pylustrator Documentation

Release 1.3.0

Richard Gerum

Feb 28, 2023

CONTENTS

1 Installation **3**

2 Usage **5**

3 Note **9**

4 Citing Pylustrator **11**

5 License **13**

5.1 Styling Figures 13

5.2 Composing Figures 14

5.3 API 18

Index **19**

Pylustrator is a software to prepare your figures for publication in a reproducible way. This means you receive a figure representing your data and alongside a generated code file that can exactly reproduce the figure as you put them in the publication, without the need to readjust things in external programs.

Pylustrator offers an interactive interface to find the best way to present your data in a figure for publication. Added formatting an styling can be saved by automatically generated code. To compose multiple figures to panels, pylustrator can compose different subfigures to a single figure.

CHAPTER

ONE

INSTALLATION

Just get pylustrator over the pip installation:

```
pip install pylustrator
```

The package depends on:

numpy, matplotlib, pyqt5, qtpy, qtawesome, scikit-image, natsort

CHAPTER

TWO

USAGE

Using pylustrator is very easy and does not require substantial modifications to your code. Just add

```
import pylustrator
pylustrator.start()
```

before creating your first figure in your code. When calling `plt.show()` the plot will be displayed in a pylustrator window.

You can test pylustrator with the following example code example_pylustrator.py:

```
# import matplotlib and numpy as usual
import matplotlib.pyplot as plt
import numpy as np

# now import pylustrator
import pylustrator

from icecream import install
install()
import matplotlib.pyplot as plt
import numpy as np

# now import pylustrator
import pylustrator

# activate pylustrator
pylustrator.start()

if 1:
    # build plots as you normally would
    np.random.seed(1)
    t = np.arange(0.0, 2, 0.001)
    y = 2 * np.sin(np.pi * t)
    a, b = np.random.normal(loc=(5., 3.), scale=(2., 4.), size=(100,2)).T
    b += a

    fig = plt.figure(1)
    plt.clf()
    fig.text(0.5, 0.5, "new", transform=plt.figure(1).transFigure)
    plt.subplot(131)
    plt.plot(t, y)
```

(continues on next page)

(continued from previous page)

```
    plt.text(0.5, 0.5, "new", transform=plt.figure(1).axes[0].transAxes)
    plt.xlabel("bla")
    plt.ylabel("foo")
    plt.subplot(132)
    plt.plot(a, b, "o")

    plt.subplot(133)
    plt.bar(0, np.mean(a), label="a")
    plt.bar(1, np.mean(b), label="b")
    #plt.legend()
    plt.xticks
    plt.figure(1).axes[0].set(position=[0.125, 0.11, 0.2279, 0.77], xlim=(-0.09995, 3.),␣
↪xlabel='blaa', xticks=[0., 1., 2., 3.], xticklabels=['0', '1', '2', '3'], ylim=(-3., 3.
↪), ylabel='fooo', yticks=[-3., -2., -1., 0., 1., 2., 3.], yticklabels=['3', '2', '1',
↪'0', '1', '2', '3'])

    plt.figure(1).axes[0].spines[['right', 'top']].set_visible(False)

    #% start: automatic generated code from pylustrator
    plt.figure(1).ax_dict = {ax.get_label(): ax for ax in plt.figure(1).axes}
    import matplotlib as mpl
    getattr(plt.figure(1), '_pylustrator_init', lambda: ...)()
    plt.figure(1).axes[0].grid(True)
    plt.figure(1).axes[0].set(position=[0.1033, 0.1945, 0.2279, 0.77], xlim=(-0.09995,␣
↪10.))
    plt.figure(1).axes[0].texts[0].set(position=(0.2854, 0.5627))
    plt.figure(1).axes[1].spines[['right', 'top']].set_visible(False)
    plt.figure(1).axes[2].set(ylim=(0., 8.82))
    plt.figure(1).axes[2].spines[['right', 'top']].set_visible(False)
    #% end: automatic generated code from pylustrator
    plt.show()
    exit()
# activate pylustrator
pylustrator.start()

# build plots as you normally would
np.random.seed(1)
t = np.arange(0.0, 2, 0.001)
y = 2 * np.sin(np.pi * t)
a, b = np.random.normal(loc=(5., 3.), scale=(2., 4.), size=(100,2)).T
b += a

plt.figure(1)
sub_fig = plt.gcf().add_subfigure(plt.GridSpec(2, 1)[0, 0])
sub_fig2 = sub_fig.add_subfigure(plt.GridSpec(1, 2)[0, 0])
sub_fig2.add_subplot(121)
plt.plot(t, y)
plt.text(0.5, 0.5, "neu!")

sub_fig.add_subplot(122)
plt.plot(a, b, "o")

```

(continues on next page)

(continued from previous page)

```
sub_fig = plt.gcf().add_subfigure(plt.GridSpec(2, 1)[1, 0])
sub_fig.add_subplot(111)
plt.bar(0, np.mean(a), label="A")
plt.bar(1, np.mean(b), label="B")


#plt.figure(1).axes[0].set(position=[0.213022, 0.498889, 0.227941, 0.381111], xlim=[0.7,␣
↪1448], xticks=[0.1, 0.001], xticklabels=["A", "B"])
#for name in dir(plt.figure(1).axes[0]):
#    if name.startswith("set_"):
#        print(name)

plt.show()

```

Saving by pressing `Ctrl+S` or confirming to save when closing the window will add some lines of code at the end of your python script (before your `plt.show()`) that defines these changes:

```
#% start: automatic generated code from pylustrator
plt.figure(1).set_size_inches(8.000000/2.54, 8.000000/2.54, forward=True)
plt.figure(1).axes[0].set_position([0.191879, 0.148168, 0.798133, 0.742010])
plt.figure(1).axes[0].set_xlabel("data x")
plt.figure(1).axes[0].set_ylabel("data y")
plt.figure(1).axes[1].set_position([0.375743, 0.603616, 0.339534, 0.248372])
plt.figure(1).axes[1].set_xlabel("data x")
plt.figure(1).axes[1].set_ylabel("data y")
plt.figure(1).axes[1].set_ylim(-40.0, 90.0)
#% end: automatic generated code from pylustrator
```

Note: Because pylustrator can optionally save changes you've made in the GUI to update your source code, it cannot be used from a shell. To use pylustrator, call it directly from a python file and use the command line to execute.

Note: In case you import matplotlib.pyplot to the global namespace (e.g. *from matplotlib.pyplot import* *), pylustrator has to be started before this import to be able to overload the *show* command.

Also using the *show* from the *pylab* import does not work. And is anyways discouraged, see https://matplotlib.org/stable/api/index.html?highlight=pylab#module-pylab

The good thing is that this generated code is plain matplotlib code, so it will still work when you remove pylustrator from your code! This is especially useful if you want to distribute your code and do not want to require pylustrator as a dependency.

Can styling plots be any easier?

CHAPTER THREE

NOTE

If you encounter any bugs or unexpected behaviour, you are encouraged to report a bug in our Github bugtracker.

CHAPTER

FOUR

CITING PYLUSTRATOR

If you use Pylustrator for your publications I would highly appreciate it if you cite the Pylustrator:

- Gerum, R., (2020). **pylustrator: code generation for reproducible figures for publication**. Journal of Open Source Software, 5(51), 1989. doi:10.21105/joss.01989

CHAPTER

FIVE

LICENSE

Pylustrator is released under the GPLv3 license. The generated output code of Pylustrator can be freely used according to the MIT license, but as it relys on Matplotlib also the Matplotlib License has to be taken into account.

5.1 Styling Figures

5.1.1 Opening

To open the pylustrator editor to style a figure, you just have to call the function *pylustrator.start()* before any figure is created in your code.

```
import pylustrator
pylustrator.start()
```

This will overload the commands *plt.figure()* and *plt.show()*. *plt.figure()* (which is often indirectly called when creating plots) now initializes a figure in a GUI window and *plt.show()* then shows this GUI window.

In the GUI window elements can be dragged around using a click and drag with the left mouse button. If you want to cycle though different elements on the same spot double click on the same position multiple times. To zoom in in the plot window use *strg+mousewheel* and you can pan the figure with holding the middle mouse button.

To select multiple elements hold shift while clicking on multiple elements.

5.1.2 Saving

To save the figure press *ctrl+s* or select *File->Save*. This will generate code that corresponds to the changes you made to the figure and paste it into your script file or your jupyter notebook cell. The code will be pasted directly over the *plt.show()* command that started the editor or, if there already is a generated code block for this figure, it will replace the existing code block.

5.1.3 Increasing Performance

Often plots with lots of elements can slow down the performance of pylustrator as with every edit, the whole plot is rerendered. To circumvent this problem, pylustrator offers a function to calculate a rasterized representation (e.g. pixel data) of the contents of each axes and only display the pixel data instead of rendering the vector data with every draw.

It can be activated with the button "rasterize". It can be clicked again to update the rasterisation or deactivated with a click on the button "derasterize" next to it.

5.1.4 Color editor

Pylustrator comes with a powerful color editor which allows to test different color configurations for your figure easily. On the right hand side of the window you see a list of all currently used colors. You can right click on any color to open a color choosed dialog. You can also directly edit the color using the html notation (e.g. #FF0000) provided on the button. You can drag and drop colors to different slots to test different configurations for your figure.

The field below allows to copy and paste color lists from different sources. For example using color palette generators on the internet, e.g. https://medialab.github.io/iwanthue/.

Additionally, if you generate plot lines with colors from a colormap, pylustrator can recognize that and allow you to choose different colormaps for the set of plot lines.

5.1.5 Tick Editor

To edit the ticks of an axes click on . There, a windows opens that allows to set major and minor ticks every line in the edit window corresponds to one tick. Texts are directly interpreted as float values if possible and the text used as tick label, e.g. you can but a tick at 5.0 with the text "5" (e.g. formated without decimal point). To specify an exponent, it is also possible to write e.g. 5*10^2 (to put a tick at 500 with the label $5 \cdot 10^2$). If the label cannot be directly writen as a number, add the label after the number enclosed in tick marks e.g. 5 "start", to add a tick at position 5 with the label "start".

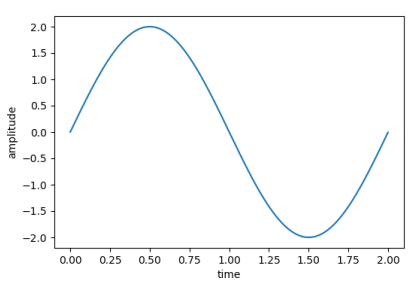
5.2 Composing Figures

Pylustrator can also be used to compose panels of different subplots using the function the function *pylustrator.load()*.

5.2.1 Example

Suppose we have two plot files that we want to include in our figure:

Plot 1



Listing 1: plot1.py

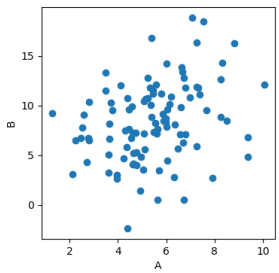
```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2, 0.001)
y = 2 * np.sin(np.pi * t)

plt.figure(0, (6, 4))
plt.plot(t, y)
plt.xlabel("time")
plt.ylabel("amplitude")
plt.savefig("plot1.png")

plt.show()
```

Plot 2



Listing 2: plot2.py

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

np.random.seed(1)
a, b = np.random.normal(loc=(5., 3.),
                        scale=(2., 4.),
                        size=(100, 2)).T
b += a

plt.figure(0, (4, 4))
plt.plot(a, b, "o")
plt.xlabel("A")
plt.ylabel("B")
plt.savefig("plot2.png")
plt.show()
```

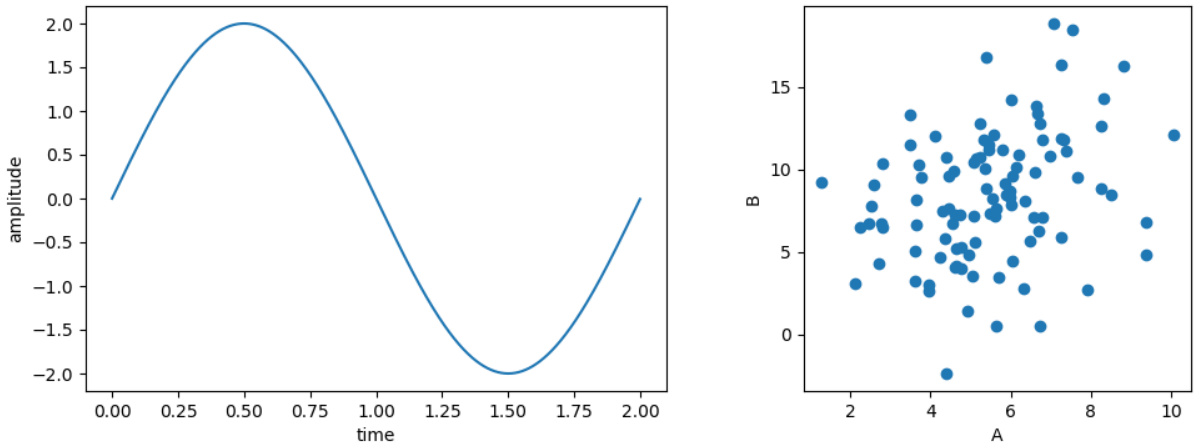
Now we can create a script that generates the composite figure:

Listing 3: figure1.py

```
import matplotlib.pyplot as plt
import pylustrator

pylustrator.load("plot1.py")
pylustrator.load("plot2.py", offset=[1, 0])

plt.show()
```



The subfigures can both contain also multiple axes (e.g. subplots) or be previously styled with pylustrator. The composite figure can also be styled with pylustrator to finialize the figure.

Note: Please note that the code of the target script will be executed, to only load script files from trusted sources. This also holds true when loading scripts from a cached pickle representation, see *Caching*.

5.2.2 Supported Formats

With *pylustrator.load()* different types of inputs can be used to add to a composite figure.

Python Files

If the input is a ".py" file then the file is compiled and executed. All the figure elements that are generated from the python file are then inserted into the target figure.

Note: All your plotting code should be in the main part of the script, everything hidden behind a *if __name__ == "__main__":* will be ignored.

Image Files

If the input is any image format that can be opened with *plt.imread()* then the file is loaded and its content is displayed in a separate axis using *plt.imshow()*. To scale the image, pylustrator uses the dpi settings of the figure. You can also provide the *dpi* keyword to *pylustrator.load()* to scale the image.

Svg Files

If the input file is a *svg* file, then pylustrator tries to generate matplotlib patches to display the content of the svg file in a new axes.

> **Warning:** The svg specification is broader than the patches supported by matplotlib. Therefore, gradients, filters, fill patterns and masks cannot be supported. Also svg offers some advances positioning features for text (e.g. letter-spacing, word-spacing) which are difficult to match in matplotlib. There might be some more differences in details in the implementations. If you thing something can be addressed in matplotlib, you can report it in the bugtracker.

5.2.3 Positioning

Loaded subfigures can be positioned using the *offset* keyword. Offsets can have different units.

The default unit is to interpret it as a percentage of the current figure size. Note, that when calling multiple imports, the figure size changes between calls. Therefore, a 2x2 grid can be achieved with relative offsets as follows:

```
pylustrator.load("plot1.py")
pylustrator.load("plot2.png", offset=[1, 0])
pylustrator.load("plot3.jpg", offset=[0, 1])
pylustrator.load("plot4.svg", offset=[0.5, 0.5])
```

The offset can also be specified in cm or in in. Therefore, the third entry of the tuple can be the string "cm" or "in". For example:

```
pylustrator.load("plot1.py")
pylustrator.load("plot2.png", offset=[2, 1, "cm"])
pylustrator.load("plot3.jpg", offset=[0.3, 0.9, "in"])
```

5.2.4 Caching

Pylustrator also offers the possibility to cache the figures generated by a script file. Therefore, the figure is pickled after it has been created from the script and saved next to the script. If the script is newer (last modified timestamp) as the pickle file with the cached figure, the script is executed again. The caching behavior can be disabled with the keyword *cache=False*.

5.3 API

The API of pylustrator is kept quite simple. Most interaction with the pylustrator package is by using the interactive interface.

`pylustrator.start(use_global_variable_names=False, use_exception_silencer=False, disable_save=False)`

This will overload the commands `plt.figure()` and `plt.show()`. If a figure is created after this command was called (directly or indirectly), a GUI window will be initialized that allows to interactively manipulate the figure and generate code in the calling script to define these changes. The window will be shown when `plt.show()` is called.

See also *Styling Figures*.

Parameters

use_global_variable_names (*bool, optional*) – if used, try to find global variables that reference a figure and use them in the generated code.

`pylustrator.load(filename: str, figure: Optional[Figure] = None, offset: Optional[list] = None, dpi: Optional[int] = None, cache: bool = False, label: str = '')`

Add contents to the current figure from the file defined by filename. It can be either a python script defining a figure, an image (filename or directly the numpy array), or an svg file.

See also *Composing Figures*.

Parameters

- **filename** (*str*) – The file to load. Can point to a python script file, an image file or an svg file.
- **figure** (*Figure, optional*) – The figure where to add the loaded file. Defaults to the current figure.
- **offset** (*list, optional*) – The offset where to import the file. The first two parts define the x and y position and the third part defines the units to use. Default is "%", a percentage of the current figure size. It can also be "cm" or "in".
- **cache** (*bool, optional*) – Whether to try to cache the figure generated from the file. Only for python files. This option is experimental and may not be stable.

INDEX

L

`load()` (*in module pylustrator*), 18

S

`start()` (*in module pylustrator*), 18