table-compositor Documentation

Release 1.0.0

Guru Devanla

Jun 29, 2021

Contents

Contents:

CHAPTER 1

Introduction

The table-compositor library provides the API to render data stored in table-like data structures. Currently the library only supports rendering data available in a Panda's DataFrames. The DataFrame layout is used as the table layout(including single and multi hierarchical columns/indices) by the library. The table layout is rendered directly on to an xslx sheet or to a html page. Styling and layout attributes can be used to render colorful xlsx or html reports. The library also supports rendering of multiple data frames into a single xlsx sheet or html page (with horizontal/vertical layouts). The objective of the library is to be able to use the DataFrame as the API to configure the style and layout properties of the report. Callback functions are provided to customize all styling properties. The nice thing about the callback functions are that the style properties are set on cells indexed with index/column values available in the original dataframe used during rendering.

Code: https://github.com/InvestmentSystems/table-compositor

Docs: http://table-compositor.readthedocs.io

Packages: https://pypi.python.org/pypi/table-compositor

1.1 Getting Started

The table-compositor library builds on the concept of Panda's DataFrame as API to render colorful reports. The various ways of providing styling attributes and choosing layouts are demonstrated with numerous examples in the documentation. Please refer to the Bacis section of the documentation to get started with the HelloWorld example.

1.2 Installation

A standard setuptools installer is available via PyPI:

https://pypi.python.org/pypi/table-compositor

Or, install via pip3:

```
pip3 install table-compositor
```

Source code can be obtained here:

https://github.com/InvestmentSystems/table-compositor

CHAPTER 2

Basics

The purpose of this library is to use the Pandas DataFrame as an interface to represent the layout of a table that needs to be rendered to an xlsx file or as an html table. The library abstracts away the tedious work of working at the *cell* level of an xlsx sheet or a html table. It provides a call-back mechanism by which the user is able to provide values that need to be rendered and also the styling that needs to be used for each cell in the rendered table. The library is also capable of laying out multiple tables in the same sheet which are evenly spaced vertically or horizontally based on the layout configuration provided.

2.1 Sample Data

During the later part of this documentation, we will use the sample data from the Social Security Administration which contains the U.S. child birth name records. We choose this sample data for two reasons. We reuse some of the discussion that are outlined by Wes McKinnery's Python For Data Analysis, 2nd Edition(2017). The same data is also used in the documentation of another library *function-pipe <http://function-pipe.readthedocs.io/en/latest/index.html>* that the Investment Systems Group has open-sourced.

https://www.ssa.gov/oact/babynames/names.zip

Further more, we will assume that a flattened file from all the smaller files in the .zip file is available after we invoke the following function.

Please refer to the XLSX Examples section for code that loads this data.

2.2 A Hello World Example: Dataframe to Xlsx

Every use of this library involves four steps.

1. We build a dataframe that resembles the shape of the table that will be rendered.
2. The dataframe is passed as an argument to the function called `build_presentation_model`. This function accepts a *dataframe* and also a number of functions as arguments. We call the value returned by this function, the *presentation_model*.

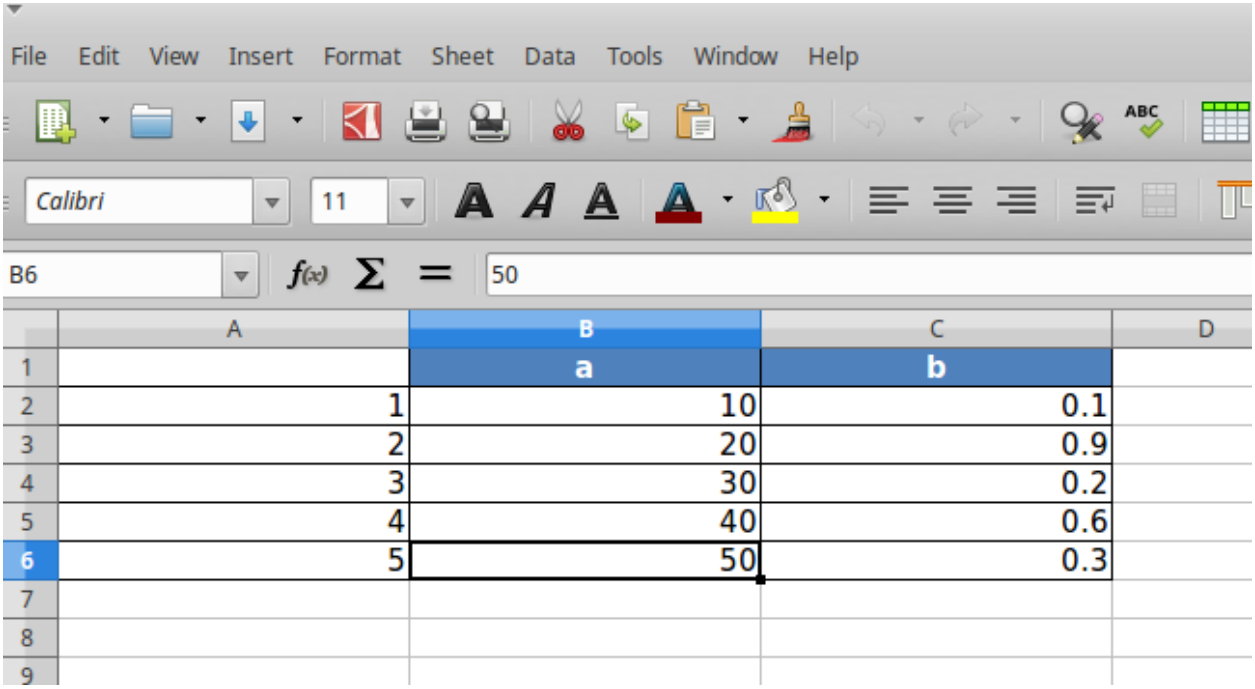
3. Create a *layout* of multiple *presentation models* (if we want more than one table rendered in same xlsx sheet or same html page)
4. Call the *render_xlsx* or *render_html* functions on the respective writers. For xlsx files either OpenPyxlCompositor(uses *openpyxl* library) or XlsxWriterCompositor(uses *xlsxwriter* library). For HTML use the *HTMLWriter*.

2.2.1 A Quick Look at a Xlsx example

We will start with a simple dataframe and render the dataframe as-is to a xlsx file

```
import pandas as pd
from table_compositor.table_compositor import build_presentation_model
from table_compositior.xlsx_writer import OpenPyxlCompositor
# Note: use XlsxWriterCompositor to use xlsxwriter library

sample_df = pd.DataFrame(dict(a=[10, 20, 30, 40, 50], b=[0.1, 0.9,0.2, 0.6,0.3]),␣
↪index=[1,2,3,4,5])

# create a presentation model
# defaults to engine='openpyxl'. Needs to be set to 'xlsxwriter' to use `xlsxwriter`␣
↪library instead.
presentation_model = build_presentation_model(df=sample_df)

# create a layout, which is usually a nested list of presentation models
layout = [presentation_model]

# render to xlsx
output_fp = '/tmp/example1.xlsx'
OpenPyxlCompositor.to_xlsx(layout, output_fp=output_fp)
```

Running this code produces the following output:



	A	B	C	D
1		a	b	
2	1	10	0.1	
3	2	20	0.9	
4	3	30	0.2	
5	4	40	0.6	
6	5	50	0.3	
7				
8				
9				

In the above code snippet, we first created a dataframe called `sample_df`.

To render this *dataframe*, we first invoke *build_presentation_model*. The *build_presentation_model* accepts the *dataframe* as its first argument. In this example, we use the *defaults* provided by this method for all other arguments. The *build_presentation_model* returns an *presentation_model* object.

Before we call *OpenPyxlCompositor.to_xlsx* we create a *layout*. A *layout* is a nested list of *presentation_models*. In our case, since we have only one *presentation_model* we create a list with a single element. Later on when we work with multiple presentation models that need to be rendered on to the same sheet, we could create nested list such as *[[model1, model2], [model3]]* etc.

2.3 Building the Presentation Model

The *build_presentation_model* function is the most important interface in this library. This function exposes all the functionality that is required to render beautiful looking excel worksheets or html tables.

We will now build up on our previous example and add styling to the report we generate. Before, we do that lets take a quick look at the signature of *build_presentation__model*.

`table_compositor.table_compositor.`**`build_presentation_model`**(*, *df*, *output_format='xlsx'*, *data_value_func=None*, *column_style_func=None*, *data_style_func=None*, *header_style_func=None*, *header_value_func=None*, *index_style_func=None*, *index_value_func=None*, *index_name_func=None*, *index_name_style_func=None*, *engine='openpyxl'*, ***kwargs*)

Construct and return the presentation model that will be used while rendering to html/xlsx formats. The returned object has all the information required to render the tables in the requested format. The details of the object is transparent to the caller. It is only exposed for certain advanced operations.

Parameters

- **`df`** – The dataframe representation of the table. The shape of the dataframe closely resembles the table that will be rendered in the requested format.
- **`output_format`** – 'html' or 'xlsx'
- **`data_value_func`** – example: lambda idx, col: df.loc[idx, col], assuming df is in the closure. This can be None, if no data transformation is required to the values already present in the source df
- **`column_style_func`** – the function can substitute the data_style_func, if the same style can be applied for the whole column. This argument should be prefered over the *data_style_func* argument. Using this option provides better performance since the fewer objects will be created internally and fewer callbacks are made to this function when compared to data_style_func.This argument only applies to the data contained in the dataframe and not the cell where the headers are rendered. For fine grained control at *cell* level, the

data_style_func argument can be used. For more information on return values of this function, refer to the documentation for *data_style_func* argument.

- **data_style_func** – used to provide style at the cell level. Example: lambda idx, col: return dict(font=Font(...)), where Font is the openpyxl object and *font* is the attr available in the *cell* instance of openpyxl. For xlsx, the keys in the dict are the attrs of the *cell* object in openpyxl and the values correspond to the value of that attribute. Example are found in xlsx_styles module. For html, the key-value pairs are any values that go into to the style attribute of a td, th cell in html. Examples are found in html_styles module. example: dict(background-color='#F8F8F8'). When performance becomes an issue, and cell level control is not needed, it is recommended to use the *column_style_func* argument rathat than this argument. If the prefered engine is XlswWriter, then the style dictionary returned should have key/values compatible with the *Format* object declarted in the *XlsxWriter* library. A reference can be found in ''xlsx_styles.XlsxWriterStyleHelper' class
- **header_value_func** – func that takes a object of type *IndexNode*. The *IndexNode* contains the attributes that refer to the header being rendered. The returned value from this function is displayed in place of the header in the dataframe at the location. The two properties available on the *IndexNode* object are *value* and *key*. The *key* is useful to identify the exact index and level in context while working with multi-hierarchical columns.
- **header_style_func** – func that takes a object of type *IndexNode*. The return value of this function is similar to data_style_func.
- **index_value_func** – func that takes a object of type *IndexNode*. The *IndexNode* contains the attributes that refer to the index being rendered. The returned value from this function is displayed in place of the index in the dataframe at the location.
- **index_style_func** – func that takes a object of type *IndexNode*. The return value of this function is similar to data_style_func.
- **index_name_func** – func that returns a string for index name (value to be displayed on top-left corner, above the index column)
- **index_name_style** – the style value same as data_style_func that will be used to style the cell
- **engine** – required while building presentation model for xlsx. Argument ignored for HTML rendering. This argument is used to provide the default callback style functions, where the style dictionary returned by the callback functions should be compatible with the engine being used.
- **kwargs** – 'hide_index' - if True, then hide the index column, default=False

  'hide_header, - if True, then hide the header, default=False

  'use_convert' - if True, do some conversions from dataframe values to values excel can understand for example np.NaN are converted to NaN strings

Returns A presentation model, to be used to create layout and provide the layout to the html or xlsx writers.

About the callback functions provided as arguments:

Note that callback function provided as arguments to this function are provided with either a tuple of index, col arguments are some information regarding the index or headers being rendered. Therefore, a common pattern would be to capture the *dataframe* being rendered in a closure of this callback func before passing them as arugments.

For example:

df = pd.DataFrame(dict(a=[1, 2, 3]))

def data_value_func():

def _inner(idx, col): return df.loc[idx, col] * 10.3

return _inner

pm = build_presentation_model(df=df, data_value_func=data_value_func())

2.4 Improving on our first iteration

Now, that we got a overview of the *build_presentation_mode* function, lets try setting these arguments to improve the look of our reports.

Say, we have the following requirements:

1. Display column 'A' as in dollar format.
2. Display column 'B' as percentage values.'
3. Set back-ground color of column 'B' to red if value is less than 50%
4. Capitalize all the column headers and add a yellow background
5. Multiply all index values by 100 while rendering and add a color to the background.
6. Display a 'custom text' on the top left corner, where pandas whole usually display the index name if available.

We update our previous example to do the following:

```
import os
import tempfile
import pandas as pd
from table_compositor.table_compositor import build_presentation_model

# There are equivalent classes for using xlsxwriter library. Namely,
# XlsxWriterCompositor and XlsxWriterStyleHelper
from table_compositor.xlsx_writer import OpenPyxlCompositor
from table_compositor.xlsx_styles import OpenPyxlStyleHelper

```

```
def basic_example2():

    df = pd.DataFrame(dict(a=[10, 20, 30, 40, 50], b=[0.1, 0.9,0.2, 0.6,0.3]),␣
↪index=[1,2,3,4,5])

    def style_func(idx, col):
        if col == 'b':
            return OpenPyxlStyleHelper.get_style(number_format='0.00%')
        else:
            # for 'a' we do dollar format
            return OpenPyxlStyleHelper.get_style(number_format='$#,##.00')

    # create a presentation model
    # note the OpenPyxlStyleHelper function available in xlsx_styles module. But a␣
↪return value of style function
    # can be any dict whose keys are attributes of the OpenPyxl cell object.
    presentation_model = build_presentation_model(
        df=df,
        data_value_func=lambda idx, col: df.loc[idx, col] * 10 if col == 'a' else df.
↪loc[idx, col],
```

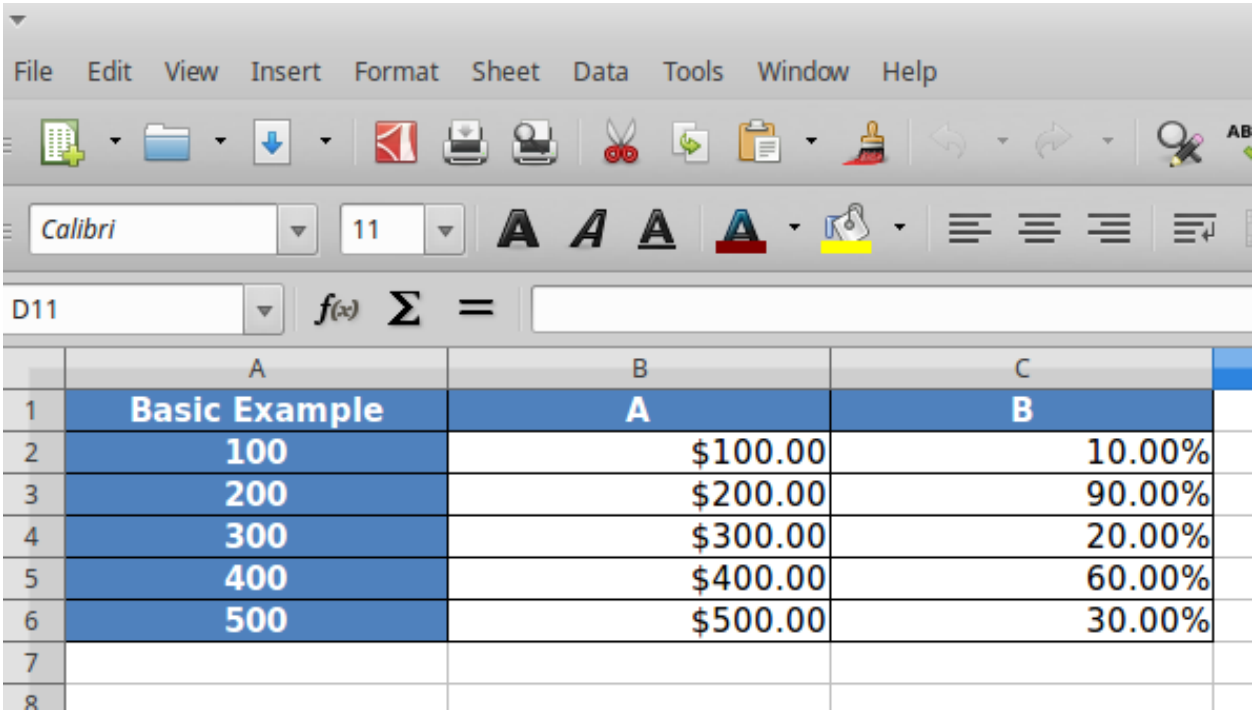
(continues on next page)

(continued from previous page)

```
        data_style_func=style_func,
        header_value_func=lambda node: node.value.capitalize(),
        header_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
        index_value_func=lambda node: node.value * 100,
        index_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
        index_name_func=lambda _: 'Basic Example',
        index_name_style_func=lambda _: OpenPyxlStyleHelper.default_header_style())

    # create a layout, which is usually a nested list of presentation models
    layout = [presentation_model]

    # render to xlsx
    output_fp = os.path.join(tempfile.gettempdir(), 'basic_example2.xlsx')
    OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp)

```

On line 3 we create the dataframe.

To satisfy the requirements we listed above we pass the callback function to the *build_presentation_model*. Note that some helper functions are available in xlsx_style function to create styles for openpyxl. But, any other dict with keys that are *attr* of cell object of openpyxl should work. The above example produces the output as shown below:



	A	B	C
1	Basic Example	A	B
2	100	$100.00	10.00%
3	200	$200.00	90.00%
4	300	$300.00	20.00%
5	400	$400.00	60.00%
6	500	$500.00	30.00%
7			
8			

2.5 Multi-hierarchical columns and indices

Rendering dataframes with multi-hierarchical columns or indices are very similar to rendering the simpler dataframes. The *data_value_func* and *data_style_func* work the same way. The functions that handle *index* cell rendering and *column* header rendering can access the *IndexNode* object that is passed to those functions to determine the value and level that is currently being rendered. This becomes clearer with an example.

We demonstrate this by setting a variety of colors to each cell that holds one of the values of the hierarchical columns

or indices.

Note that the *IndexNode* argument passed to the callback function has a *node.key* field that unique identifies each cell with a name that is built appending the value of each item in the index or column hierarchy.

```
import os
import tempfile
import pandas as pd
from table_compositor.table_compositor import build_presentation_model

# There are equivalent classes for using xlsxwriter library. Namely,
# XlsxWriterCompositor and XlsxWriterStyleHelper
from table_compositor.xlsx_writer import OpenPyxlCompositor
from table_compositor.xlsx_styles import OpenPyxlStyleHelper

```

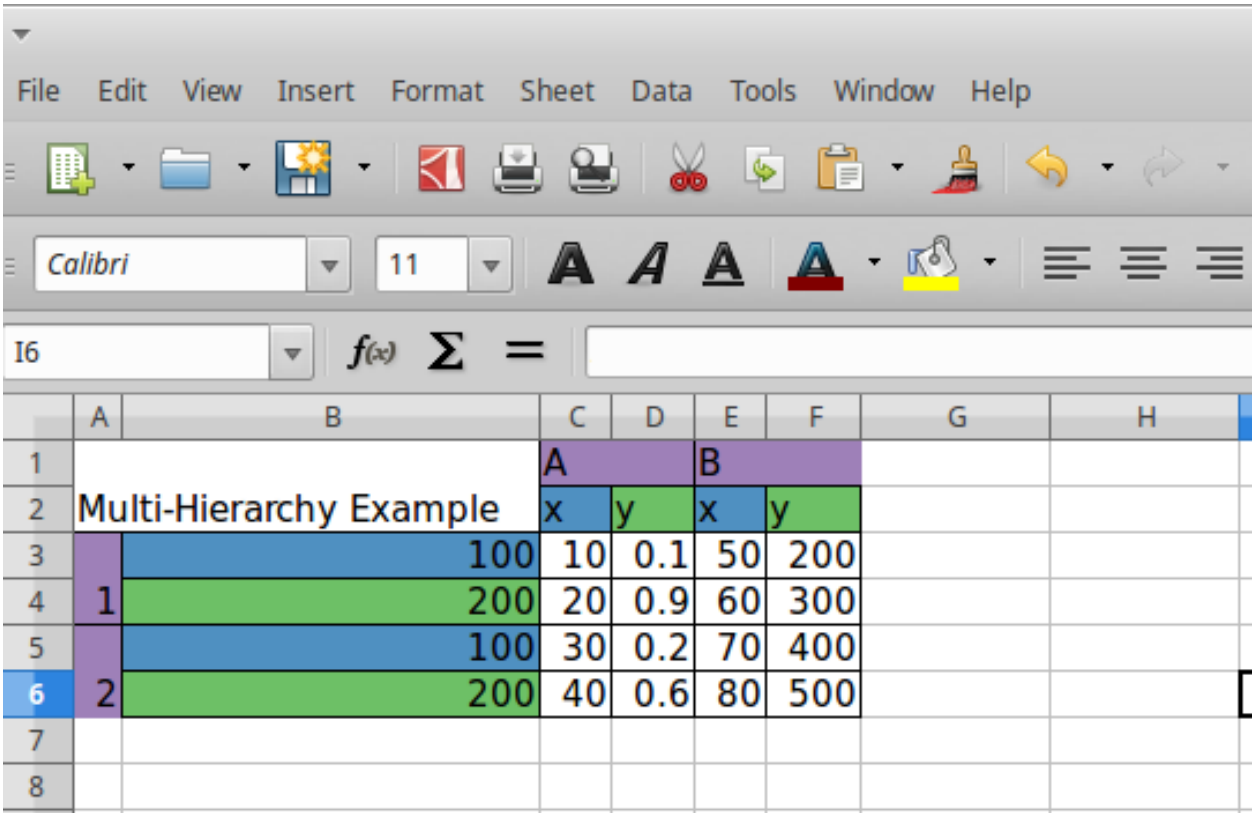
```
def basic_example3():

    df = pd.DataFrame(dict(a=[10, 20, 30, 40],
                           b=[0.1, 0.9,0.2, 0.6],
                           d=[50, 60, 70, 80],
                           e=[200, 300, 400, 500]))
    df.columns = pd.MultiIndex.from_tuples([('A', 'x'), ('A', 'y'), ('B', 'x'), ('B',
↪'y')])
    df.index = pd.MultiIndex.from_tuples([(1, 100), (1, 200), (2, 100), (2, 200)])
    print(df)

    def index_style_func(node):
        # node.key here could be one of (1,), (1, 100), (2,), (2, 100), (2, 200)
        bg_color = 'FFFFFF'
        if node.key == (1,) or node.key == (2,):
            bg_color = '9E80B8'
        elif node.key[1] == 100:
            bg_color = '4F90C1'
        elif node.key[1] == 200:
            bg_color = '6DC066'
        return OpenPyxlStyleHelper.get_style(bg_color=bg_color)

    def header_style_func(node):
        bg_color = 'FFFFFF'
        if node.key == ('A',) or node.key == ('B',):
            bg_color = '9E80B8'
        elif node.key[1] == 'x':
            bg_color = '4F90C1'
        elif node.key[1] == 'y':
            bg_color = '6DC066'
        return OpenPyxlStyleHelper.get_style(bg_color=bg_color)

    # create a presentation model
    # note the OpenPyxlStyleHeloer function available in xlsx_styles module. But a␣
↪return value of style function
    # can be any dict whose keys are attributes of the OpenPyxl cell object.
    presentation_model = build_presentation_model(
        df=df,
        index_style_func=index_style_func,
        header_style_func=header_style_func,
        index_name_func=lambda _: 'Multi-Hierarchy Example')

```

(continues on next page)

(continued from previous page)

```
    # create a layout, which is usually a nested list of presentation models
    layout = [presentation_model]

    # render to xlsx
    output_fp = os.path.join(tempfile.gettempdir(), 'basic_example3.xlsx')
    OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp)

```

The above function gives us the *xlsx* file shown below. Note the colors used to render the indices and columns and review how the two functions, namely, *index_style_function* and *header_style_function* provide the colors based on the *IndexNode* attributes. You will notice the use of *node.key* in these functions to identify each cell uniquely.



CHAPTER 3

Layouts

Apart from providing styling and formatting facilities, the library also provides a powerful way to layout multiple tables on one sheet. In this section we will look at some examples.

We will use the same presentation model from *basic_example2()*. We will layout the presentation models with different layouts.

```
import os
import tempfile
import pandas as pd
from table_compositor.table_compositor import build_presentation_model

# There are equivalent classes for using xlsxwriter library. Namely,
# XlsxWriterCompositor and XlsxWriterStyleHelper
from table_compositor.xlsx_writer import OpenPyxlCompositor
from table_compositor.xlsx_styles import OpenPyxlStyleHelper

```

```
def layout_example1():

    df = pd.DataFrame(dict(a=[10, 20, 30, 40, 50], b=[0.1, 0.9,0.2, 0.6,0.3]),␣
↪index=[1,2,3,4,5])

    def style_func(idx, col):
        if col == 'b':
            return OpenPyxlStyleHelper.get_style(number_format='0.00%')
        else:
            # for 'a' we do dollar format
            return OpenPyxlStyleHelper.get_style(number_format='$#,##.00')

    # create a presentation model
    # note the OpenPyxlStyleHeloer function available in xlsx_styles module. But a␣
↪return value of style function
    # can be any dict whose keys are attributes of the OpenPyxl cell object.
    presentation_model = build_presentation_model(
```

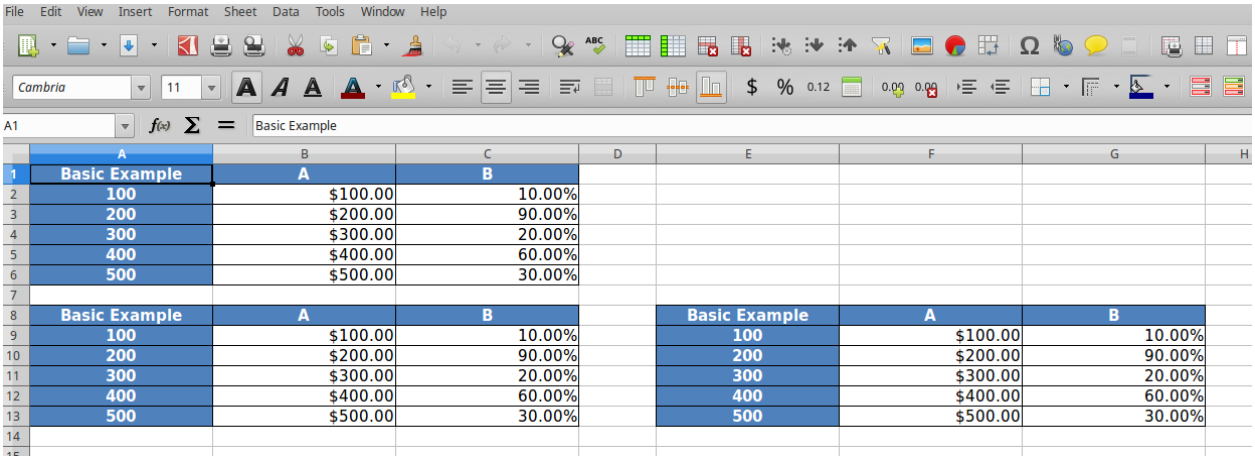
(continues on next page)

(continued from previous page)

```
16          df=df,
17          data_value_func=lambda idx, col: df.loc[idx, col] * 10 if col == 'a' else df.
    ↪loc[idx, col],
18          data_style_func=style_func,
19          header_value_func=lambda node: node.value.capitalize(),
20          header_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
21          index_value_func=lambda node: node.value * 100,
22          index_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
23          index_name_func=lambda _: 'Basic Example',
24          index_name_style_func=lambda _: OpenPyxlStyleHelper.default_header_style())
25
26      # start_layout_code_1
27      # create a layout, which is usually a nested list of presentation models
28      layout = [[presentation_model], [[presentation_model], [presentation_model]]]
29
30      # render to xlsx
31      output_fp = os.path.join(tempfile.gettempdir(), 'layout_vertical_example1.xlsx')
32      # the default value for orientation is 'vertical'
33      OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp, orientation=
    ↪'vertical')
34
35      output_fp = os.path.join(tempfile.gettempdir(), 'layout_horizontal_example1.xlsx')
36      OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp, orientation=
    ↪'horizontal')
37      print('Writing xlsx file=', output_fp)
38
39      # mutiple nesting
40      layout_complex = [presentation_model,
41                        [presentation_model, [presentation_model, presentation_model]]]
42
43      output_fp = os.path.join(tempfile.gettempdir(), 'layout_complex_example1.xlsx')
44      OpenPyxlCompositor.to_xlsx(layout=layout_complex, output_fp=output_fp,␣
    ↪orientation='vertical')
45      print('Writing xlsx file=', output_fp)
46      # end_layout_code_1
47
```

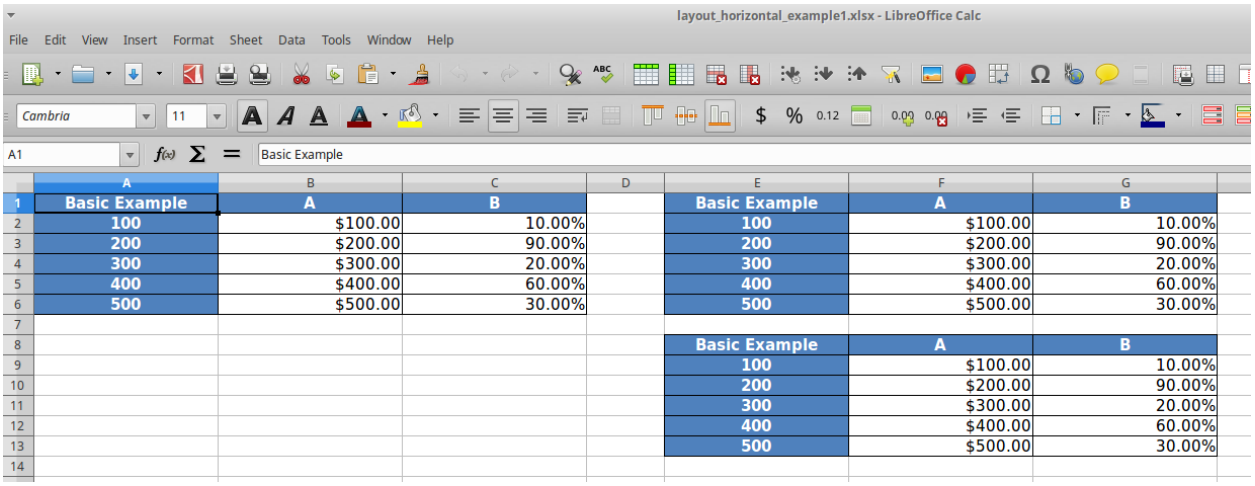
In the preceding example, we create two layouts. On line 28, we have a layout defined and then rendered to two files with different *orientations*.

When the orientation is *vertical*, then each item (presentation_model) in the list is layed out vertically. The orientation flips between *vertical* and *horizontal* for every nested listed that is encountered. In this example, you will notice that since the second item in the outer list is a list, the two presentation models in the inner list are rendered side-by-side (i.e. with horizontal orientation)



	A	B	C	D	E	F	G	H
1	Basic Example	A	B					
2	100	$100.00	10.00%					
3	200	$200.00	90.00%					
4	300	$300.00	20.00%					
5	400	$400.00	60.00%					
6	500	$500.00	30.00%					
7								
8	Basic Example	A	B		Basic Example	A	B	
9	100	$100.00	10.00%		100	$100.00	10.00%	
10	200	$200.00	90.00%		200	$200.00	90.00%	
11	300	$300.00	20.00%		300	$300.00	20.00%	
12	400	$400.00	60.00%		400	$400.00	60.00%	
13	500	$500.00	30.00%		500	$500.00	30.00%	
14								

When the value of orientation argument is changed to *horizontal*, the renderer renders the outerlist horizontally and flips the orientation of inner lists to vertical. The second output is show below.



	A	B	C	D	E	F	G
1	Basic Example	A	B		Basic Example	A	B
2	100	$100.00	10.00%		100	$100.00	10.00%
3	200	$200.00	90.00%		200	$200.00	90.00%
4	300	$300.00	20.00%		300	$300.00	20.00%
5	400	$400.00	60.00%		400	$400.00	60.00%
6	500	$500.00	30.00%		500	$500.00	30.00%
7							
8					Basic Example	A	B
9					100	$100.00	10.00%
10					200	$200.00	90.00%
11					300	$300.00	20.00%
12					400	$400.00	60.00%
13					500	$500.00	30.00%
14							

CHAPTER 4

Example of XLSX Styles

In the preceding examples, we have used the functions provided by *xslx_styles.OpenPyxlStyleHelper* to return the required style dictionary. Some examples of style dictionaries that can be returned by functions returning styles are provided below for reference. For details on how to build style attributes refer to the *openpyxl* documentation.

4.1 Style with background color

```
from openpyxl.styles import PatternFill
from openpyxl.styles import fills

fill = PatternFill(fgColor=Color('4f81BD'), patternType=fills.FILL_SOLID)

# note that we return a dict, whose key = `fill` which is an
# attrbute of  `cell` object in `openpyxl`
style = dict(fill=fill)
```

4.2 Style with percentage formatting

```
number_format = '0.00%'

fill = PatternFill(fgColor=Color('4f81BD'), patternType=fills.FILL_SOLID)

# note that we return a dict, whose key = `number_format` which is an
# attrbute of  `cell` object in `openpyxl`
style = dict(number_format=number_format)
```

4.3 Style with alignment and fonts

```
from openpyxl.styles import Alignment
from openpyxl.styles import Font
font=Font(bold=True, color='FFFFFF')

# note that we return a dict, whose key = `number_format` which is an
# attrbute of  `cell` object in `openpyxl`
style = dict(alignment=Alignment(horizontal=center, font=font)
```

4.4 Using a different XLXS Writer Engine

Note that if *xlsxwriter* library is uses, the keys in the dictionary returned by the callback funcs should match the keys required to build the *Format* object declared in the *xlsxwriter* library. Some examples of these keys can be found in *xlsx_styles.XlsxWriterStyleHelper* class.

CHAPTER 5

Example of HTML Styles

In the Basic section, we only saw examples of how to render dataframes to a xlsx format. The same setup can be used to render dataframes to HTML using the *html_writer.HTMLWriter.to_html* function. The only thing that has to be changed is the style attributes that are being returned. We want our style providing functions to return style attributes that can be inlined into the *style* attribute of a <*td*> or <*th*> tag.

Some examples of style dictionaries that can be return by functions returning styles are provided below for reference.

5.1 Style for headers

```
style = dict(
 text_align='center',
 background_color='#4F81BD',
 color='#FFFFFF',
 font_weight='bold',
 white_space='pre',
 padding='10px',
border=1)
```

5.2 Style for cell holding numeric values

```
numeric_style = dict(
             text_align='right',
             background_color='#FFFFFF',
             color='#000000',
             font_weight='normal',
             white_space='pre',
             padding='10px',
             border=None)
```

5.3 Style using the html_styles.td_style object

```
style = td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)
```

CHAPTER 6

XLSX Examples

This page provides a list of examples that demonstrate rendering xlsx output from the given dataframe. Each example is self-contained inside a class. We have some helper functions to provide the data we need for this examples

All examples listed in this section use *OpenPyxlStyleHelper* and *OpenPyxlCompositor*. To use the *xlsxwriter* library replace these with *XlsxWriterStyleHelper* and *XlsxWriterCompositor* respectively.

If the callback funcs do not use the **StyleHelpers* and return their own *Style* objects then the objects returned should be compatible with the value *engine* provided to the engine attribute.

Examples of relevant style objects can be found respective documentations for the *engine* being used.

6.1 Helper Functions (Data loading routines)

```
import tempfile
import zipfile
import collections
import os
import webbrowser

import requests
import pandas as pd

import table_compositor.table_compositor as tc
import table_compositor.xlsx_writer as xlsxw
import table_compositor.xlsx_styles as xlsstyle
```

```
# code snippet adapted from http://function-pipe.readthedocs.io/en/latest/usage_df.
↪html
# source url
URL_NAMES = 'https://www.ssa.gov/oact/babynames/names.zip'
ZIP_NAME = 'names.zip'

```

(continues on next page)

(continued from previous page)

```
def load_names_data():
    fp = os.path.join(tempfile.gettempdir(), ZIP_NAME)
    if not os.path.exists(fp):
        r = requests.get(URL_NAMES)
        with open(fp, 'wb') as f:
            f.write(r.content)

    post = collections.OrderedDict()
    with zipfile.ZipFile(fp) as zf:
        # get ZipInfo instances
        for zi in sorted(zf.infolist(), key=lambda zi: zi.filename):
            fn = zi.filename
            if fn.startswith('yob'):
                year = int(fn[3:7])
                df = pd.read_csv(
                    zf.open(zi),
                    header=None,
                    names=('name', 'gender', 'count'))
                df['year'] = year
                post[year] = df

        df = pd.concat(post.values())
        df.set_index('name', inplace=True, drop=True)
        return df

def sample_names_data():
    df = load_names_data()
    df = df[(df['year'] == 2015) & (df['count'] > 1000)]
    return df.sample(100, random_state=0).sort_values('count')

def top_names_for_year(year=2015, gender='F', top_n=5):
    df = load_names_data()
    df = df[(df['year'] == year) & (df['gender'] == gender)]
    df = df.sort_values('count')[:top_n]
    return df
```

6.2 Example 1 - DataFrame with default styles

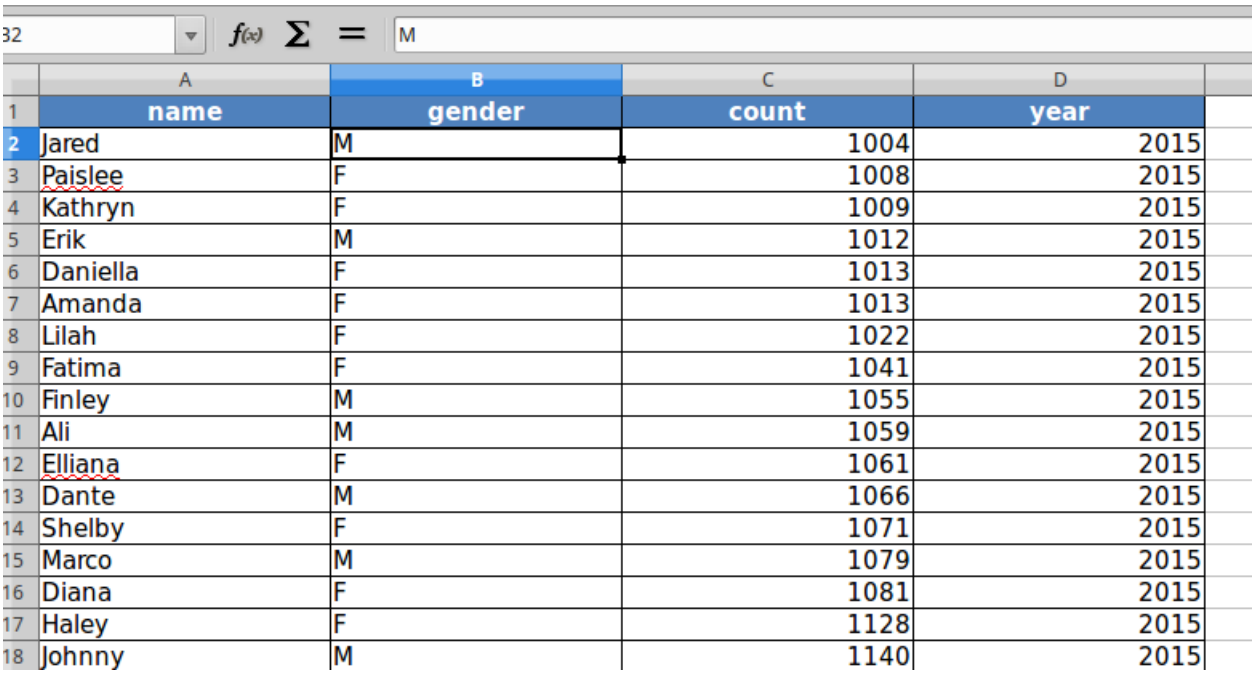
Demonstrates converting dataframe into html format with default styles.

```
class XLSXExample1:
    '''
    Demonstrates rendering a simple dataframe to a xlsx file
    using the default styles
    '''
    @classmethod
    def render_xlsx(cls):
        '''
        Render the df to a xlsx file.
        '''

        # load data
        df = sample_names_data()
        # build presentation model
```

(continues on next page)

(continued from previous page)

```
        pm = tc.build_presentation_model(df=df, output_format='xlsx')

        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example1.xlsx')
        layout = [pm]
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp)
```



	A	B	C	D
1	name	gender	count	year
2	Jared	M	1004	2015
3	Paislee	F	1008	2015
4	Kathryn	F	1009	2015
5	Erik	M	1012	2015
6	Daniella	F	1013	2015
7	Amanda	F	1013	2015
8	Lilah	F	1022	2015
9	Fatima	F	1041	2015
10	Finley	M	1055	2015
11	Ali	M	1059	2015
12	Elliana	F	1061	2015
13	Dante	M	1066	2015
14	Shelby	F	1071	2015
15	Marco	M	1079	2015
16	Diana	F	1081	2015
17	Haley	F	1128	2015
18	Johnny	M	1140	2015

6.3 Example 2 - DataFrame with custom styles

In this example, we format the different components of dataframe with various styling attributes

```
class XLSXExample2:
    '''
    Demonstrates using call-backs that help set the display and style
    properties of each cell in the xlsx sheet.
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
```

(continues on next page)

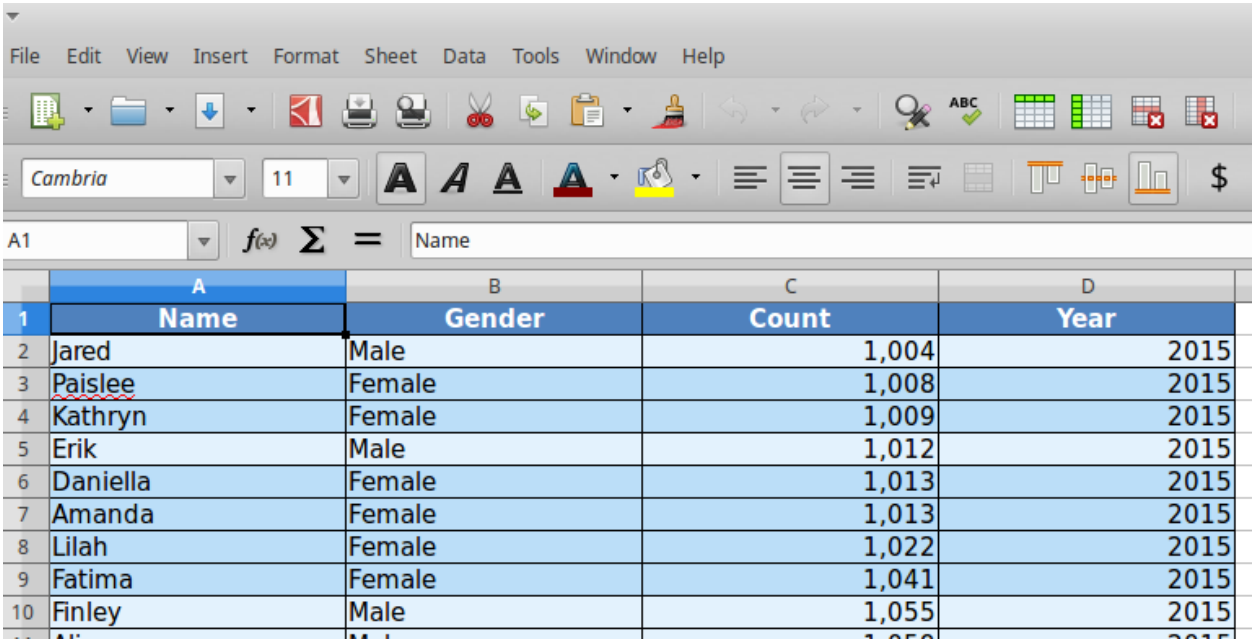
(continued from previous page)

```
        def _inner(idx, col):
            bg_color = None
            number_format='General'
            if col == 'count':
                number_format='#,##0'
            if df.loc[idx, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(
                bg_color=bg_color,
                number_format=number_format)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return value.capitalize()

    @staticmethod
    def index_name_style_func(value):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def index_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def index_style_func(df):
        def _inner(node):
            bg_color = None
            if df.loc[node.value, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(bg_color=bg_color)
        return _inner

    @classmethod
    def render_xlsx(cls):
        # load data
        df = sample_names_data()
        # build presentation model
        klass_ = XLSXExample2
        pm = tc.build_presentation_model(
            df=df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
```

(continues on next page)

(continued from previous page)

```
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example2.xlsx')
        layout = [pm]
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp)
```



	A	B	C	D
1	Name	Gender	Count	Year
2	Jared	Male	1,004	2015
3	Paislee	Female	1,008	2015
4	Kathryn	Female	1,009	2015
5	Erik	Male	1,012	2015
6	Daniella	Female	1,013	2015
7	Amanda	Female	1,013	2015
8	Lilah	Female	1,022	2015
9	Fatima	Female	1,041	2015
10	Finley	Male	1,055	2015

6.4 Example 3 - Simple DataFrame with Layouts

Demonstrates rendering multi-dataframes in one worksheet along with common functions for styling

```
class XLSXExample3:
    '''
    Demonstrates using call-backs and also rendering multiple tables to single
    worksheet.
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner
```

(continues on next page)

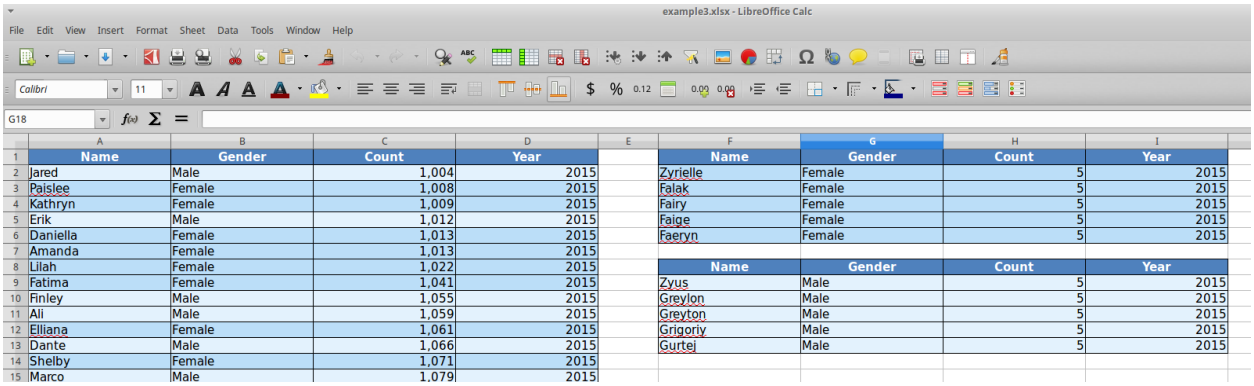
(continued from previous page)

```
    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            bg_color = None
            number_format='General'
            if col == 'count':
                number_format='#,##0'
            if df.loc[idx, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(
                bg_color=bg_color,
                number_format=number_format)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return value.capitalize()

    @staticmethod
    def index_name_style_func(value):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def index_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def index_style_func(df):
        def _inner(node):
            bg_color = None
            if df.loc[node.value, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(bg_color=bg_color)
        return _inner


    @classmethod
    def render_xlsx(cls):
        # Prepare first data frame (same as in render_xlsx)
        df = sample_names_data()
        # build presentation model
        klass_ = XLSXExample3
        pm_all = tc.build_presentation_model(
            df=df,
```

(continues on next page)

(continued from previous page)

```
            output_format='xlsx',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        male_df = top_names_for_year(gender='M')
        pm_top_male = tc.build_presentation_model(
            df=male_df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(male_df),
            data_style_func=klass_.data_style_func(male_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(male_df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        female_df = top_names_for_year(gender='F')
        pm_top_female = tc.build_presentation_model(
            df=female_df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(female_df),
            data_style_func=klass_.data_style_func(female_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(female_df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)


        layout = [pm_all, [pm_top_female, pm_top_male]]
        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example3.xlsx')
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp, orientation=
↪'horizontal')
```



	A	B	C	D	E	F	G	H	I
1	Name	Gender	Count	Year		Name	Gender	Count	Year
2	Jared	Male	1,004	2015		Zyrielle	Female	5	2015
3	Paislee	Female	1,008	2015		Falak	Female	5	2015
4	Kathryn	Female	1,009	2015		Fairy	Female	5	2015
5	Erik	Male	1,012	2015		Faige	Female	5	2015
6	Daniella	Female	1,013	2015		Faeryn	Female	5	2015
7	Amanda	Female	1,013	2015					
8	Lilah	Female	1,022	2015		Name	Gender	Count	Year
9	Fatima	Female	1,041	2015		Zyus	Male	5	2015
10	Finley	Male	1,055	2015		Greylon	Male	5	2015
11	Ali	Male	1,059	2015		Greyton	Male	5	2015
12	Elliana	Female	1,061	2015		Grigoriy	Male	5	2015
13	Dante	Male	1,066	2015		Gurtej	Male	5	2015
14	Shelby	Female	1,071	2015					
15	Marco	Male	1,079	2015					

6.5 Example 4 - DataFrames with Multi-hierarchical columns and indices

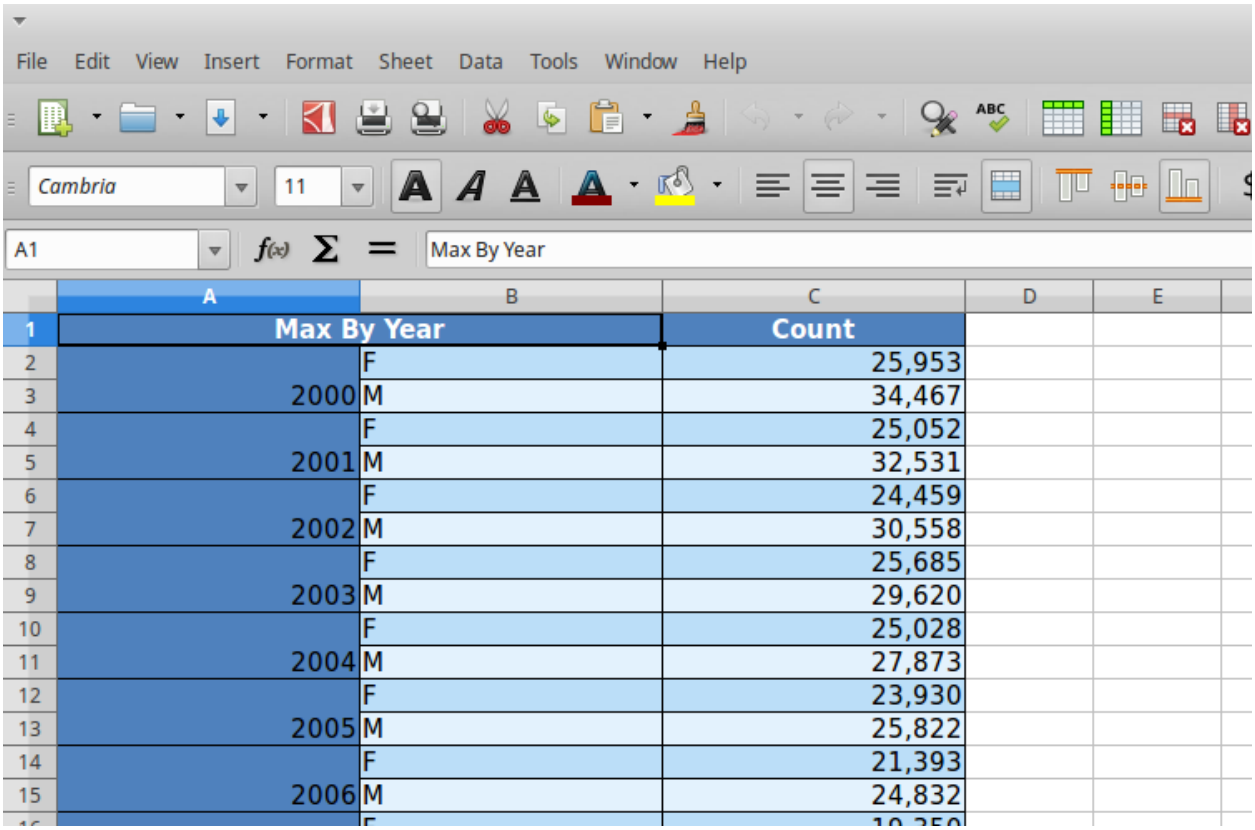
Demonstrates rendering dataframes with multi-hierarchical indices and mult-hierarchical columns

```
class XLSXExample4:
    '''
    Demonstrate styling and rendering of multi-hierarchical indexed dataframe
    into a xlsx file.
    '''

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            bg_color = None
            number_format = 'General'
            if col == 'count':
                number_format = '#,##0'
            if idx[1] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(
                bg_color=bg_color,
                number_format=number_format)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return 'Max By Year'

    @staticmethod
    def index_name_style_func(value):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
```

(continues on next page)

(continued from previous page)

```
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def index_value_func(node):
        if isinstance(node.value, str):
            return node.value.capitalize()
        return node.value

    @staticmethod
    def index_style_func(df):
        def _inner(node):
            bg_color = None
            if len(node.key) == 1:
                bg_color = '4f81bd'
            elif node.key[1] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(bg_color=bg_color)
        return _inner

    @classmethod
    def render_xlsx(cls):

        # Prepare first data frame (same as in render_xlsx)
        data_df = load_names_data()
        data_df = data_df[data_df['year'] >= 2000]
        g = data_df.groupby(('year', 'gender'))
        df = g.max()

        klass_ = cls
        pm = tc.build_presentation_model(
            df=df,
            output_format='xlsx',
            #data_value_func=None,   # use default
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        layout = [pm]
        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example4.xlsx')
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp, orientation=
↪'horizontal')
```



Max By Year		Count
2000	F	25,953
	M	34,467
2001	F	25,052
	M	32,531
2002	F	24,459
	M	30,558
2003	F	25,685
	M	29,620
2004	F	25,028
	M	27,873
2005	F	23,930
	M	25,822
2006	F	21,393
	M	24,832

CHAPTER 7

HTML Examples

This page provides a list of examples that demonstrate rendering html tables from the given dataframe. Each example is self-contained inside a class. We have some helper functions to provide the data we need for this examples

7.1 Helper Functions (Data loading routines)

```
import tempfile
import zipfile
import collections
import os
import webbrowser

import requests
import pandas as pd

import table_compositor.table_compositor as tc
import table_compositor.html_writer as htmlw
import table_compositor.html_styles as html_style

```

```
# code snippet adapted from http://function-pipe.readthedocs.io/en/latest/usage_df.
↪html
# source url
URL_NAMES = 'https://www.ssa.gov/oact/babynames/names.zip'
ZIP_NAME = 'names.zip'

def load_names_data():
    fp = os.path.join(tempfile.gettempdir(), ZIP_NAME)
    if not os.path.exists(fp):
        r = requests.get(URL_NAMES)
        with open(fp, 'wb') as f:
            f.write(r.content)
```

(continues on next page)

(continued from previous page)

```
    post = collections.OrderedDict()
    with zipfile.ZipFile(fp) as zf:
        # get ZipInfo instances
        for zi in sorted(zf.infolist(), key=lambda zi: zi.filename):
            fn = zi.filename
            if fn.startswith('yob'):
                year = int(fn[3:7])
                df = pd.read_csv(
                    zf.open(zi),
                    header=None,
                    names=('name', 'gender', 'count'))
                df['year'] = year
                post[year] = df

        df = pd.concat(post.values())
        df.set_index('name', inplace=True, drop=True)
        return df

def sample_names_data():
    df = load_names_data()
    df = df[(df['year'] == 2015) & (df['count'] > 1000)]
    return df.sample(50, random_state=0).sort_values('count')

def top_names_for_year(year=2015, gender='F', top_n=5):
    df = load_names_data()
    df = df[(df['year'] == year) & (df['gender'] == gender)]
    df = df.sort_values('count')[:top_n]
    return df
```

7.2 Example 1 - DataFrame with default styles

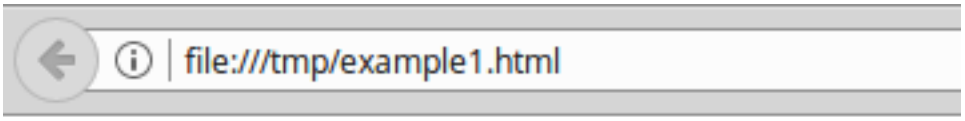
Demonstrates converting dataframe into html format with default styles.

```
class HTMLExample1:
    '''
    Demonstrate rendering of a simple dataframe into html
    '''
    @classmethod
    def render_html(cls):

        # load data
        df = load_names_data()
        df = df[:100]

        # build presentation model
        pm = tc.build_presentation_model(df=df, output_format='html')

        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example_1.html')
        layout = [pm]
        print('Writing to ' + fp)
        html = htmlw.HTMLWriter.to_html(layout, border=1)
```

(continues on next page)

(continued from previous page)

```
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example1.html')
        with open(output_fp, 'w') as f:
            f.write(html)
```



name	gender	count	year
Mary	F	7065	1880
Anna	F	2604	1880
Emma	F	2003	1880
Elizabeth	F	1939	1880
Minnie	F	1746	1880
Margaret	F	1578	1880
Ida	F	1472	1880

7.3 Example 2 - DataFrame with custom styles

In this example, we format the different components of dataframe with various styling attributes

```
class HTMLExample2:
    '''
    Demonstrate rendering of a simple dataframe into html
    '''

    @staticmethod
```

(continues on next page)

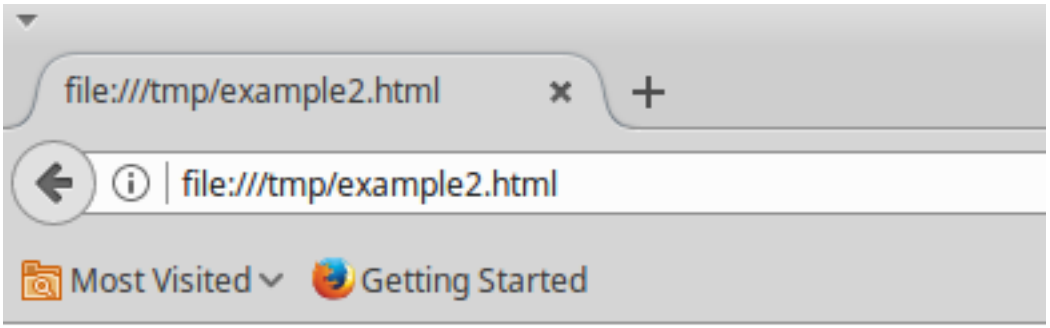
(continued from previous page)

```
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            color = '#FFFFFF'
            text_align = 'left'
            if col == 'count':
                text_align = 'right'
            if df.loc[idx, 'gender'] == 'F':
                color = '#bbdef8'
            else:
                color = '#e3f2fd'
            return html_style.td_style(
                text_align=text_align,
                background_color=color,
                color='#000000',
                font_weight='normal',
                white_space='pre',
                padding='10px',
                border=None)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return value.capitalize()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @staticmethod
    def index_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def index_style_func(node):
        return html_style.td_style(
            text_align='center',
```

(continues on next page)

(continued from previous page)

```
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @classmethod
    def render_html(cls):
        # load data
        df = sample_names_data()
        # build presentation model
        klass_ = HTMLExample2
        pm = tc.build_presentation_model(
            df=df,
            output_format='html',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=klass_.index_name_value_func)

        layout = [pm]
        html = htmlw.HTMLWriter.to_html(layout, border=1)
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example2.html')
        print('Writing to =', output_fp)
        with open(output_fp, 'w') as f:
            f.write(html)
```



Name	Gender	Count	Year
Paislee	Female	1008	2015
Kathryn	Female	1009	2015
Erik	Male	1012	2015
Lilah	Female	1022	2015
Fatima	Female	1041	2015
Dante	Male	1066	2015
Shelby	Female	1071	2015
Marco	Male	1079	2015
Diana	Female	1081	2015
Haley	Female	1128	2015

7.4 Example 3 - Simple DataFrame with Layouts

Demonstrates rendering dataframes with multi-hierarchical indices and mult-hierarchical columns

```
class HTMLExample3:
    '''
    Demonstrate styling and rendering of multiple multi-hierarchical indexed dataframe
    into a html file
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            color = '#FFFFFF'
            text_align = 'left'
            if col == 'count':
                text_align = 'right'
            if df.loc[idx, 'gender'] == 'F':
                color = '#bbdef8'
            else:
                color = '#e3f2fd'
            return html_style.td_style(
                text_align=text_align,
                background_color=color,
                color='#000000',
                font_weight='normal',
                white_space='pre',
                padding='10px',
                border=None)
        return _inner


    @staticmethod
    def index_name_value_func(value):
        return 'Max By Year'

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
```

(continues on next page)

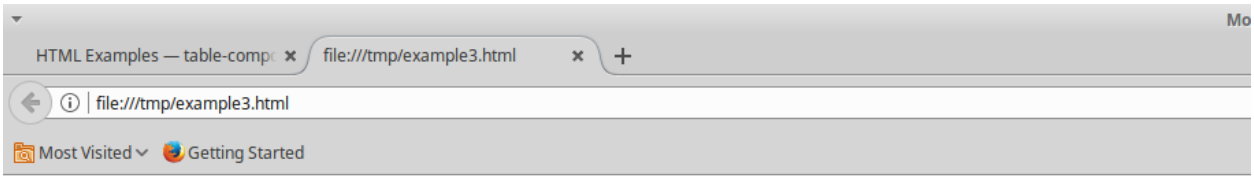
(continued from previous page)

```
            white_space='pre',
            padding='10px',
            border=1)

    @staticmethod
    def index_value_func(node):
        if isinstance(node.value, str):
            return node.value.capitalize()
        return node.value

    @staticmethod
    def index_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @classmethod
    def render_html(cls):

        # Prepare first data frame (same as in render_xlsx)
        df = sample_names_data()
        # build presentation model
        klass_ = HTMLExample4
        pm_all = tc.build_presentation_model(
            df=df,
            output_format='html',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=lambda _: 'Sample Data')

        male_df = top_names_for_year(gender='M')
        pm_top_male = tc.build_presentation_model(
            df=male_df,
            output_format='html',
            data_value_func=klass_.data_value_func(male_df),
            data_style_func=klass_.data_style_func(male_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=lambda _: 'Max by Year')

        female_df = top_names_for_year(gender='F')
        pm_top_female = tc.build_presentation_model(
            df=female_df,
            output_format='html',
            data_value_func=klass_.data_value_func(female_df),
            data_style_func=klass_.data_style_func(female_df),
```

(continues on next page)

(continued from previous page)

```
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=lambda _ : 'Max by Year')

        layout = [pm_all, [pm_top_female, pm_top_male]]
        # render to xlsx
        html = htmlw.HTMLWriter.to_html(
            layout, border=1, orientation='horizontal')
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example3.html')
        print('Writing to =', output_fp)
        with open(output_fp, 'w') as f:
            f.write(html)
```



Sample Data	Gender	Count	Year
Paislee	Female	1008	2015
Kathryn	Female	1009	2015
Erik	Male	1012	2015
Lilah	Female	1022	2015
Fatima	Female	1041	2015
Dante	Male	1066	2015
Shelby	Female	1071	2015
Marco	Male	1079	2015
Diana	Female	1081	2015
Haley	Female	1128	2015
Johnny	Male	1140	2015
Kali	Female	1179	2015

Max by Year	Gender	Count	Year
Zyrielle	Female	5	2015
Falak	Female	5	2015
Fairy	Female	5	2015
Faige	Female	5	2015
Faeryn	Female	5	2015

Max by Year	Gender	Count	Year
Zyus	Male	5	2015
Greylon	Male	5	2015
Greyton	Male	5	2015
Grigoriy	Male	5	2015
Gurtej	Male	5	2015

7.5 Example 4 - DataFrames with Multi-hierarchical columns and indices

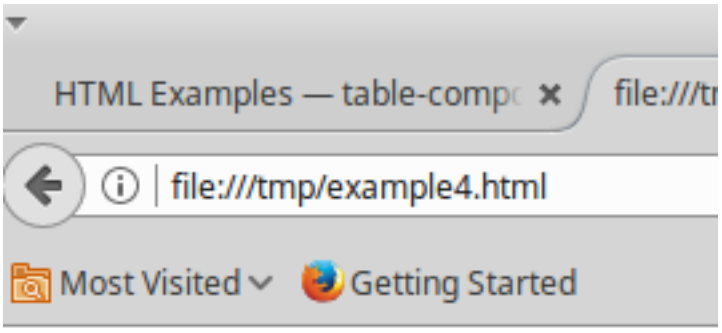
Demonstrates rendering dataframes with multi-hierarchical indices and mult-hierarchical columns

```
class HTMLExample4:
    '''
    Demonstrate styling and rendering of multi-hierarchical indexed dataframe
    into a html file.
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            color = '#FFFFFF'
            text_align = 'left'
            if col == 'count':
                text_align = 'right'
            if idx[1] == 'F':
                color = '#bbdef8'
            else:
                color = '#e3f2fd'

            return html_style.td_style(
                text_align=text_align,
                background_color=color,
                color='#000000',
                font_weight='normal',
                white_space='pre',
                padding='10px',
                border=None)
        return _inner


    @staticmethod
    def index_name_value_func(value):
        return 'Max By Year'

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return html_style.td_style(
            text_align='center',
```

(continues on next page)

(continued from previous page)

```
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @staticmethod
    def index_value_func(node):
        if isinstance(node.value, str):
            return node.value.capitalize()
        return node.value

    @staticmethod
    def index_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @classmethod
    def render_html(cls):

        # Prepare first data frame (same as in render_xlsx)
        data_df = load_names_data()
        data_df = data_df[data_df['year'] >= 2000]
        g = data_df.groupby(('year', 'gender'))
        df = g.max()

        klass_ = cls
        pm = tc.build_presentation_model(
            df=df,
            output_format='html',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=klass_.index_name_value_func)

        layout = [pm]
        # render to xlsx
        html = htmlw.HTMLWriter.to_html(layout, border=1)
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example4.html')
        print('Writing to =', output_fp)
        with open(output_fp, 'w') as f:
            f.write(html)
```



Max By Year		Count
2000	F	25953
	M	34467
2001	F	25052
	M	32531
2002	F	24459
	M	30558
2003	F	25685
	M	29620
2004	F	25028
	M	27873

CHAPTER 8

API

8.1 Building the presentation model

table_compositor.table_compositor.**build_presentation_model**(*, *df*, *output_format='xlsx'*, *data_value_func=None*, *column_style_func=None*, *data_style_func=None*, *header_style_func=None*, *header_value_func=None*, *index_style_func=None*, *index_value_func=None*, *index_name_func=None*, *index_name_style_func=None*, *engine='openpyxl'*, ***kwargs*)

Construct and return the presentation model that will be used while rendering to html/xlsx formats. The returned object has all the information required to render the tables in the requested format. The details of the object is transparent to the caller. It is only exposed for certain advanced operations.

Parameters

- **df** – The dataframe representation of the table. The shape of the dataframe closely resembles the table that will be rendered in the requested format.
- **output_format** – 'html' or 'xlsx'
- **data_value_func** – example: lambda idx, col: df.loc[idx, col], assuming df is in the closure. This can be None, if no data transformation is required to the values already present

in the source df

- **column_style_func** – the function can substitute the data_style_func, if the same style can be applied for the whole column. This argument should be prefered over the *data_style_func* argument. Using this option provides better performance since the fewer objects will be created internally and fewer callbacks are made to this function when compared to data_style_func.This argument only applies to the data contained in the dataframe and not the cell where the headers are rendered. For fine grained control at *cell* level, the *data_style_func* argument can be used. For more information on return values of this function, refer to the documentation for *data_style_func* argument.
- **data_style_func** – used to provide style at the cell level. Example: lambda idx, col: return dict(font=Font(...)), where Font is the openpyxl object and *font* is the attr available in the *cell* instance of openpyxl. For xlsx, the keys in the dict are the attrs of the *cell* object in openpyxl and the values correspond to the value of that attribute. Example are found in xlsx_styles module. For html, the key-value pairs are any values that go into to the style attribute of a td, th cell in html. Examples are found in html_styles module. example: dict(background-color='#F8F8F8'). When performance becomes an issue, and cell level control is not needed, it is recommended to use the *column_style_func* argument rathat than this argument. If the prefered engine is XlswWriter, then the style dictionary returned should have key/values compatible with the *Format* object declarted in the *XlsxWriter* library. A reference can be found in ‘‘xlsx_styles.XlsxWriterStyleHelper‘ class
- **header_value_func** – func that takes a object of type *IndexNode*. The *IndexNode* contains the attributes that refer to the header being rendered. The returned value from this function is displayed in place of the header in the dataframe at the location. The two properties available on the *IndexNode* object are *value* and *key*. The *key* is useful to identify the exact index and level in context while working with multi-hierarchical columns.
- **header_style_func** – func that takes a object of type *IndexNode*. The return value of this function is similar to data_style_func.
- **index_value_func** – func that takes a object of type *IndexNode*. The *IndexNode* contains the attributes that refer to the index being rendered. The returned value from this function is displayed in place of the index in the dataframe at the location.
- **index_style_func** – func that takes a object of type *IndexNode*. The return value of this function is similar to data_style_func.
- **index_name_func** – func that returns a string for index name (value to be displayed on top-left corner, above the index column)
- **index_name_style** – the style value same as data_style_func that will be used to style the cell
- **engine** – required while building presentation model for xlsx. Argument ignored for HTML rendering. This argument is used to provide the default callback style functions, where the style dictionary returned by the callback functions should be compatible with the engine being used.
- **kwargs** – ‘hide_index’ - if True, then hide the index column, default=False

  ’hide_header, - if True, then hide the header, default=False

  ’use_convert’ - if True, do some conversions from dataframe values to values excel can understand for example np.NaN are converted to NaN strings

Returns A presentation model, to be used to create layout and provide the layout to the html or xlsx writers.

About the callback functions provided as arguments:

Note that callback function provided as arguments to this function are provided with either a tuple of index, col arguments are some information regarding the index or headers being rendered. Therefore, a common pattern would be to capture the *dataframe* being rendered in a closure of this callback func before passing them as arugments.

For example:

df = pd.DataFrame(dict(a=[1, 2, 3]))

def data_value_func():

def _inner(idx, col): return df.loc[idx, col] * 10.3

return _inner

pm = build_presentation_model(df=df, data_value_func=data_value_func())

8.2 Rendering to XLSX

8.3 Rendering to HTML

class table_compositor.html_writer.**HTMLWriter**

static to_html(*layout*, *orientation='vertical'*, ***kwargs*)

Take a layout which contains a list of presentation models builts using the build_presentation_model function.

Parameters

- **layout** – An nested list of presentation_models, examples: [presentation_model] or [presentation_model1, presentation_mode2]. Not all nested layouts work very well in HTML, currently
- **orientation** – if vertical, the top level presentation model elements are rendered vertically, and for every nested level the orientation is flipped. if horizontal, then the behavior is inverse
- **kwargs** – all key-value pairs available in kwargs are directly set as value of the style attribute of *table* tag. example dict(backgroud-color='#FF88FF'), is used as <table style='background-color:#FF88FF'>..</table>

Returns Return a HTML formatted string. The outermost tag of the returned string is the <*table*>

8.4 Helper XLSX Styles

class table_compositor.xlsx_styles.**OpenPyxlStyleHelper**

default_header_style

Provides styles for default headers for OpenPyxl engine

Parameters

- **alignment** – 'center', 'left', 'right' used for horizontal alignment
- **font** – an openpyxl.Font instance

- **bgColor** – hex color that will be used as background color in the fill pattern
- **border** – an openpyxl.Border instance, defaults to thin white border

Returns A dict of key-values pairs, where each key is a attr of the *cell* object in openyxl and value is valid value of that attr.

get_style
Helper method to return a openpyxl Style

Parameters

- **number_format** – an xlsx compatibale number format string
- **bg_color** – hex color that will be used as background color in the fill pattern
- **border** – an openpyxl.Border instance, defaults to thin white border

Returns A dict of key-values pairs, where each key is a attr of the *cell* object in openyxl and value is valid value of that attr.

class table_compositor.xlsx_styles.**XlsxWriterStyleHelper**
Class provides style objects for XlsxWriter library uses to render xlsx files

static default_header_style(*, *number_format='General'*, *alignment='center'*, *font=None*, *bgColor='#4F81BD'*, *border=<object object>*)
Provides styles for default headers for XlsxWriter engine

Parameters

- **alignment** – 'center', 'left', 'right' used for horizontal alignment
- **font** – an openpyxl.Font instance
- **bgColor** – hex color that will be used as background color in the fill pattern
- **border** – an openpyxl.Border instance, defaults to thin white border

Returns A dict of key-values pairs, where each key is a attr of the *cell* object in openyxl and value is valid value of that attr.

static get_style(*number_format='General'*, *bg_color=None*, *border=<object object>*)
Helper method to return Style dictionary for XlsxWriter engine

Parameters

- **number_format** – an xlsx compatibale number format string
- **bg_color** – hex color that will be used as background color in the fill pattern
- **border** – an openpyxl.Border instance, defaults to thin white border

Returns A dict of key-values pairs, where each key/value is compatible with the *Format* object in XlsxWriter library.

class table_compositor.xlsx_styles.**XLSXWriterDefaults**
Class provides defaults callback funcs that can be used while calling the build_presentation_model.

static data_style_func(*df*)
Default value that can be used as callback for data_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes idx, col as arguments and returns a openpyxl compatible style dictionary

static data_value_func(*df*)
Default value that can be used as callback for data_value_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns A function that takes idx, col as arguments and returns the df.loc[idx, col] value

static header_style_func(*df*)
Default value that can be used as callback for data_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes *node* as arguments and returns a openpyxl compatible style dictionary

static header_value_func(*df*)
Default value that can be used as callback for data_header_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes *node* as arguments and returns node.value

static index_name_style_func(*df*)
Default value that can be used as callback for index_name_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes index.name as arguments and returns a openpyxl compatible style dictionary

static index_name_value_func(*df*)
Default value that can be used as callback for index_name_value_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes index.name as arguments and returns index.name if not None, else ‘’

static index_style_func(*df*)
Default value that can be used as callback for index_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes *node* as arguments and returns a openpyxl compatible style dictionary

static index_value_func(*df*)
Default value that can be used as callback for index_header_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes *node* as arguments and returns node.value

8.5 Helper HTML Styles

class table_compositor.html_styles.**HTMLWriterDefaults**
Class provides defaults callback funcs that can be used while calling the build_presentation_model.

static data_style_func(*df*)
Default value that can be used as callback for data_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes idx, col as arguments and returns a dictionary of html style attributes

static data_value_func(*df*, *dollar_columns=None*)
Default value that can be used as callback for data_value_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns A function that takes idx, col as arguments and returns the df.loc[idx, col] value

static header_style_func(*df*)
Default value that can be used as callback for header_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes *node* as argument and returns a dictionary of html style attributes

static header_value_func(*df*)
Default value that can be used as callback for header_value_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns A function that takes node as arguments and returns the node.value

static index_name_style_func(*df*)
Default value that can be used as callback for index_name_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes index.name as argument and returns a dictionary of html style attributes

static index_name_value_func(*df*)
Default value that can be used as callback for index_name_value_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns A function that takes index.name as argument and return index.name if not None else ‘’

static index_style_func(*df*)
Default value that can be used as callback for index_style_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns a function table takes *node* as argument and returns a dictionary of html style attributes

static index_value_func(*df*)
Default value that can be used as callback for index_value_func

Parameters **df** – the dataframe that will be used to build the presentation model

Returns A function that takes node as arguments and returns the node.value

CHAPTER 9

Code Used in documentation

9.1 Basic Usage

```
# start_imports
import os
import tempfile
import pandas as pd
from table_compositor.table_compositor import build_presentation_model

# There are equivalent classes for using xlsxwriter library. Namely,
# XlsxWriterCompositor and XlsxWriterStyleHelper
from table_compositor.xlsx_writer import OpenPyxlCompositor
from table_compositor.xlsx_styles import OpenPyxlStyleHelper

# end_imports

# start_basic_example_2
def basic_example2():

    df = pd.DataFrame(dict(a=[10, 20, 30, 40, 50], b=[0.1, 0.9,0.2, 0.6,0.3]),␣
↪index=[1,2,3,4,5])

    def style_func(idx, col):
        if col == 'b':
            return OpenPyxlStyleHelper.get_style(number_format='0.00%')
        else:
            # for 'a' we do dollar format
            return OpenPyxlStyleHelper.get_style(number_format='$#,##.00')

    # create a presentation model
    # note the OpenPyxlStyleHelper function available in xlsx_styles module. But a␣
↪return value of style function
    # can be any dict whose keys are attributes of the OpenPyxl cell object.
    presentation_model = build_presentation_model(
```

(continues on next page)

(continued from previous page)

```
        df=df,
        data_value_func=lambda idx, col: df.loc[idx, col] * 10 if col == 'a' else df.
↪loc[idx, col],
        data_style_func=style_func,
        header_value_func=lambda node: node.value.capitalize(),
        header_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
        index_value_func=lambda node: node.value * 100,
        index_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
        index_name_func=lambda _: 'Basic Example',
        index_name_style_func=lambda _: OpenPyxlStyleHelper.default_header_style())

    # create a layout, which is usually a nested list of presentation models
    layout = [presentation_model]

    # render to xlsx
    output_fp = os.path.join(tempfile.gettempdir(), 'basic_example2.xlsx')
    OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp)

# end_basic_example_2

# start_basic_example_3
def basic_example3():

    df = pd.DataFrame(dict(a=[10, 20, 30, 40],
                           b=[0.1, 0.9,0.2, 0.6],
                           d=[50, 60, 70, 80],
                           e=[200, 300, 400, 500]))
    df.columns = pd.MultiIndex.from_tuples([('A', 'x'), ('A', 'y'), ('B', 'x'), ('B',
↪'y')])
    df.index = pd.MultiIndex.from_tuples([(1, 100), (1, 200), (2, 100), (2, 200)])
    print(df)

    def index_style_func(node):
        # node.key here could be one of (1,), (1, 100), (2,), (2, 100), (2, 200)
        bg_color = 'FFFFFF'
        if node.key == (1,) or node.key == (2,):
            bg_color = '9E80B8'
        elif node.key[1] == 100:
            bg_color = '4F90C1'
        elif node.key[1] == 200:
            bg_color = '6DC066'
        return OpenPyxlStyleHelper.get_style(bg_color=bg_color)

    def header_style_func(node):
        bg_color = 'FFFFFF'
        if node.key == ('A',) or node.key == ('B',):
            bg_color = '9E80B8'
        elif node.key[1] == 'x':
            bg_color = '4F90C1'
        elif node.key[1] == 'y':
            bg_color = '6DC066'
        return OpenPyxlStyleHelper.get_style(bg_color=bg_color)

    # create a presentation model
    # note the OpenPyxlStyleHeloer function available in xlsx_styles module. But a␣
↪return value of style function
    # can be any dict whose keys are attributes of the OpenPyxl cell object.
```

(continues on next page)

(continued from previous page)

```
    presentation_model = build_presentation_model(
        df=df,
        index_style_func=index_style_func,
        header_style_func=header_style_func,
        index_name_func=lambda _: 'Multi-Hierarchy Example')

    # create a layout, which is usually a nested list of presentation models
    layout = [presentation_model]

    # render to xlsx
    output_fp = os.path.join(tempfile.gettempdir(), 'basic_example3.xlsx')
    OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp)

# end_basic_example_3


# start_layout_example_1
def layout_example1():

    df = pd.DataFrame(dict(a=[10, 20, 30, 40, 50], b=[0.1, 0.9,0.2, 0.6,0.3]),␣
↪index=[1,2,3,4,5])

    def style_func(idx, col):
        if col == 'b':
            return OpenPyxlStyleHelper.get_style(number_format='0.00%')
        else:
            # for 'a' we do dollar format
            return OpenPyxlStyleHelper.get_style(number_format='$#,##.00')

    # create a presentation model
    # note the OpenPyxlStyleHeloer function available in xlsx_styles module. But a␣
↪return value of style function
    # can be any dict whose keys are attributes of the OpenPyxl cell object.
    presentation_model = build_presentation_model(
        df=df,
        data_value_func=lambda idx, col: df.loc[idx, col] * 10 if col == 'a' else df.
↪loc[idx, col],
        data_style_func=style_func,
        header_value_func=lambda node: node.value.capitalize(),
        header_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
        index_value_func=lambda node: node.value * 100,
        index_style_func=lambda _: OpenPyxlStyleHelper.default_header_style(),
        index_name_func=lambda _: 'Basic Example',
        index_name_style_func=lambda _: OpenPyxlStyleHelper.default_header_style())

    # start_layout_code_1
    # create a layout, which is usually a nested list of presentation models
    layout = [[presentation_model], [[presentation_model], [presentation_model]]]

    # render to xlsx
    output_fp = os.path.join(tempfile.gettempdir(), 'layout_vertical_example1.xlsx')
    # the default value for orientation is 'vertical'
    OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp, orientation=
↪'vertical')

    output_fp = os.path.join(tempfile.gettempdir(), 'layout_horizontal_example1.xlsx')
    OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=output_fp, orientation=
↪'horizontal')
```

(continues on next page)

(continued from previous page)

```
    print('Writing xlsx file=', output_fp)

    # mutiple nesting
    layout_complex = [presentation_model,
                      [presentation_model, [presentation_model, presentation_model]]]

    output_fp = os.path.join(tempfile.gettempdir(), 'layout_complex_example1.xlsx')
    OpenPyxlCompositor.to_xlsx(layout=layout_complex, output_fp=output_fp,␣
↪orientation='vertical')
    print('Writing xlsx file=', output_fp)
    # end_layout_code_1

# end_layout_example_1


if __name__ == '__main__'    :
    basic_example2()
    basic_example3()
    layout_example1()
```

9.2 XLSX Examples

```
'''
This module is referred to by the Sphinx documentation. If you need to run this
module, install table_compositor in an separate  environment and then run this module
in that environment. This helps the imports find the modules in the right place
'''

# start_imports
import tempfile
import zipfile
import collections
import os
import webbrowser

import requests
import pandas as pd

import table_compositor.table_compositor as tc
import table_compositor.xlsx_writer as xlsxw
import table_compositor.xlsx_styles as xlsstyle
# end_imports

# start_data_routine
# code snippet adapted from http://function-pipe.readthedocs.io/en/latest/usage_df.
↪html
# source url
URL_NAMES = 'https://www.ssa.gov/oact/babynames/names.zip'
ZIP_NAME = 'names.zip'

def load_names_data():
    fp = os.path.join(tempfile.gettempdir(), ZIP_NAME)
    if not os.path.exists(fp):
        r = requests.get(URL_NAMES)
```

(continues on next page)

(continued from previous page)

```
        with open(fp, 'wb') as f:
            f.write(r.content)

    post = collections.OrderedDict()
    with zipfile.ZipFile(fp) as zf:
        # get ZipInfo instances
        for zi in sorted(zf.infolist(), key=lambda zi: zi.filename):
            fn = zi.filename
            if fn.startswith('yob'):
                year = int(fn[3:7])
                df = pd.read_csv(
                    zf.open(zi),
                    header=None,
                    names=('name', 'gender', 'count'))
                df['year'] = year
                post[year] = df

        df = pd.concat(post.values())
        df.set_index('name', inplace=True, drop=True)
        return df

def sample_names_data():
    df = load_names_data()
    df = df[(df['year'] == 2015) & (df['count'] > 1000)]
    return df.sample(100, random_state=0).sort_values('count')

def top_names_for_year(year=2015, gender='F', top_n=5):
    df = load_names_data()
    df = df[(df['year'] == year) & (df['gender'] == gender)]
    df = df.sort_values('count')[:top_n]
    return df
# end_data_routine

# start_XLSXExample1
class XLSXExample1:
    '''
    Demonstrates rendering a simple dataframe to a xlsx file
    using the default styles
    '''
    @classmethod
    def render_xlsx(cls):
        '''
        Render the df to a xlsx file.
        '''

        # load data
        df = sample_names_data()
        # build presentation model
        pm = tc.build_presentation_model(df=df, output_format='xlsx')

        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example1.xlsx')
        layout = [pm]
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp)
# end_XLSXExample1
```

(continues on next page)

(continued from previous page)

```
# start_XLSXExample2
class XLSXExample2:
    '''
    Demonstrates using call-backs that help set the display and style
    properties of each cell in the xlsx sheet.
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            bg_color = None
            number_format='General'
            if col == 'count':
                number_format='#,##0'
            if df.loc[idx, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(
                bg_color=bg_color,
                number_format=number_format)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return value.capitalize()

    @staticmethod
    def index_name_style_func(value):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def index_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def index_style_func(df):
```

(continues on next page)

(continued from previous page)

```
        def _inner(node):
            bg_color = None
            if df.loc[node.value, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(bg_color=bg_color)
        return _inner

    @classmethod
    def render_xlsx(cls):
        # load data
        df = sample_names_data()
        # build presentation model
        klass_ = XLSXExample2
        pm = tc.build_presentation_model(
            df=df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example2.xlsx')
        layout = [pm]
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp)
# end_XLSXExample2

# start_XLSXExample3
class XLSXExample3:
    '''
    Demonstrates using call-backs and also rendering multiple tables to single
    worksheet.
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            bg_color = None
            number_format='General'
```

(continues on next page)

(continued from previous page)

```
            if col == 'count':
                number_format='#,##0'
            if df.loc[idx, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(
                bg_color=bg_color,
                number_format=number_format)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return value.capitalize()

    @staticmethod
    def index_name_style_func(value):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def index_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def index_style_func(df):
        def _inner(node):
            bg_color = None
            if df.loc[node.value, 'gender'] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(bg_color=bg_color)
        return _inner


    @classmethod
    def render_xlsx(cls):
        # Prepare first data frame (same as in render_xlsx)
        df = sample_names_data()
        # build presentation model
        klass_ = XLSXExample3
        pm_all = tc.build_presentation_model(
            df=df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(df),
```

(continues on next page)

(continued from previous page)

```
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        male_df = top_names_for_year(gender='M')
        pm_top_male = tc.build_presentation_model(
            df=male_df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(male_df),
            data_style_func=klass_.data_style_func(male_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(male_df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        female_df = top_names_for_year(gender='F')
        pm_top_female = tc.build_presentation_model(
            df=female_df,
            output_format='xlsx',
            data_value_func=klass_.data_value_func(female_df),
            data_style_func=klass_.data_style_func(female_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(female_df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)


        layout = [pm_all, [pm_top_female, pm_top_male]]
        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example3.xlsx')
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp, orientation=
↪'horizontal')
# end_XLSXExample3

# start_XLSXExample4
class XLSXExample4:
    '''
    Demonstrate styling and rendering of multi-hierarchical indexed dataframe
    into a xlsx file.
    '''

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            bg_color = None
            number_format = 'General'
            if col == 'count':
                number_format = '#,##0'
            if idx[1] == 'F':
                bg_color = 'bbdef8'
            else:
```

(continues on next page)

(continued from previous page)

```
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(
                bg_color=bg_color,
                number_format=number_format)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return 'Max By Year'

    @staticmethod
    def index_name_style_func(value):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return xlsstyle.OpenPyxlStyleHelper.default_header_style()

    @staticmethod
    def index_value_func(node):
        if isinstance(node.value, str):
            return node.value.capitalize()
        return node.value

    @staticmethod
    def index_style_func(df):
        def _inner(node):
            bg_color = None
            if len(node.key) == 1:
                bg_color = '4f81bd'
            elif node.key[1] == 'F':
                bg_color = 'bbdef8'
            else:
                bg_color = 'e3f2fd'
            return xlsstyle.OpenPyxlStyleHelper.get_style(bg_color=bg_color)
        return _inner

    @classmethod
    def render_xlsx(cls):

        # Prepare first data frame (same as in render_xlsx)
        data_df = load_names_data()
        data_df = data_df[data_df['year'] >= 2000]
        g = data_df.groupby(('year', 'gender'))
        df = g.max()

        klass_ = cls
        pm = tc.build_presentation_model(
            df=df,
            output_format='xlsx',
            #data_value_func=None,    # use default
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
```

(continues on next page)

(continued from previous page)

```
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func(df),
            index_value_func=klass_.index_value_func,
            index_name_style_func=klass_.index_name_style_func,
            index_name_func=klass_.index_name_value_func)

        layout = [pm]
        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example4.xlsx')
        print('Writing to ' + fp)
        xlsxw.OpenPyxlCompositor.to_xlsx(layout=layout, output_fp=fp, orientation=
↪'horizontal')
# end_XLSXExample4

def main():
    XLSXExample1.render_xlsx()
    XLSXExample2.render_xlsx()
    XLSXExample3.render_xlsx()
    XLSXExample4.render_xlsx()

if __name__ == '__main__':
    main()
```

9.3 HTML Examples

```
'''
This module is referred to by the Sphinx documentation. If you need to run this
module, install table_compositor in an separate  environment and then run this module
in that environment. This helps the imports find the modules in the right place
'''


# start_imports
import tempfile
import zipfile
import collections
import os
import webbrowser

import requests
import pandas as pd

import table_compositor.table_compositor as tc
import table_compositor.html_writer as htmlw
import table_compositor.html_styles as html_style

# end_imports

# start_data_routine
# code snippet adapted from http://function-pipe.readthedocs.io/en/latest/usage_df.
↪html
# source url
URL_NAMES = 'https://www.ssa.gov/oact/babynames/names.zip'
```

(continues on next page)

(continued from previous page)

```
ZIP_NAME = 'names.zip'

def load_names_data():
    fp = os.path.join(tempfile.gettempdir(), ZIP_NAME)
    if not os.path.exists(fp):
        r = requests.get(URL_NAMES)
        with open(fp, 'wb') as f:
            f.write(r.content)

    post = collections.OrderedDict()
    with zipfile.ZipFile(fp) as zf:
        # get ZipInfo instances
        for zi in sorted(zf.infolist(), key=lambda zi: zi.filename):
            fn = zi.filename
            if fn.startswith('yob'):
                year = int(fn[3:7])
                df = pd.read_csv(
                    zf.open(zi),
                    header=None,
                    names=('name', 'gender', 'count'))
                df['year'] = year
                post[year] = df

        df = pd.concat(post.values())
        df.set_index('name', inplace=True, drop=True)
        return df

def sample_names_data():
    df = load_names_data()
    df = df[(df['year'] == 2015) & (df['count'] > 1000)]
    return df.sample(50, random_state=0).sort_values('count')

def top_names_for_year(year=2015, gender='F', top_n=5):
    df = load_names_data()
    df = df[(df['year'] == year) & (df['gender'] == gender)]
    df = df.sort_values('count')[:top_n]
    return df
# end_data_routine

# start_HTMLExample1
class HTMLExample1:
    '''
    Demonstrate rendering of a simple dataframe into html
    '''
    @classmethod
    def render_html(cls):

        # load data
        df = load_names_data()
        df = df[:100]

        # build presentation model
        pm = tc.build_presentation_model(df=df, output_format='html')

        # render to xlsx
        tempdir = tempfile.gettempdir()
        fp = os.path.join(tempdir, 'example_1.html')
```

(continues on next page)

(continued from previous page)

```
        layout = [pm]
        print('Writing to ' + fp)
        html = htmlw.HTMLWriter.to_html(layout, border=1)
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example1.html')
        with open(output_fp, 'w') as f:
            f.write(html)
# end_HTMLExample1

# start_HTMLExample2
class HTMLExample2:
    '''
    Demonstrate rendering of a simple dataframe into html
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            color = '#FFFFFF'
            text_align = 'left'
            if col == 'count':
                text_align = 'right'
            if df.loc[idx, 'gender'] == 'F':
                color = '#bbdef8'
            else:
                color = '#e3f2fd'
            return html_style.td_style(
                text_align=text_align,
                background_color=color,
                color='#000000',
                font_weight='normal',
                white_space='pre',
                padding='10px',
                border=None)
        return _inner

    @staticmethod
    def index_name_value_func(value):
        return value.capitalize()

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
```

(continues on next page)

(continued from previous page)

```
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @staticmethod
    def index_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def index_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @classmethod
    def render_html(cls):
        # load data
        df = sample_names_data()
        # build presentation model
        klass_ = HTMLExample2
        pm = tc.build_presentation_model(
            df=df,
            output_format='html',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=klass_.index_name_value_func)

        layout = [pm]
        html = htmlw.HTMLWriter.to_html(layout, border=1)
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example2.html')
        print('Writing to =', output_fp)
        with open(output_fp, 'w') as f:
            f.write(html)
# end_HTMLExample2


# start_HTMLExample3
class HTMLExample3:
    '''
    Demonstrate styling and rendering of multiple multi-hierarchical indexed dataframe
    into a html file
```

(continues on next page)

(continued from previous page)

```
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            color = '#FFFFFF'
            text_align = 'left'
            if col == 'count':
                text_align = 'right'
            if df.loc[idx, 'gender'] == 'F':
                color = '#bbdef8'
            else:
                color = '#e3f2fd'
            return html_style.td_style(
                text_align=text_align,
                background_color=color,
                color='#000000',
                font_weight='normal',
                white_space='pre',
                padding='10px',
                border=None)
        return _inner


    @staticmethod
    def index_name_value_func(value):
        return 'Max By Year'

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()

    @staticmethod
    def header_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @staticmethod
    def index_value_func(node):
        if isinstance(node.value, str):
            return node.value.capitalize()
```

(continues on next page)

(continued from previous page)

```
        return node.value

    @staticmethod
    def index_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @classmethod
    def render_html(cls):

        # Prepare first data frame (same as in render_xlsx)
        df = sample_names_data()
        # build presentation model
        klass_ = HTMLExample4
        pm_all = tc.build_presentation_model(
            df=df,
            output_format='html',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=lambda _: 'Sample Data')

        male_df = top_names_for_year(gender='M')
        pm_top_male = tc.build_presentation_model(
            df=male_df,
            output_format='html',
            data_value_func=klass_.data_value_func(male_df),
            data_style_func=klass_.data_style_func(male_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=lambda _: 'Max by Year')

        female_df = top_names_for_year(gender='F')
        pm_top_female = tc.build_presentation_model(
            df=female_df,
            output_format='html',
            data_value_func=klass_.data_value_func(female_df),
            data_style_func=klass_.data_style_func(female_df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=lambda _ : 'Max by Year')

        layout = [pm_all, [pm_top_female, pm_top_male]]
        # render to xlsx
```

(continues on next page)

(continued from previous page)

```
        html = htmlw.HTMLWriter.to_html(
            layout, border=1, orientation='horizontal')
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example3.html')
        print('Writing to =', output_fp)
        with open(output_fp, 'w') as f:
            f.write(html)
# end_HTMLExample3

# start_HTMLExample4
class HTMLExample4:
    '''
    Demonstrate styling and rendering of multi-hierarchical indexed dataframe
    into a html file.
    '''

    @staticmethod
    def data_value_func(df):
        def _inner(idx, col):
            if col == 'gender':
                if df.loc[idx, col] == 'F':
                    return "Female"
                return 'Male'
            return df.loc[idx, col]
        return _inner

    @staticmethod
    def data_style_func(df):
        def _inner(idx, col):
            color = '#FFFFFF'
            text_align = 'left'
            if col == 'count':
                text_align = 'right'
            if idx[1] == 'F':
                color = '#bbdef8'
            else:
                color = '#e3f2fd'

            return html_style.td_style(
                text_align=text_align,
                background_color=color,
                color='#000000',
                font_weight='normal',
                white_space='pre',
                padding='10px',
                border=None)
        return _inner


    @staticmethod
    def index_name_value_func(value):
        return 'Max By Year'

    @staticmethod
    def header_value_func(node):
        return node.value.capitalize()
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def header_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @staticmethod
    def index_value_func(node):
        if isinstance(node.value, str):
            return node.value.capitalize()
        return node.value

    @staticmethod
    def index_style_func(node):
        return html_style.td_style(
            text_align='center',
            background_color='#4F81BD',
            color='#FFFFFF',
            font_weight='bold',
            white_space='pre',
            padding='10px',
            border=1)

    @classmethod
    def render_html(cls):

        # Prepare first data frame (same as in render_xlsx)
        data_df = load_names_data()
        data_df = data_df[data_df['year'] >= 2000]
        g = data_df.groupby(('year', 'gender'))
        df = g.max()

        klass_ = cls
        pm = tc.build_presentation_model(
            df=df,
            output_format='html',
            data_value_func=klass_.data_value_func(df),
            data_style_func=klass_.data_style_func(df),
            header_value_func=klass_.header_value_func,
            header_style_func=klass_.header_style_func,
            index_style_func=klass_.index_style_func,
            index_value_func=klass_.index_value_func,
            index_name_func=klass_.index_name_value_func)

        layout = [pm]
        # render to xlsx
        html = htmlw.HTMLWriter.to_html(layout, border=1)
        output_fp = os.path.join(
            tempfile.gettempdir(),
            'example4.html')
        print('Writing to =', output_fp)
```

(continues on next page)

(continued from previous page)

```
        with open(output_fp, 'w') as f:
            f.write(html)
# end_HTMLExample4


def main():
    HTMLExample1.render_html()
    HTMLExample2.render_html()
    HTMLExample3.render_html()
    HTMLExample4.render_html()


if __name__ == '__main__':
    main()
```

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index

X