mplstyle Documentation

Release 0.3

Thomas Zipperl and Dennis Atabay

Sep 19, 2017

Contents

Maintainers Thomas Zipperle <thomas.zipperle@tum.de>, Dennis Atabay <dennis.atabay@tum.de>

Organization Institute for Energy Economy and Application Technology, Technische Universität München

Version 0.3

Date Sep 19, 2017

CHAPTER 1

Contents

This documentation contains the following pages:

Overview

mplstyle is a Python package, which allows **matplotlib** users to simplify the process of improving plots' quality. Quite often font, size, legend, colors and other settings should be changed for making plots look better. Such changes can be remembered and stored in your own plot style, which can automatically change the way of making plots by importing it as a usual Python toolbox. In other words, with **mplstyle** you can set plotting settings once and use created configuration many times.

The core of **mplstyle** is a **PLTbase** class contained list of methods, which change initial settings of **matplotlib**. Figure 1 illustrates these changes by plotting several trigonometric functions with and without **PLTbase**.

All changes are distributed between three settings: **color style**, **color order style** and **plt style**. Each of them according to **PLTbase** has only one way of rewriting initial corresponding **matplotlib** setting. This way is called a **default** style. Figure 2 illustrates above mentioned **PLTbase** structure.

Besides **PLTbase** there are five more classes: **PLTdatabay**, **PLTdynamis**, **PLTenfo**, **PLTewk** and **PLTtz**. Each class inherits functionality of **PLTbase** class and contains new other styles of reconfiguring initial settings. The structure of one of those classes is presented in the Figure 3.

As can be seen **PLTtz** keeps **PLTbase** styles and also contains additionals. Other four classes have exactly the same structure, but with their own additional styles. With **mplstyle** you can use already uploaded classes or create your own class.

How to Install

There are two ways of **mplstyle** installation. All of them are described below properly.

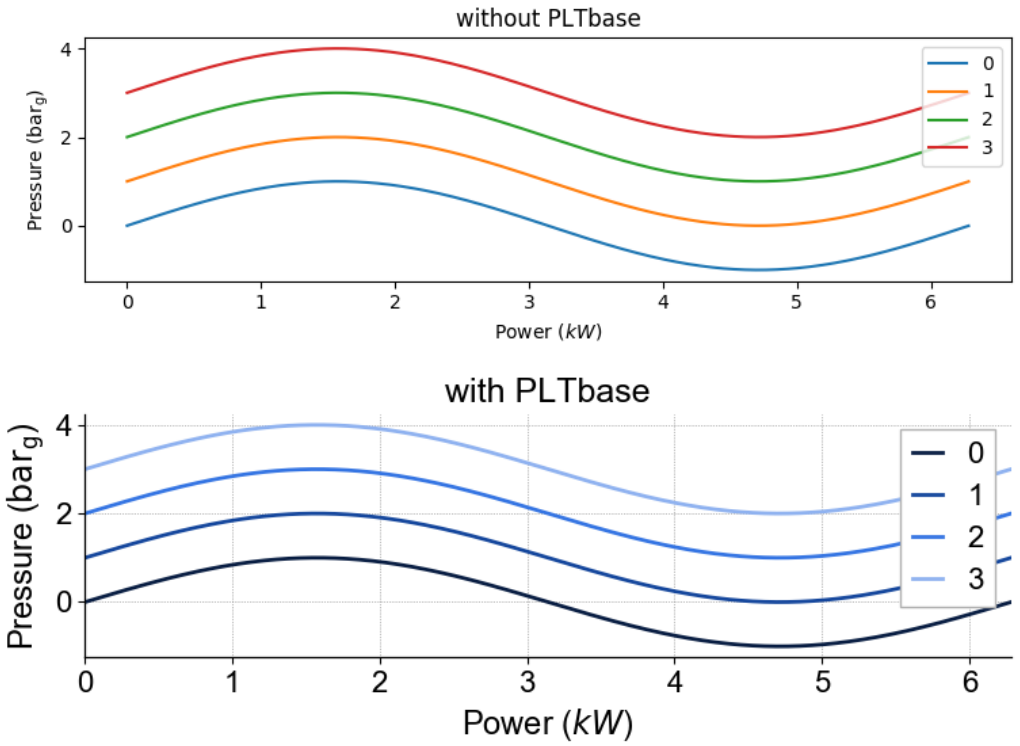

Fig. 1.1: Figure 1: Plotting changes by applying **PLTbase** class

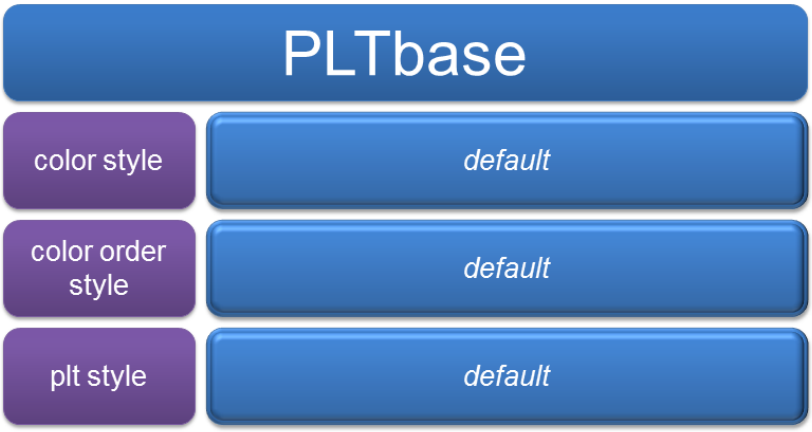

Fig. 1.2: Figure 2: **PLTbase** class structure

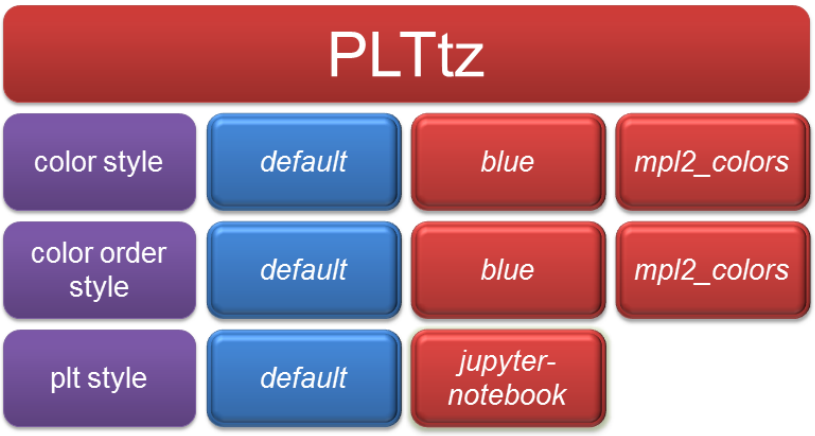

Fig. 1.3: Figure 3: **PLTtz** class structure

From Source

Open Command Promt (cmd) , paste the line written in the following and push enter. **mplstyle** package will be added to Python search path automatically.

```
pip install git+https://github.com/tzipperle/mplstyle.git@master
```

From Python

Put the following command at the beginning of your code.

```
sys.path.append('C:/.../mplstyle/')
```

It allows to use **mplstyle** for the case, when the package wasn't added to Python search path. Written in the brackets part `'C:/.../mplstyle/'` is the path to the directory, where **mplstyle** is located. This step is really necessary, if, for example, you have just downloaded or cloned (with git) this repository to a directory of your choice, and you want to run examples' codes **inside** or **outside** this directory trying to figure out how **mplstyle** is working.

How to Use

This section explains and shows how any of the classes from **mplstyle** can be used. **PLTtz** class, which was already mentioned in the *overview*, is taken as an example. Its implementation in the `trigonometric_functions.py` is explained very detailed by moving step by step through the script. Other classes are used completely similar.

trigonometric_functions.py

Several trigonometric functions are plotted fourfold in this file showing each time the change in current configuration of plot settings. All plots with chosen configuration are splitted between two figures presented in the middle and in the end of the description. Detailed comments of the script are in the following.

```
import matplotlib.pyplot as plt
import numpy as np
import os
import sys
```

Four packages are included:

- matplotlib.pyplot is a plotting library which allows present results in a diagram form quite easily;
- numpy is the fundamental package for scientific computing with Python;
- os is the module for using operating system according to its functionality in a portable way;
- sys is the module, which provides access to variables and functions used by the interpreter.

```
from mplstyle.tz import PLTtz
tz_plt=PLTtz()
```

Imports **PLTtz** class from a file `tz.py`, where this class is described as a child of **PLTbase**. Then creates an instance of the class and assigns it to the local variable **tz_plt**.

Note: When a new initialized instance of the chosen class is obtained, all three settings (**color style**, **color order style** and **plt style**) are immediately rewritten. Since this moment each setting works according to its own **default** style, kept in **PLTbase** class.

```
fig1 = plt.figure(figsize=[8, 6])
fig2 = plt.figure(figsize=[8, 6])
```

Chooses the size of two figures, where three plots will be printed.

```
#1st plot
#########
ax0 = fig1.add_subplot(211)

#setting four trigonometric functions and plotting them
x = np.arange(0, 2 * np.pi, 0.01)
for c in range(4):
        y = np.sin(x) + c
        ax0.plot(x, y, label=c)

#labeling axises, putting legend and title
ax0.set_title('default - default - default')
ax0.set_ylabel(r'Pressure ($\mathrm{bar}_{\mathrm{g}}$)')
ax0.set_xlabel(r'Power ($kW$)')
ax0.legend()
```

Makes 1st plot by choosing its location on the first figure, setting trigonometric functions and configuring additional parameters.

```
#2nd plot
#########
#changing all plot settings
tz_plt.set_style(color_style='mpl2_colors')
tz_plt.set_style(color_order_style='mpl2_colors')
tz_plt.set_style(plt_style='jupyter-notebook')
```

Changes each plot setting by **set_style()** function. Since now the initial configuration (**default - default - default**) is changed on the new one (**mpl2_colors - mpl2_colors - jupyter-notebook**).

```
ax1 = fig1.add_subplot(212)

x1 = np.arange(0, 2 * np.pi, 0.01)
for c1 in range(4):
        y1 = np.sin(x1) + c1
        ax1.plot(x1, y1, label=c1)

ax1.set_title('mpl2_colors - mpl2_colors - jupyter-notebook')
ax1.set_ylabel(r'Pressure ($\mathrm{bar}_{\mathrm{g}}$)')
ax1.set_xlabel(r'Power ($kW$)')
ax1.legend()

fig1.tight_layout()
plt.show()
```

Does similar operations for making 2nd plot. Figure 4 illustrates these two plots with their respective configurations written in the title and placed near each plot.

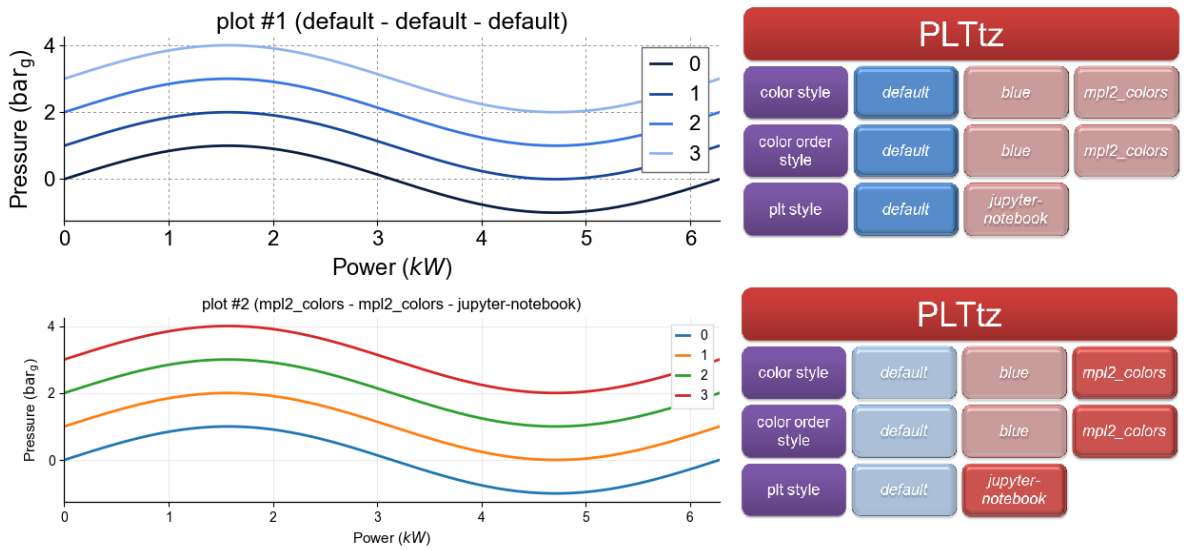

Fig. 1.4: Figure 4: Comparison of two identical plots made with initial and specifically chosen configurations

As can be seen from above written script, with **set_style()** function any from three plot settings (**color style**, **color order style** and **plt style**) can be changed quite easily. Other part of the script is described below.

```
#3rd plot
#########
#changing only plt_style setting
tz_plt.set_style(plt_style='default')
```

Starts work with 3rd plot by changing only **plt_style** setting. According to it, configuration changes only partially from old (**mpl2_colors - mpl2_colors - jupyter-notebook**) to new (**mpl2_colors - mpl2_colors - default**).

```
ax2 = fig2.add_subplot(212)

x2 = np.arange(0, 2 * np.pi, 0.01)
for c2 in range(4):
        y2 = np.sin(x2) + c2
        ax2.plot(x2, y2, label=c2)

ax2.set_title('mpl2_colors - mpl2_colors - default')
ax2.set_ylabel(r'Pressure ($\mathrm{bar}_{\mathrm{g}}$)')
ax2.set_xlabel(r'Power ($kW$)')
ax2.legend()
```

Makes 3rd plot similar to 1st and 2nd.

```
#4rth plot
#########
#changing all plot settings with one function
tz_plt.set_style('default')
```

Uses **set_style()** function once for changing all plot settings (**color style**, **color order style** and **plt style**) simultaneously to their own **default** style.

```
ax3 = fig2.add_subplot(212)

x3 = np.arange(0, 2 * np.pi, 0.01)
for c3 in range(4):
        y3 = np.sin(x3) + c3
        ax3.plot(x3, y3, label=c3)

ax3.set_title('mpl2_colors - mpl2_colors - default')
ax3.set_ylabel(r'Pressure ($\mathrm{bar}_{\mathrm{g}}$)')
ax3.set_xlabel(r'Power ($kW$)')
ax3.legend()

fig2.tight_layout()
plt.show()
```

Does similar to previous operations for making the 4th plot. 3rd and 4th plots are presented in the figure 5 with their respective configurations written in the title and placed near each plot.

Comparing 2nd and 3rd plots partial change in the configuration can be observed. From the part of script dedicated to the 4th plot can be seen, that **set_style()** function also allows to change all plot settings at one time. Only the name of the style without mentioning a specific parameter (**color style**, **color order style** and **plt style**) should be written in brackets for such change.

Note: All three settings (**color style**, **color order style** and **plt style**) can be changed at one time with **set_style()** function only to the style, which exists for all of them. The command won't work, if at least one plot setting doesn't have a style with name mentioned in brackets.

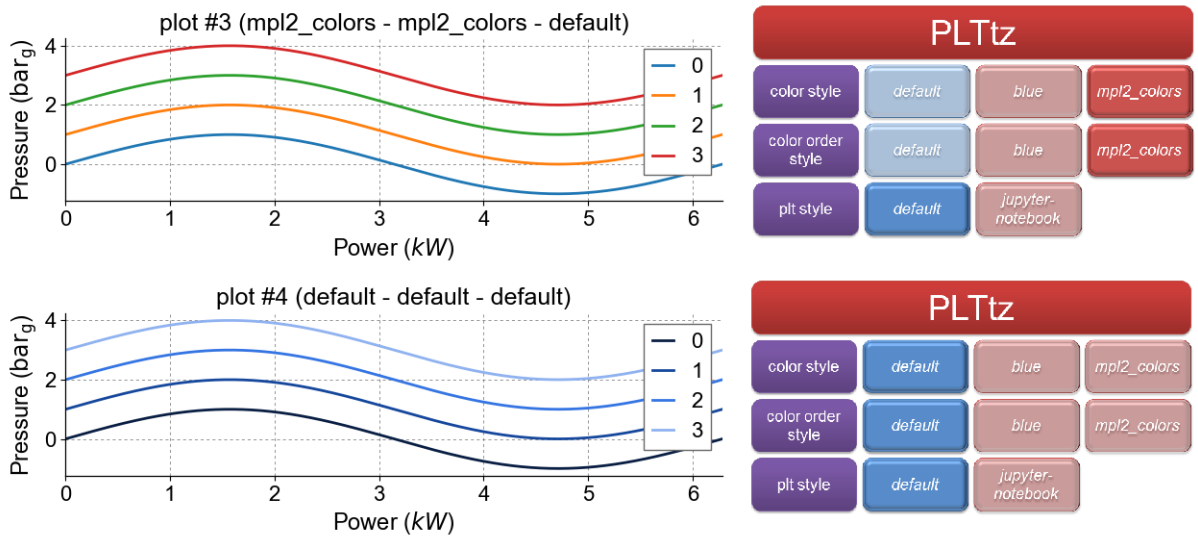

Fig. 1.5: Figure 5: Comparison of two identical plots made with two different configurations

How to Create

Current section explains the procedure of creating your own class, where all desired styles for each plot setting (**color style**, **color order style**, **plt style**) will be stored. Mentioned before **PLTtz** class is taken as an example. Its script in the `tz.py` is explained below.

tz.py

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from cycler import cycler
from .base import PLTbase
```

Four packages are included:

- matplotlib is a plotting library which allows present results in a diagram form quite easily;
- matplotlib.pyplot is a specified module of matplotlib;
- cycler is the composable style cycles;
- **base** is the file, where fundamental class **PLTbase** is described.

```
class PLTtz(PLTbase):
  #PLTtz class, child of PLTbase#

  _BLUE = 'blue'
  _JUPYTER_NOTEBOOK = 'jupyter-notebook'
  _MPL_V2 = 'mpl2_colors'
```

Gives the name to the new class mentioning in brackets the so-called “father”. Then links user-friendly names of all available styles stored inside this class with formal ones used in the script.

```
def __init__(self):
    PLTbase.__init__(self)
```

```
    available_styles = {
        'color_style': [self._BLUE, self._MPL_V2],
        'color_order_style': [self._BLUE, self._MPL_V2],
        'plt_style': [self._JUPYTER_NOTEBOOK]}

    self._add_available_styles(available_styles)
```

Function, which connects each plotting setting (**color style**, **color order style** and **plt style**) with corresponding list of available styles.

```
def _get_colors(self, style):
    if style == self._BLUE:
        return {
            'darkblue': (11, 85, 159),
            'mdarkblue': (42, 122, 185),
            'mediumblue': (83, 157, 204),
            'mlightblue': (136, 190, 220),
            'lightblue': (186, 214, 234),
            'pastelblue': (218, 232, 245),
        }
    if style == self._MPL_V2:
        return {
            'c0': '#1f77b4',
            'c1': '#ff7f0e',
            'c2': '#2ca02c',
            'c3': '#d62728',
            'c4': '#9467bd',
            'c5': '#8c564b',
            'c6': '#e377c2',
            'c7': '#7f7f7f',
            'c8': '#bcbd22',
            'c9': '#17becf'
        }
    return None
```

Function, which is responsible for **color style** plot setting. Keeps available colors for corresponding style. Style **blue** uses RGB codes for identification, **mpl2_colors** - HEX codes.

```
def _get_colors_order(self, style):
    if style == self._BLUE:
        return ['darkblue', 'mdarkblue', 'mediumblue', 'mlightblue',
                'lightblue', 'pastelblue']
    elif style == self._MPL_V2:
        return ['c0', 'c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9']
    return None
```

Current function regulates **color order style** setting. Orders corresponding colors in dependence on style.

```
def _set_plt_style(self, style, colors, prop_cycle_colors):
    if style == self._JUPYTER_NOTEBOOK:
        plt.style.use('default')
        fntsz = 12
        lw = 2
        fntcol = 'black'
        font = {'family': 'arial', 'weight': 'normal', 'size': fntsz}
```

```
        mpl.rc('font', **font)
        mpl.rc('figure', titlesize=fntsz)
        mpl.rc('legend', framealpha=None,
               edgecolor='gainsboro',
               fontsize=fntsz - 2, numpoints=1, handlelength=1,
               loc='best', frameon=True, shadow=False,
               fancybox=False)
        mpl.rcParams['text.color'] = fntcol
        mpl.rc('axes', edgecolor=fntcol, grid=True,
               xmargin=0, labelsize=fntsz - 1, titlesize=fntsz,
               linewidth=0.9)
        mpl.rcParams['axes.spines.right'] = False
        mpl.rcParams['axes.spines.top'] = False
        mpl.rc('grid', linestyle='-', color='darkgrey',
               linewidth=0.5, alpha=0.35)
        mpl.rc('lines', lw=lw, markersize=10)
        mpl.rc('xtick', color=fntcol, labelsize=fntsz - 2)
        mpl.rc('ytick', color=fntcol, labelsize=fntsz - 2)
        mpl.rcParams['axes.prop_cycle'] = cycler('color',
                                                 prop_cycle_colors)
        return True
    return False
```

These strings control **plt style** setting. All terms (font of the text, fontsize, legend, ticks on the axes, etc.), which form the view of the plot, are described here.

API

List of commands, which can be applied on initialized instance of the chosen class, is presented in the following.

`mplstyle.`**`set_style`**`(`**args*, ***kwargs*`)`

Chooses style for **color**, **color order** and **plt style**. Arguments can be entered in two ways. First one assumes, that only a name of a style in the form of string is given. In this case all settings will be changed at one time. Second case allows to change a specific setting (**color**, **color order** and **plt style**) by putting a couple **setting='style'** as an argument. Two or three arguments are also possible in this case. Concept of inputs *args and **kwargs make it possible. More detailed explanation about nature of this concept is written here.

Parameters

- **`*args`** – name of the style;
- **`**kwargs`** – couple in the form setting='style'.

`mplstyle.`**`get_colors`**`()`

Returns dictionary of colors used in the chosen **color style**.

`mplstyle.`**`get_color_order`**`()`

Returns list of colors from the chosen **color order style** in order of appearance.

`mplstyle.`**`get_available_styles`**`()`

Returns all available styles for each plotting setting (**color**, **color order** and **plt style**).

mplstyle.**get_selected_style**()

Returns chosen style for **color**, **color order** and **plt style**.

mplstyle.**get_cmap**(*colors*, *position=None*, *bit=False*)

Generates custom color maps for Matplotlib. The method allows to create a list of tuples with 8-bit (0 to 255) or arithmetic (0.0 to 1.0) RGB values for making linear color maps. Color tuple placed first characterizes the lowest value of the color bar. The last tuple represents the the highest value.

Parameters

- **colors** – list of RGB tuples with 8-bit (0 to 255) or arithmetic (0 to 1), default: arithmetic;
- **position** – list from 0 to 1, which dictates the location for each color;
- **bit** – boolean, default: False (arithmetic), True (RGB);

Returns **cmap** - a color map with equally spaced colors.

Example1 cmap = mplstyle.get_cmap(colors=[(255, 0, 0), (0, 157, 0)], bit=True)

Example2 cmap = mplstyle.get_cmap([(1, 1, 1), (0.5, 0, 0)], position=[0, 1]))

mplstyle.**add_zbild**(*ax*, *x*, *y*, *text*, *tum=True*, *fontsize=10*, *color='grey'*)

Sets ZBild number as a text in the chart.

Parameters

- **ax** – instance of matplotlib axes;
- **x** – float for position (xmin=0, xmax=1);
- **y** – float for position (ymin=0, ymax=1);
- **text** – string for the text;
- **tum** – optional boolean for copyright, default: (tum=True);
- **fontsize** – optional float for font size, default: (10);
- **color** – optional string for mpl color, default: (grey).

CHAPTER 2

Dependencies (Python)

- numpy for mathematical operations
- matplotlib for making plots
- cycler

Python Module Index

m

Index