PyBloqs Documentation

Release 1.0.0

Man AHL Technology

Aug 21, 2017

Contents

CHAPTER 1

Installation

For installation steps, please refer to the project README.

CHAPTER 2

General use

The following section is available as a Jupyter notebook in *docs/source/examples.ipynb*.

Composable Blocks & Interactive Charts

This notebook is a simple illustration of the python API for Blocks and Highcharts interactive charts.

What are Blocks? Blocks are composable layout elements for easily building HTML, PDF, PNG and JPG based reports. At the same time, all block constructs can be rendered in-line in IPython Notebooks (as will be shown later). This has practical benefits like a single backtest function that can be used for quick analysis during research work in a notebook, but also directly injected into more formal reports with having to fumble around with intermediate formats.

Practically all the functionality is based on HTML rendering. HTML is a declarative, tree based language that is easy to work with and fungible. Blocks are also declarative, composable into a tree and are meant to be dead simple. The match was thus quite natural.

The blocks do not try to match the power and precision of latex. Such an undertaking would be not only out of the scope of a simple library, but would mean the reinvention of latex with all the gnarliness that comes with it.

This notebook aims to showcase the functionality and offers some examples to help people get started.

Imports & Data

```
In [9]: import numpy as np
        import pandas as pd
        import pandas.util.testing as pt
        from datetime import datetime

        # === Begin bits required for interactive plotting and reports ===
        from pybloqs import *
        from pybloqs.plot import *

        interactive()
        # === End interactive bits ===
```

```
In [10]: wp = pt.makePanel().ix[:, :, :2]

         df = pd.DataFrame((np.random.rand(200, 4)-0.5)/10,
                           columns=list("ABCD"),
                           index=pd.date_range(datetime(2000,1,1), periods=200))

         df_cr = (df + 1).cumprod()

         a = df_cr.A
         b = df_cr.B
         c = df_cr.C
         c.name = "C"
```

Using Blocks

Obligatory "Hello World!"

```
In [11]: Block("Hello World!")
```

```
Out[11]: <pybloqs.block.text.Raw at 0x7f658ec3be50>
```

Play around with alignment

```
In [12]: Block("Hello World!", h_align="left")
```

```
Out[12]: <pybloqs.block.text.Raw at 0x7f658eb7f310>
```

Adding a title

```
In [13]: Block("Hello World!", title="Announcement", h_align="left")
```

```
Out[13]: <pybloqs.block.text.Raw at 0x7f658eb7f810>
```

Writing out dataframes

```
In [14]: Block(df.head())
```

```
Out[14]: <pybloqs.block.table.HTMLJinjaTableBlock at 0x7f658eb7f690>
```

Writing out matplotlib plots

```
In [15]: Block(df.A.plot())
```

```
Out[15]: <pybloqs.block.image.PlotBlock at 0x7f658ec3bdd0>
```

Raw HTML output

```
In [16]: Block("<b>this text is bold</b>")
```

```
Out[16]: <pybloqs.block.text.Raw at 0x7f658efbf2d0>
```

Composing blocks

```
In [17]: Block([Block("Hello World!", title="Announcement"), Block("<b>this text is bold</b>")])
```

```
Out[17]: <pybloqs.block.layout.Grid at 0x7f65950d6e90>
```

In most cases, one does not need to explicitly wrap elements in blocks

```
In [18]: Block(["Block %s" % i for i in xrange(8)])
```

```
Out[18]: <pybloqs.block.layout.Grid at 0x7f65950ced10>
```

Splitting composite blocks into columns

```
In [19]: Block(["Block %s" % i for i in xrange(8)], cols=4)
```

```
Out[19]: <pybloqs.block.layout.Grid at 0x7f658eb7f7d0>
```

Layout styling is cascading - styles will cascade from parent blocks to child blocks by default. This behavior can be disabled by setting *inherit_cfg* to false on the child blocks, or simply specifying the desired settings explicitly.

```
In [20]: Block(["Block %s" % i for i in xrange(8)], cols=4, text_align="right")
```

```
Out[20]: <pybloqs.block.layout.Grid at 0x7f658eb95450>
```

Using specific block types is simple as well. As an example - the Paragraph block:

```
In [21]: Block([Paragraph("First paragraph."),
                Paragraph("Second paragraph."),
                Paragraph("Third paragraph.")], text_align="right")
```

```
Out[21]: <pybloqs.block.layout.Grid at 0x7f658eb95550>
```

The Pre block preserves whitespace formatting and is rendered using a fixed width font. Useful for rendering code-like text.

```
In [22]: Pre("""
         some:
           example:
             yaml: [1,2,3]
           data: "text"
         """)
```

```
Out[22]: <pybloqs.block.text.Pre at 0x7f658eb95410>
```

Creating custom blocks is trivial. For the majority of the cases, one can just inherit from the Container block, which has most of the plumbing already in place:

```
In [23]: class Capitalize(Raw):
             def __init__(self, contents, **kwargs):
                 # Stringify and capitalize
                 contents = str(contents).upper()

                 super(Capitalize, self).__init__(contents, **kwargs)

         Capitalize("this here text should look like shouting!")
```

```
Out[23]: <__main__.Capitalize at 0x7f65950cd790>
```

Simple line chart

When evaluated as the last expression in a Notebook Cell, the plot is automatically displayed inline.

Note how the plot name (hover over the line to see the little popup) is taken from the input data (if available).

```
In [24]: Plot(a)
```

```
Out[24]: <pybloqs.plot.core.Plot at 0x7f65950c12d0>
```

Saving as interactive HTML

```
In [25]: from pybloqs.plot import Plot
         Plot(a).save("chart_sample.html")
```

```
Out[25]: 'chart_sample.html'
```

Scatter Plot

Regular scatter plot, with zooming on both the x and y axes.

```
In [26]: Plot(df.values[:,:2], Scatter(Marker(enabled=True)), Chart(zoom_type="xy"))
```

```
Out[26]: <pybloqs.plot.core.Plot at 0x7f658ec3ba10>
```

Bar Charts

Notice how when viewing all the data at once, the chart shows monthly data, yet zooming in reveals additional detail at up to daily resolution. This is accomplished by using a custom data grouping.

```
In [27]: bar_grouping = DataGrouping(approximation="open", enabled=True, group_pixel_width=100)
```

```
In [28]: # Bar chart from a dataframe
         Plot(df, Column(bar_grouping))
```

```
Out[28]: <pybloqs.plot.core.Plot at 0x7f65950d6790>
```

Stacked bar chart

Plot(df, Column(bar_grouping, stacking="normal"))

```
In [29]: # Composite bar chart from two separate plots.
         s2 = Plot([Plot(a, Column(bar_grouping)),
                    Plot(b, Column(bar_grouping))])
         s2
```

```
Out[29]: <pybloqs.plot.core.Plot at 0x7f65950cdc90>
```

Comparing series in a dataframe

Plot the cumulative percentage difference between input series (or columns of a dataframe). The cumulative difference is always calculated from the start of the observation window. This results in intuitively correct behavior when zooming in or sliding the window, as the chart will dynamically recalculate the values. Incredibly useful for comparing model performance over time for example as one doesn't need to manually normalize money curves for different periods.

```
In [30]: s3 = Plot(df_cr,
                   PlotOptions(Series(compare="percent")),
                   TooltipPct(),
                   YAxisPct())
         s3
```

```
Out[30]: <pybloqs.plot.core.Plot at 0x7f65950d67d0>
```

Three series on separate side-by-side Y axes

```
In [31]: s4 = Plot([Plot(a),
                    Plot(b, YAxis(Title(text="B Axis"), opposite=True)),
                    Plot(c, YAxis(Title(text="C Axis"), opposite=True, offset=40))])
         s4
```

```
Out[31]: <pybloqs.plot.core.Plot at 0x7f65950d6f50>
```

Two series on separate subplots

```
In [32]: s5 = Plot([Plot(a, Line(), YAxis(Title(text="a only"), height=150)),
                    Plot(b, Column(), YAxis(Title(text="b only"),
                                                top=200, height=100, offset=0))],
                   Tooltip(value_decimals=2), height="400px")
         s5
```

```
Out[32]: <pybloqs.plot.core.Plot at 0x7f65950cddd0>
```

Writing out multiple charts as HTML

```
In [33]: b = Block([Block(pt.makeTimeDataFrame().tail(10), title="A table", title_level=1),
                    Block([s2, s3], title="Side-by-side Plots", cols=2),
                    Block(title="Full Width Plots", break_before="always"),
                    Block(s4, title="Side by Side Axes"),
                    Block(s5, title="Composite Plots")], title="Dynamic Reports")
```

```
In [34]: b.save("charts_test.pdf")
         b.save("charts_test.html")
```

```
['/users/is/dchrist/pyenvs/dev_RHEL7_pydev/bin/wkhtmltopdf', '--debug-javascript', '--page-size', 'A4

None
```

```
Out[34]: 'charts_test.html'
```

```
In [35]: # Emails the report. The emailing is independent of previous reports being saved (e.g. there
         # before emailing).
         from smtplib import SMTPServerDisconnected

         try:
             b.email(fmt="html")
         except SMTPServerDisconnected:
             print "Please create ~/.pybloqs.cfg with entry for smtp_server . See README.md and pybl
```

```
Please create ~/.pybloqs.cfg with entry for smtp_server . See README.md and pybloqs/config.py for det
```

CHAPTER 3

Table formatting

The following section is available as a Jupyter notebook in *docs/source/table_formatting.ipynb*.

HTML Tables and Table Formatting

Creating a HTML Table from pandas.DataFrame

The following is hopefully sufficient for most applications (feedback welcome!):

```
from pybloqs import Blockb = Block(df)
```

When called only with DataFrame as parameter, a set of default formatters is applied:

```
table_block = Block(df, formatters=None, use_default_formatters=True)
```

```
In [3]: import pybloqs as abl
        from pybloqs import Block
        import pandas as pd
        import numpy as np
        df = pd.DataFrame(np.random.rand(4,4), index=['a','b','c','d'], columns = ['aa','bb','cc','dd
        df.index.name = 'ABC'
        table_block = Block(df)
        output = table_block.render_html()


        # Displaying pybloqs out in jupyter requires rendering html output
        from IPython.core.display import display, HTML
        display(HTML(output))
```

```
<IPython.core.display.HTML object>
```

NB: The grid between cells is from jupyter default CSS. It will not show if the block is displayed with b.show() .

Formatting Tables with Table Formatters

Formatters are functions which add a single specific formatting aspect (e.g. bold, font-size, alignment, multi-index display). Formatters can be stacked together as a list to produce desired layout. The list is then passed to `HTMLJinjaTableBlock`.

Use of default formatters can be disabled completely. Additional formatters can be used on top or instead of default formatters. Formatters change appearance by modifying cell values and adding CSS styles.

'Exotic' formatters, which are used only in a single context, can be defined locally. A set of general use formatters can be found in pybloqs.block.table_formatters.

All formatters take the following parameters:

```
:rows List of rows, where formatter should be applied
:columns List of columns, where formatter should be applied
:apply_to_header_and_index True/False, if set to True, formatter will be applied to␣
↪all index and header cells
```

If rows and columns are both `None`, formatter is applied to all cells in table.

An example:

```
In [4]: import pybloqs.block.table_formatters as tf
        table_block = Block(df)
        table_block_raw = Block(df, use_default_formatters=False)

        fmt_pct = tf.FmtPercent(1, columns=['bb','cc'], apply_to_header_and_index=False)
        fmt_totals = tf.FmtAppendTotalsRow(total_columns=['aa','dd'])
        fmt_highlight = tf.FmtHighlightText(columns=['bb'], rows=['d'], apply_to_header_and_index=Fal
        formatters=[fmt_pct,fmt_totals, fmt_highlight]
        table_block_additional_formatters = Block(df, formatters=formatters)

        fmt_mult = tf.FmtMultiplyCellValue(1e6, '')
        fmt_sep = tf.FmtThousandSeparator()
        formatters=[fmt_mult, fmt_sep]
        table_block_new_formatters = Block(df, formatters=formatters, use_default_formatters=False)

        row1 = abl.HStack([table_block, table_block_raw])
        row2 = abl.HStack([table_block_additional_formatters, table_block_new_formatters])
        all_tables = abl.VStack([row1,row2])

        display(HTML(all_tables.render_html()))


<IPython.core.display.HTML object>
```

General formatters

The following formatters handle miscalleneous general tasks #### Replace NaN

```
FmtReplaceNaN(value='')
```

Replaces all np.nan values in specified range with provided `value`. By default uses empty string.

FmtAlignCellContents

```
FmtAlignCellContents(alignment='center')
```

Aligns content inside cells within specified range. Valid values for `alignment` are `left|right|center|justify|initial|inherit` (anything that the CSS tag `text-align` understands).

FmtAlignTable

```
FmtAlignTable(alignment)
```

Aligns the entire table relative to its entironment. Valid values for alignment are `center`, `right`, `left`.

FmtHeader

```
FmtHeader(fixed_width='100%', index_width=None, column_width=None, rotate_deg=0,top_
↪padding=None, no_wrap=True)
```

Creates a table with fixed-width columns.

```
:fixed_width Total width of table
:index_width Fixed width of index column
:column_width Fixed width of all other columns
:rotate_deg Value in degrees by which to rotate table header cells
:top_padding: additional vertical space above table, may be necessary when using␣
↪rotated headers
:no_wrap True/False, disables line-breaks within header cell when set to True
```

An example (NB, jypiter ignores top-padding, which otherwise works in direkt browser display and PDF output):

```
In [7]: fmt_header = tf.FmtHeader(fixed_width='20cm',index_width='30%', top_padding='3cm', rotate_deg
        table_block = Block(df, formatters=[fmt_header])
        display(HTML(table_block.render_html()))
```

```
<IPython.core.display.HTML object>
```

FmtStripeBackground

```
FmtStripeBackground(first_color=colors.LIGHT_GREY, second_color=colors.WHITE, header_
↪color=colors.WHITE,
```

Creates a repeating color patters in the background of the specified cells.

```
:first_color CSS color, for odd row numbers
:second_color CSS color, for even row numbers
:header_color CSS color applied to header row
```

FmtAddCellPadding

```
FmtAddCellPadding(left=None, right=None, top=None, bottom=None, length_unit='px')
```

Add space on sides of selected cells. #### FmtAppendTotalsRow

```
FmtAppendTotalsRow(row_name='Total', operator=OP_SUM, bold=True, background_
↪color=colors.LIGHT_GREY, font_color=None, total_columns=None)
```

Adds a line at the end of the table filled with values computed columnwise. For an example, see section [Formatting Tables with Table Formatters]

```
:row_name Label for additional row shown in index
:operator Computational operation to perform on columns, e.g. tf.OP_SUM, tf.OP_MEAN,␣
↪tf.OP_NONE
:total_columns Names of columns to apply operator to. If None, operator is applied to␣
↪all columns.
:bold True/False, applied bold font-style to cells in added row
:font_color CSS color for cell text in added row
:background_color CSS color for cell background in added row
```

FmtHideRows

```
FmtHideCells(rows=None, columns=None)
```

Hides cells in the intersection of rows and columns list. If only rows or columns is specified and the other is left None, the entire row or column is hidden. #### FmtPageBreak

```
FmtPageBreak(no_break=True, repeat_header=True)
```

Defines table behaviour around page-breaks. Please note that Webkit-based browsers (Chrome, Safari and wkhtmltopdf as well) do not handle the `repeat-header` property properly, especially when headers are rotated. This bug is reported and not resolved since 6 years. Functionality in Firefox is fine, including rotated headers.

```
:no_break True/False, avoids splitting table on page break
:repeat_header True/False, when table is split accross page, repeat header on the␣
↪next page
```

Displaying text

The following formatters modify the display of both text and numbers. #### FmtFontsize

```
FmtFontsize(fontsize, format='px')
```

Sets the font-size property of specified cells. A nice property is `vw`, which gives the font-size as percent of viewport-width. This will scale the font with the witdh of the page and is thus suited for wide tables which should still look fine (but small) when output as PDF.

FmtHighlightText

```
FmtHighlightText(bold=True, italic=True, font_color=colors.BLUE)
```

Sets various properties of character display.

```
:bold True/False, sets bold font-style
:italic True/False, sets italic font-style
:font_color CSS color, sets text color
```

FmtHighlightBackground

```
FmtHighlightBackground(color=colors.RED)
```

Sets the background color of specified cells.

FmtBold

```
FmtBold()
```

Sets font-style bold in specified cells.

Displaying numbers

The following formatters only apply to numbers. Please note that some convert the number to a string when applied.

FmtNumbers, FmtDecimals, FmtThousandSeparator, FmtPercent

E.g.

```
FmtPercent(n_decimals)
```

If cell content is a number, it is changed to a string with approriate formatting, e.g. number of decimals (FmtDecimals), with a comma as thousands separator (FmtThousandSeparator), or as percent with a trailing '%' sign(FmtPercent).

```
FmtNumbers(fmt_string)
```

is the general purpose formatting class, which accets any formatting string. For more information about formatting strings, see https://pyformat.info/

FmtMultiplyCellValue, FmtValueToMillion, FmtValueToPercent, FmtValueToBps

E.g.

```
FmtValueToPercent(suffix='%')
```

Multiplies number in cell by a given factor, thus keeping is a number. A suffix an be added to the columns header. This is useful for tables, which all contain percentage values and where a '%' sign after each value is undesireable.

```
FmtMultiplyCellValue(d, suffix)
```

is the general purpose function, multiplying by any given factor.

Heatmaps

The table formatting has a very flexible heatmap formatter. #### FmtHeatmap

```
FmtHeatmap(min_color=colors.HEATMAP_RED, max_color=colors.HEATMAP_GREEN, threshold=0.,
↪axis=None)
```

Creates a heatmap in the intersection of specified columns and rows.

```
:min_color CSS color, which is the color applied as background-color at the minimum␣
↪negative value
:max_color CSS color, which is the color applied as background-color at the maximum␣
↪positive value
:threshold specifies an interval around 0, in which no heatmapping is applied
:axis Number, either 0 (horizontal), or 1 (vertical) or None. If None, heatmap is␣
↪applied over all selected cells. If set to a number, heatmap is applied column-wise␣
↪or row-wise reprectively.
```

```
In [8]: import string
        # Create DataFrame
        df_size = 15
        index = [c for c in string.ascii_lowercase[:df_size]]
        columns = [c+c for c in string.ascii_lowercase[:df_size]]
        df = pd.DataFrame(np.random.rand(df_size,df_size), index=index, columns=columns)

        # Specify heatmap formatters
        # All values in range
        fmt_heatmap1 = tf.FmtHeatmap(rows=index[10:16],columns=columns[:5])
        # By row
        fmt_heatmap2 = tf.FmtHeatmap(rows=index[:3], axis=0, max_color=(255,0,255))
        # By column
        fmt_heatmap3 = tf.FmtHeatmap(rows=index[5:],columns=columns[10:], axis=1, max_color=(255,255,

        formatters =[fmt_heatmap1, fmt_heatmap2, fmt_heatmap3]

        # Display
        table_block = Block(df, formatters=formatters)
        display(HTML(table_block.render_html()))
```

```
<IPython.core.display.HTML object>
```

Multi-index tables

Multi-index dataframes can be expanded to sinlesimple index dataframes with special formatting applied. #### Fmt-ExpandMultiIndex

```
FmtExpandMultiIndex(total_columns=None, operator=OP_SUM, bold=True, indent_px=20,␣
↪hline_color=colors.DARK_BLUE, level_background_colors=None)
```

See example below. Can handle non-unique indices.

```
:total_columns List of columns on which to apply operator at higher index levels
:operator Computational operation to perform on columns, e.g. tf.OP_SUM, tf.OP_MEAN,␣
↪tf.OP_NONE
:bold True/False, changes higher-level font-style to bold
:index_px Indentation space per level in pixels
```

```
:hline_color CSS color, sets the color of the horizontal line separating higher-level␣
↪rows
:level_background_colors List of CSS colors with size equal to number of index-levels,
↪ background_color applied in each index-level row
```

```
In [9]: def make_multiindex_table():
            fmt = tf.FmtExpandMultiIndex()
            idx = np.array([['a', 'a', 'b', 'b'], ['aa', 'ab', 'ba', 'bb']])
            idx_tuples = list(zip(*idx))
            multi_index = pd.MultiIndex.from_tuples(idx_tuples, names=['a-level', 'aa-level'])
            columns = ['column0', 'column1', 'column2']
            data = pd.DataFrame(np.arange(12, dtype=float).reshape(4, 3), index=multi_index, columns=
            return data

        mi_df = make_multiindex_table()
        print mi_df

        fmt_multiindex = tf.FmtExpandMultiIndex(operator=tf.OP_SUM)
        table_block = Block(mi_df, formatters=[fmt_multiindex], use_default_formatters=False)
        display(HTML(table_block.render_html()))
```

```
                  column0  column1  column2
a-level aa-level
a       aa              0        1        2
        ab              3        4        5
b       ba              6        7        8
        bb              9       10       11
```

```
<IPython.core.display.HTML object>
```

Writing custom formatters

Custom formatters can either be either added to pybloqs or included with user-space code. The latter is useful for very specific formatters, which have little chance of being reused and thus do not need to sit in the code base. In general, formatters are classes that inherit from `TableFormatter` base class. The base class provides the following function hooks, which do not need to be all implemented by the new formatter. In fact, most formatters only make use of one or two function hooks. The available hooks are: * `_insert_additional_html`: Can be used to put HTML or JavaScript code in front of table. * `_modify_dataframe`: Access and potentially modify DataFrame before any other hook functions become active. * `_modify_cell_content`: Access and potentially modify cell value. Called for each cell separately. * `create_table_level_css`: Insert CSS inline style on `<table>` level * `_create_thead_level_css`: Insert CSS inline style on `<thead>` level * `_create_row_level_css`: Insert CSS inline style on `<th>` and `<tr>`level * `_create_cell_level_css`: Insert CSS inline style on `<td>` level

```
In [ ]:
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search