Set properties:

If the style donot depends on the values then by using `Styler.set_properties` returns the same properties for all the cells.

Bar charts:

We can also include some bar charts in out df.

Sharing styles

We can also export the style using `.style.export` which we had created and used for the first dataframe and import it to set using `.style.set` for the other dataframe without actually creating all those styles again and again.

Export to Excel

The final styled dataframe can be exported to excel worksheets by using OpenPyXL

Export to LaTeX

.to_latex method can be used if we want to export the styler to LaTeX.