ridgeplot

Release 0.1.25

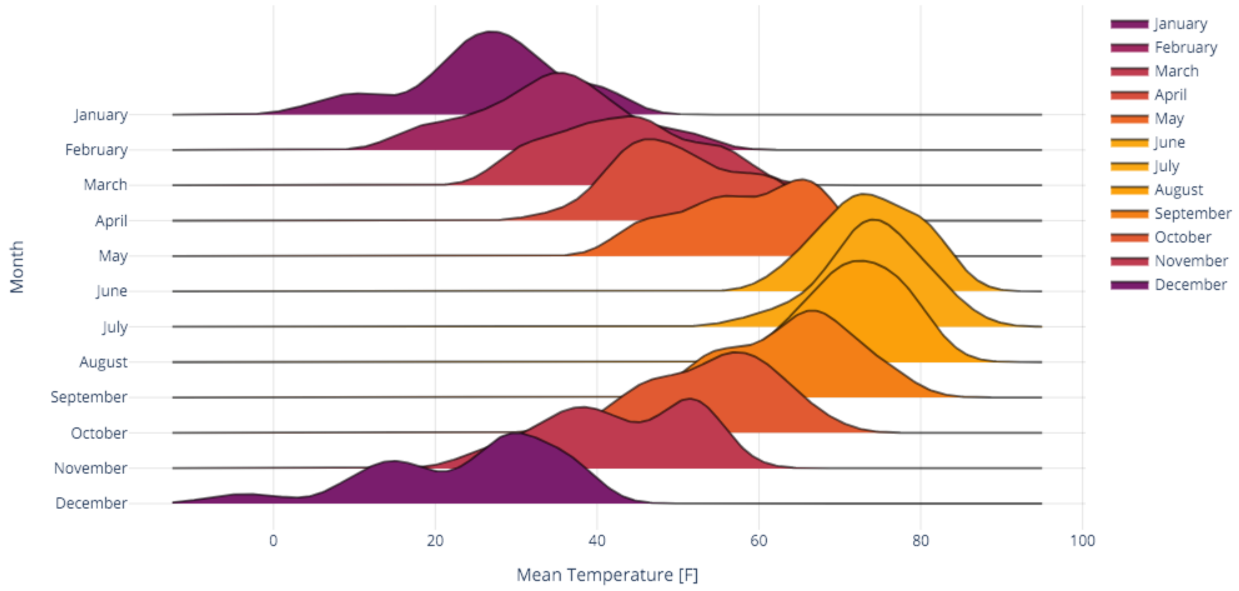
Tomas Pereira de Vasconcelos

Apr 23, 2024

GETTING STARTED

ridgeplot: beautiful ridgeline plots in Python

`ridgeplot` is a Python package that provides a simple interface for plotting beautiful and interactive *ridgeline plots* within the extensive Plotly ecosystem.



`ridgeplot` can be installed and updated from PyPi using pip:

```
pip install -U ridgeplot
```

For more information, see the *installation guide*.

Take a look at the *getting started guide*, which provides a quick introduction to the `ridgeplot` library.

For those in a hurry, here's a very basic example on how to quickly get started with the `ridgeplot()` function.

```
import numpy as np
from ridgeplot import ridgeplot

my_samples = [np.random.normal(n / 1.2, size=600) for n in range(8, 0, -1)]
fig = ridgeplot(samples=my_samples)
fig.update_layout(height=450, width=800)
fig.show()
```

CHAPTER

ONE

INSTALLING FROM PYPI

`ridgeplot` can be installed and updated from PyPi using pip:

```
pip install -U ridgeplot
```

CHAPTER TWO

INSTALLING FROM SOURCE

The source code for this project is hosted on GitHub at: https://github.com/tpvasconcelos/ridgeplot

Take a look at the *contributing guide* for instructions on how to build from the git source. Further, refer to the instructions on *creating a development environment* if you wish to create a local development environment, or wish to contribute to the project.

CHAPTER THREE

DEPENDENCIES

We try to keep the number of dependencies to a minimum and only use common and well-established libraries in the scientific python ecosystem. Currently, we only depend on the following 3 Python packages:

- plotly - The interactive graphing backend that powers `ridgeplot`
- statsmodels - Used for Kernel Density Estimation (KDE)
- numpy - Supporting library for multidimensional array manipulations

CHAPTER

FOUR

GETTING STARTED

This page provides a quick introduction to the `ridgeplot` library, showcasing some of its features and providing a few practical examples. All examples use the *`ridgeplot.ridgeplot()`* function, which is the main entry point to the library. For more information on the available options, take a look at the *reference page*.

4.1 Basic example

This basic example shows how you can quickly get started with a simple call to the *`ridgeplot()`* function.

```
import numpy as np
from ridgeplot import ridgeplot

my_samples = [np.random.normal(n / 1.2, size=600) for n in range(8, 0, -1)]
fig = ridgeplot(samples=my_samples)
fig.update_layout(height=450, width=800)
fig.show()
```

4.2 Flexible configuration

In this example, we will try to replicate the first ridgeline plot in this *from Data to Viz* post. The example in the post was created using the *"Perception of Probability Words"* dataset (see *`load_probly()`*) and the popular ggridges R package. In the end, we will see how the `ridgeplot` Python library can be used to create a (nearly) identical plot, thanks to its extensive configuration options.

```
import numpy as np
from ridgeplot import ridgeplot
from ridgeplot.datasets import load_probly

# Load the probly dataset
df = load_probly()

# Let's grab the subset of columns used in the example
column_names = [
    "Almost Certainly",
    "Very Good Chance",
    "We Believe",
    "Likely",
    "About Even",
```

(continues on next page)

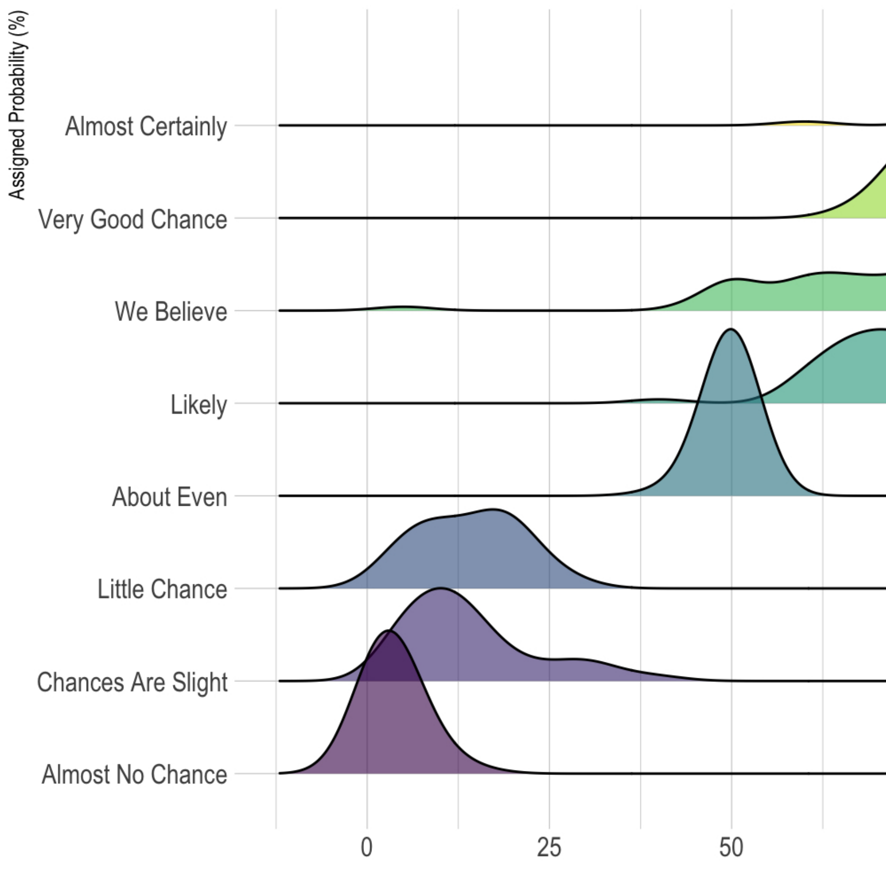
(continued from previous page)

```
    "Little Chance",
    "Chances Are Slight",
    "Almost No Chance",
]
df = df[column_names]

# Not only does 'ridgeplot(...)' come configured with sensible defaults
# but is also fully configurable to your own style and preference!
fig = ridgeplot(
    samples=df.to_numpy().T,
    bandwidth=4,
    kde_points=np.linspace(-12.5, 112.5, 500),
    colorscale="viridis",
    colormode="row-index",
    coloralpha=0.65,
    labels=column_names,
    linewidth=2,
    spacing=5 / 9,
)

# And you can still update and extend the final
# Plotly Figure using standard Plotly methods
fig.update_layout(
    height=760,
    width=900,
    font_size=16,
    plot_bgcolor="white",
    xaxis_tickvals=[-12.5, 0, 12.5, 25, 37.5, 50, 62.5, 75, 87.5, 100, 112.5],
    xaxis_ticktext=["", "0", "", "25", "", "50", "", "75", "", "100", ""],
    xaxis_gridcolor="rgba(0, 0, 0, 0.1)",
    yaxis_gridcolor="rgba(0, 0, 0, 0.1)",
    yaxis_title="Assigned Probability (%)",
    showlegend=False,
)

# Show us the work!
fig.show()
```

Output

The resulting ridgeline plot generated by the code above:

Target/reference image



The target reference from the *from Data to Viz* post:

4.3 More traces

In this example, we will dive a bit deeper into the `samples` parameter and see how we can be used to plot multiple traces per row in a ridgeline plot.

4.3.1 Final result

For the ones in a hurry, we are including the entire final code-block and resulting plot already in this section. It is here also to serve as a reference for the rest of the section and to demonstrate what the goal of this example is. That said, throughout the rest of this section, we will dive a bit deeper into the `samples` parameter and understand how flexible it is.

Code

```python
import numpy as np
from ridgeplot import ridgeplot
from ridgeplot.datasets import load_lincoln_weather

# Load test data
df = load_lincoln_weather()

# Transform the data into a 3D (ragged) array format of
# daily min and max temperature samples per month
months = df.index.month_name().unique()
samples = [
    [
        df[df.index.month_name() == month]["Min Temperature [F]"],
        df[df.index.month_name() == month]["Max Temperature [F]"],
    ]
    for month in months
]

# And finish by styling it up to your liking!
fig = ridgeplot(
    samples=samples,
    labels=months,
    coloralpha=0.98,
    bandwidth=4,
    kde_points=np.linspace(-25, 110, 400),
    spacing=0.33,
    linewidth=2,
)
fig.update_layout(
    title="Minimum and maximum daily temperatures in Lincoln, NE (2016)",
    height=650,
    width=950,
    font_size=14,
    plot_bgcolor="rgb(245, 245, 245)",
    xaxis_gridcolor="white",
    yaxis_gridcolor="white",
    xaxis_gridwidth=2,
    yaxis_title="Month",
    xaxis_title="Temperature [F]",
    showlegend=False,
)
fig.show()
```

Output

4.3.2 Step-by-step

Let's start by loading the *"Lincoln Weather"* test dataset (see `load_lincoln_weather()`).

```
>>> from ridgeplot.datasets import load_lincoln_weather
>>> df = load_lincoln_weather()
>>> df[["Min Temperature [F]", "Max Temperature [F]"]].head()
            Min Temperature [F]  Max Temperature [F]
CST
2016-01-01                   11                   37
2016-01-02                    5                   41
2016-01-03                    8                   37
2016-01-04                    4                   30
2016-01-05                   19                   38
```

The goal will be to plot the KDEs for the minimum and maximum daily temperatures for each month of 2016 (i.e. the year covered by the dataset).

```
>>> months = df.index.month_name().unique()
>>> months.to_list()
['January', 'February', 'March', 'April', 'May', 'June', 'July',
 'August', 'September', 'October', 'November', 'December']
```

The `samples` argument in the `ridgeplot()` function expects a 3D array of shape (R, T_r, S_t), where R is the number of rows, T_r is the number of traces per row, and S_t is the number of samples per trace, with:

Dimension values	Description
$R = 12$	One row per month.
$T_r = 2$ (for all rows $r \in R$)	Two traces per row (one for the minimum temperatures and one for the maximum temperatures).
$S_t \in \{29, 30, 31\}$	One sample per day of the month, where different months have different number of days.

We can create this array using a simple list comprehension, where each element of the list is a list of two arrays, one for the minimum temperatures and one for the maximum temperatures samples, for each month:

```
samples = [
    [
        df[df.index.month_name() == month]["Min Temperature [F]"],
        df[df.index.month_name() == month]["Max Temperature [F]"],
    ]
    for month in months
]
```

Note: For other use cases (like in the two previous examples), you could use a numpy ndarray to represent the samples. However, since different months have different number of days, we need to use a data container that can hold arrays of different lengths along the same dimension. Irregular arrays like this one are called ragged arrays. There are many different ways you can represent irregular arrays in Python. In this specific example, we used a list of lists of pandas Series. However,`ridgeplot` is designed to handle any object that implements the `Collection[Collection[Collection[Numeric]]]` protocol (i.e. any numeric 3D ragged array).

Finally, we can pass the `samples` list to the *`ridgeplot()`* function and specify any other arguments we want to customize the plot, like adjusting the KDE's bandwidth, the vertical spacing between rows, etc.

```
fig = ridgeplot(
    samples=samples,
    labels=months,
    coloralpha=0.98,
    bandwidth=4,
    kde_points=np.linspace(-25, 110, 400),
    spacing=0.33,
    linewidth=2,
)

fig.update_layout(
    title="Minimum and maximum daily temperatures in Lincoln, NE (2016)",
    height=650,
    width=950,
    font_size=14,
    plot_bgcolor="rgb(245, 245, 245)",
    xaxis_gridcolor="white",
    yaxis_gridcolor="white",
    xaxis_gridwidth=2,
    yaxis_title="Month",
    xaxis_title="Temperature [F]",
    showlegend=False,
)
fig.show()
```

CHAPTER

FIVE

API REFERENCE

ridgeplot.ridgeplot	Return an interactive ridgeline (Plotly) `Figure`.

5.1 ridgeplot.ridgeplot

`ridgeplot.`**`ridgeplot`**(*samples=None*, *densities=None*, *kernel='gau'*, *bandwidth='normal_reference'*, *kde_points=500*, *colorscale='plasma'*, *colormode='mean-minmax'*, *coloralpha=None*, *labels=None*, *linewidth=1.0*, *spacing=0.5*, *show_annotations=<MISSING>*, *show_yticklabels=True*, *xpad=0.05*)

Return an interactive ridgeline (Plotly) `Figure`.

Note: You must pass either *`samples`* or *`densities`* to this function, but not both. See descriptions below for more details.

Parameters

- **samples** (*`Samples`* or *`ShallowSamples`*, *optional*) – If `samples` data is specified, Kernel Density Estimation (KDE) will be computed. See *`kernel`*, *`bandwidth`*, and *`kde_points`* for more details and KDE configuration options. The `samples` argument should be an array of shape (R, T_r, S_t). Note that we support irregular (ragged) arrays, where:
  - R is the number of rows in the plot
  - T_r is the number of traces per row, where each row $r \in R$ can have a different number of traces.
  - S_t is the number of samples per trace, where each trace $t \in T_r$ can also have a different number of samples.

  The KDE will be performed over the sample values (S_t) for all traces. After the KDE, the resulting array will be a (4D) *`densities`* array with shape $(R, T_r, P_t, 2)$ (see below for more details).
- **densities** (*`Densities`* or *`ShallowDensities`*, *optional*) – If a `densities` array is specified, the KDE step will be skipped and all associated arguments ignored. Each density array should have shape $(R, T_r, P_t, 2)$ (4D). Just like the *`samples`* argument, we also support irregular (ragged) `densities` arrays, where:
  - R is the number of rows in the plot

  - T_r is the number of traces per row, where each row $r \in R$ can have a different number of traces.
  - P_t is the number of points per trace, where each trace $t \in T_r$ can also have a different number of points.
  - 2 is the number of coordinates per point (x and y)
- **kernel** (`str`) – The Kernel to be used during Kernel Density Estimation. The default is a Gaussian Kernel (`"gau"`). Choices are:
  - `"biw"` for biweight
  - `"cos"` for cosine
  - `"epa"` for Epanechnikov
  - `"gau"` for Gaussian.
  - `"tri"` for triangular
  - `"triw"` for triweight
  - `"uni"` for uniform
- **bandwidth** (*`KDEBandwidth`*) – The bandwidth to use during Kernel Density Estimation. The default is `"normal_reference"`. Choices are:
  - `"scott"` - 1.059 * A * nobs ** (-1/5.), where A is `min(std(x),IQR/1.34)`
  - `"silverman"` - .9 * A * nobs ** (-1/5.), where A is `min(std(x),IQR/1.34)`
  - `"normal_reference"` - C * A * nobs ** (-1/5.), where C is calculated from the kernel. Equivalent (up to 2 dp) to the `"scott"` bandwidth for gaussian kernels. See bandwidths.py.
  - If a float is given, its value is used as the bandwidth.
  - If a callable is given, it's return value is used. The callable should take exactly two parameters, i.e., `fn(x, kern)`, and return a float, where:
    - `x`: the clipped input data
    - `kern`: the kernel instance used
- **kde_points** (*`KDEPoints`*) – This argument controls the points at which KDE is computed. If an `int` value is passed (default=500), the densities will be evaluated at `kde_points` evenly spaced points between the min and max of each set of samples. Optionally, you can also pass a custom 1D numerical array, which will be used for all traces.
- **colorscale** (`str` or *`ColorScale`*) – Any valid Plotly color-scale or a str with a valid named color-scale. Use *`list_all_colorscale_names()`* to see which names are available or check out Plotly's built-in color-scales.
- **colormode** (`Colormode`) – This argument controls the logic for choosing the color filling of each ridgeline trace. Each option provides a different method for calculating the *`colorscale`* midpoint of each trace. The default is mode is `"mean-means"`. Choices are:
  - `"row-index"` - uses the row's index. e.g. if the ridgeplot has 3 rows of traces, then the midpoints will be `[[0, ...], [0.5, ...], [1, ...]]`.
  - `"trace-index"` - uses the trace's index. e.g. if the ridgeplot has a total of 3 traces (across all rows), then the midpoints will be 0, 0.5, and 1, respectively.

  - `"mean-minmax"` - uses the min-max normalized (weighted) mean of each density to calculate the midpoints. The normalization min and max values are the minimum and maximum x-values from all densities, respectively.
  - `"mean-means"` - uses the min-max normalized (weighted) mean of each density to calculate the midpoints. The normalization min and max values are the minimum and maximum mean values from all densities, respectively.
- **coloralpha** (`float`, *optional*) – If None (default), this argument will be ignored and the transparency values of the specifies color-scale will remain untouched. Otherwise, if a float value is passed, it will be used to overwrite the transparency (alpha) of the color-scale's colors.
- **labels** (*`LabelsArray`* or *`ShallowLabelsArray`*, *optional*) – A list of string labels for each trace. The default value is None, which will result in auto-generated labels of form "Trace n". If, instead, a list of labels is specified, it must be of the same size/length as the number of traces.
- **linewidth** (`float`) – The traces' line width (in px).
- **spacing** (`float`) – The vertical spacing between density traces, which is defined in units of the highest distribution (i.e. the maximum y-value).
- **show_annotations** (`bool`) – Whether to show the tick labels on the y-axis. The default is True.

  Deprecated since version 0.1.21: Use *`show_yticklabels`* instead.
- **show_yticklabels** (`bool`) – Whether to show the tick labels on the y-axis. The default is True.

  Added in version 0.1.21: Replaces the deprecated *`show_annotations`* argument.
- **xpad** (`float`) – Specifies the extra padding to use on the x-axis. It is defined in units of the range between the minimum and maximum x-values from all distributions.

Returns
A Plotly `Figure` with a ridgeline plot. You can further customize this figure to your liking (e.g. using the `update_layout()` method).

Return type
`plotly.graph_objects.Figure`

Raises
`ValueError` – If both *`samples`* and *`densities`* are specified, or if neither of them is specified. i.e. you may only specify one of them.

5.2 Color utilities

`ridgeplot.list_all_colorscale_names`	Get a list with all available colorscale names.

5.2.1 ridgeplot.list_all_colorscale_names

`ridgeplot.`**`list_all_colorscale_names`**`()`

Get a list with all available colorscale names.

Added in version 0.1.21: Replaces the deprecated `get_all_colorscale_names()`.

Returns
A list with all available colorscale names.

Return type
`list[str]`

5.3 Data loading utilities

`ridgeplot.datasets.load_probly`	Load a version of the "Perception of Probability Words" (a.k.a., *"probly"*) dataset.
`ridgeplot.datasets.load_lincoln_weather`	Load the "Weather in Lincoln, Nebraska in 2016" dataset.

5.3.1 ridgeplot.datasets.load_probly

`ridgeplot.datasets.`**`load_probly`**`(`*`version='zonination'`*`)`

Load a version of the “Perception of Probability Words” (a.k.a., *“probly”*) dataset.

Parameters
version (`{'zonination', 'wadefagen', 'illinois'}`, *default*: `'zonination'`) – The version of the dataset to load. Each version is slightly different and originates from different surveys. See the *Notes* section for more details on each version.

Returns
A dataframe containing a *probly* dataset.

Return type
`pandas.DataFrame`

Notes

Sherman Kent, a CIA analyst, first published his work on the perception of probabilistic words in 1964[1]. This exercise has been repeated several times since then. This function provides three different versions of the dataset, each originating from a different survey. Valid options for the `version` parameter are:

`"zonination"`
This is perhaps the most popular version of the dataset and originates from a survey conducted by the Reddit user /u/zonination.

[1] Sherman Kent. (1964). *“Words of estimative probability”*. https://www.cia.gov/static/Words-of-Estimative-Probability.pdf

Creator	@zonination
Source	https://raw.githubusercontent.com/zonination/perceptions/51207062aa173777264d3acce0131e1e2456d966/probly.csv
Accessed on	2023-06-24

`"wadefagen"`

This version of the dataset originates from a blogpost by Wade Fagen-Ulmschneider from the University of Illinois[2]. It is based on a survey conducted on different social media platforms.

Creator	Wade Fagen-Ulmschneider (@wadefagen)
Source	https://raw.githubusercontent.com/wadefagen/datasets/7e752937b72edc3126e3dd17e3cd97eb727af8f9/Perception-of-Probability-Words/survey-results.csv
Accessed on	2023-06-24

`"illinois"`

This version of the dataset originates from a survey of primarily undergraduate students conducted at The University of Illinois[3].

Creator	University of Illinois
Source	https://waf.cs.illinois.edu/discovery/words.csv
Accessed on	2023-06-24

References

5.3.2 ridgeplot.datasets.load_lincoln_weather

`ridgeplot.datasets.`**`load_lincoln_weather`**`()`

Load the “Weather in Lincoln, Nebraska in 2016” dataset.

Returns

A dataframe containing the “Lincoln Weather” dataset.

Return type

`pandas.DataFrame`

[2] Wade Fagen-Ulmschneider. *“Perception of Probability Words”*. https://waf.cs.illinois.edu/visualizations/Perception-of-Probability-Words/

[3] University of Illinois. *“Perception of Probability Words Dataset”*. https://discovery.cs.illinois.edu/dataset/words/

Notes

The version of the dataset included in this package is the same version included in the *ggridges* R package[1]. The dataset contains weather information from Lincoln, Nebraska (2016). The original data was taken from a blogpost by Austin Wehrwein in 2017[2].

Source	https://raw.githubusercontent.com/wilkelab/ggridges/543a092c601b92d7b62e630fb34d038f54485a29/data-raw/lincoln-weather.csv
Accessed on	2023-08-07

References

5.4 Internals

5.4.1 ridgeplot._colors

Color utilities.

`ridgeplot._colors.ColorScale`

A colorscale is an iterable of tuples of two elements:

0. the first element (a *scale value*) is a float bounded to the interval `[0, 1]`
1. the second element (a *color*) is a string representation of a color parsable by Plotly

For instance, the Viridis colorscale would be defined as

```
>>> get_colorscale("viridis")
((0.0, 'rgb(68, 1, 84)'),
 (0.1111111111111111, 'rgb(72, 40, 120)'),
 (0.2222222222222222, 'rgb(62, 73, 137)'),
 (0.3333333333333333, 'rgb(49, 104, 142)'),
 (0.4444444444444444, 'rgb(38, 130, 142)'),
 (0.5555555555555556, 'rgb(31, 158, 137)'),
 (0.6666666666666666, 'rgb(53, 183, 121)'),
 (0.7777777777777777, 'rgb(110, 206, 88)'),
 (0.8888888888888888, 'rgb(181, 222, 43)'),
 (1.0, 'rgb(253, 231, 37)'))
```

alias of `Iterable[Tuple[float, str]]`

`ridgeplot._colors._Color`

A color can be represented as an rgb(a) or hex string or a tuple of `(r, g, b)` values.

alias of `str | Tuple[float, float, float]`

`ridgeplot._colors._colormap_loader()`

[1] ggridges. *"Weather in Lincoln, Nebraska in 2016"*. https://wilkelab.org/ggridges/reference/lincoln_weather.html
[2] Austin Wehrwein. *"Plot inspiration via FiveThirtyEight"*. https://austinwehrwein.com/data-visualization/plot-inspiration-via-fivethirtyeight/

`ridgeplot._colors.`**`validate_colorscale`**(*colorscale*)

Validate the structure, scale values, and colors of a colorscale.

Adapted from `_plotly_utils.colors.validate_colorscale`.

`ridgeplot._colors.`**`_any_to_rgb`**(*color*)

Convert any color to an rgb string.

Parameters
color – A color. This can be a tuple of `(r, g, b)` values, a hex string, or an rgb string.

Returns
An rgb string.

Return type
`str`

Raises

- `TypeError` – If `color` is not a tuple or a string.
- `ValueError` – If `color` is a string that does not represent a hex or rgb color.

`ridgeplot._colors.`**`list_all_colorscale_names`**()

Get a list with all available colorscale names.

Added in version 0.1.21: Replaces the deprecated *`get_all_colorscale_names()`*.

Returns
A list with all available colorscale names.

Return type
`list[str]`

`ridgeplot._colors.`**`get_all_colorscale_names`**()

Get a tuple with all available colorscale names.

Deprecated since version 0.1.21: Use *`list_all_colorscale_names()`* instead.

Returns
A tuple with all available colorscale names.

Return type
`tuple[str, ]`

`ridgeplot._colors.`**`get_colorscale`**(*name*)

Get a colorscale by name.

Parameters
name – The colorscale name. This argument is case-insensitive. For instance, `"YlOrRd"` and `"ylorrd"` map to the same colorscale. Colorscale names ending in `_r` represent a *reversed* colorscale.

Returns
A colorscale.

Return type
`ColorScale`

Raises
`ValueError` – If an unknown name is provided

`ridgeplot._colors.`**`get_color`**`(`*colorscale*, *midpoint*`)`

Get a color from a colorscale at a given midpoint.

Given a colorscale, it interpolates the expected color at a given midpoint, on a scale from 0 to 1.

`ridgeplot._colors.`**`apply_alpha`**`(`*color*, *alpha*`)`

5.4.2 ridgeplot._figure_factory

Ridgeline plot figure factory logic.

`ridgeplot._figure_factory.`**`LabelsArray`**

A *`LabelsArray`* represents the labels of traces in a ridgeplot.

Example

```
>>> labels_array: LabelsArray = [
...     ["trace 1", "trace 2", "trace 3"],
...     ["trace 4", "trace 5"],
... ]
```

alias of `Collection`[`Collection`[`str`]]

`ridgeplot._figure_factory.`**`ShallowLabelsArray`**

Shallow type for *`LabelsArray`*.

Example

```
>>> labels_array: ShallowLabelsArray = ["trace 1", "trace 2", "trace 3"]
```

alias of `Collection`[`str`]

`ridgeplot._figure_factory.`**`ColorsArray`**

A *`ColorsArray`* represents the colors of traces in a ridgeplot.

Example

```
>>> colors_array: ColorsArray = [
...     ["red", "blue", "green"],
...     ["orange", "purple"],
... ]
```

alias of `Collection`[`Collection`[`str`]]

`ridgeplot._figure_factory.`**`ShallowColorsArray`**

Shallow type for *`ColorsArray`*.

Example

```
>>> colors_array: ShallowColorsArray = ["red", "blue", "green"]
```

alias of `Collection`[`str`]

`ridgeplot._figure_factory.`**`MidpointsArray`**

A *`MidpointsArray`* represents the midpoints of colorscales in a ridgeplot.

Example

```
>>> midpoints_array: MidpointsArray = [
...     [0.2, 0.5, 1],
...     [0.3, 0.7],
... ]
```

alias of `Collection`[`Collection`[`float`]]

`ridgeplot._figure_factory.`**`Colormode`**

The *`ridgeplot.ridgeplot.colormode`* argument in *`ridgeplot.ridgeplot()`*.

alias of `Literal`['row-index', 'trace-index', 'mean-minmax', 'mean-means']

`ridgeplot._figure_factory.`**`_D3HF = '.7'`**

Default (d3-format) format for floats in hover labels.

After trying to read through the plotly.py source code, I couldn't find a simple way to replicate the default hover format using the d3-format syntax in Plotly's 'hovertemplate' parameter. The closest I got was by using the string below, but it's not quite the same… (see '.7~r' as well)

`ridgeplot._figure_factory.`**`_DEFAULT_HOVERTEMPLATE = '(%{x:.7}, %{customdata[0]:.7})<br><extra>%{fullData.name}</extra>'`**

Default `hovertemplate` for density traces.

See *`ridgeplot._figure_factory.RidgePlotFigureFactory.draw_density_trace()`*.

`ridgeplot._figure_factory.`**`get_xy_extrema`**(*densities*)

Get the global x-y extrema (x_min, x_max, y_min, y_max) over all *`DensityTrace`*s in the *`Densities`* array.

Parameters
densities – A *`Densities`* array.

Returns
A tuple of the form (x_min, x_max, y_min, y_max).

Return type
`Tuple[Numeric`, *`Numeric`*, *`Numeric`*, `Numeric]`

Examples

```
>>> get_xy_extrema(
...     [
...         [
...             [(0, 0), (1, 1), (2, 2), (3, 3)],
...             [(0, 0), (1, 1), (2, 2)],
...             [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)],
```

(continues on next page)

(continued from previous page)

```
...         ],
...         [
...             [(-2, 2), (-1, 1), (0, 1)],
...             [(2, 2), (3, 1), (4, 1)],
...         ],
...     ]
... )
(-2, 4, 0, 4)
```

`ridgeplot._figure_factory._mul`(*a*, *b*)

Multiply two tuples element-wise.

class `ridgeplot._figure_factory.RidgePlotFigureFactory`(*densities*, *colorscale*, *coloralpha*, *colormode*, *labels*, *linewidth*, *spacing*, *show_yticklabels*, *xpad*)

Bases: `object`

Refer to *`ridgeplot.ridgeplot()`*.

property colormode_maps: dict[str, Callable[[], MidpointsArray]]

`draw_base`(*x*, *y_shifted*)

Draw the base for a density trace.

Adds an invisible trace at constant y that will serve as the fill-limit for the corresponding density trace.

`draw_density_trace`(*x*, *y*, *y_shifted*, *label*, *color*)

Draw a density trace.

Adds a density 'trace' to the Figure. The `fill="tonexty"` option fills the trace until the previously drawn trace (see *`draw_base()`*). This is why the base trace must be drawn first.

`update_layout`(*y_ticks*)

Update figure's layout.

`_compute_midpoints_row_index`()

colormode='row-index'

Uses the row's index. e.g. if the ridgeplot has 3 rows of traces, then the midpoints will be [[1, …], [0.5, …], [0, …]].

`_compute_midpoints_trace_index`()

colormode='trace-index'

Uses the trace's index. e.g. if the ridgeplot has a total of 3 traces (across all rows), then the midpoints will be 0, 0.5, and 1, respectively.

_compute_midpoints_mean_minmax()

colormode='mean-minmax'

Uses the min-max normalized (weighted) mean of each density to calculate the midpoints. The normalization min and max values are the minimum and maximum x-values from all densities, respectively.

_compute_midpoints_mean_means()

colormode='mean-means'

Uses the min-max normalized (weighted) mean of each density to calculate the midpoints. The normalization min and max values are the minimum and maximum mean values from all densities, respectively.

pre_compute_colors()

make_figure()

5.4.3 ridgeplot._kde

Kernel density estimation (KDE) utilities.

ridgeplot._kde.**KDEPoints**

The *ridgeplot.ridgeplot.kde_points* parameter.

alias of `int | Collection[int | np.integer[Any] | float | np.floating[Any]]`

ridgeplot._kde.**KDEBandwidth**

The *ridgeplot.ridgeplot.bandwidth* parameter.

alias of `str | float | Callable[[Collection[int | np.integer[Any] | float | np.floating[Any]], CustomKernel], float]`

ridgeplot._kde.**estimate_density_trace**(*trace_samples*, *points*, *kernel*, *bandwidth*)

Estimates a density trace from a set of samples.

For a given set of sample values, computes the kernel densities (KDE) at the given points.

ridgeplot._kde.**_validate_densities**(*x*, *y*, *kernel*)

ridgeplot._kde.**estimate_densities**(*samples*, *points*, *kernel*, *bandwidth*)

Perform KDE for a set of samples.

5.4.4 ridgeplot._missing

Missing sentinel class.

`class ridgeplot._missing._Missing(value)`

Bases: `Enum`

A singleton class that represents a missing value.

This implementation was mainly inspired by the discussions in #236 (typing), #40397 (pandas), and on the current implementation of pandas._libs.lib._NoDefault.

For reference, here are other discussions and implementations that were also considered:

- PEP 484 - Support for singleton types in unions
- #7844 (numpy)
- #16241 (numpy)
- numpy._globals._NoValueType
- dataclasses.MISSING

`MISSING = 'MISSING'`

5.4.5 ridgeplot._types

Miscellaneous types, type aliases, and related utilities.

`ridgeplot._types.CollectionL1`

A `TypeAlias` for a 1-level-deep `Collection`.

Example

```
>>> c1 = [1, 2, 3]
```

alias of `Collection`[_T]

`ridgeplot._types.CollectionL2`

A `TypeAlias` for a 2-level-deep `Collection`.

Example

```
>>> c2 = [[1, 2, 3], [4, 5, 6]]
```

alias of `Collection`[`Collection`[_T]]

`ridgeplot._types.CollectionL3`

A `TypeAlias` for a 3-level-deep `Collection`.

Example

```
>>> c3 = [
...     [[1, 2], [3, 4]],
...     [[5, 6], [7, 8]],
... ]
```

alias of `Collection[Collection[Collection[_T]]]`

`ridgeplot._types.`**`Float`**

A `TypeAlias` for float types.

alias of `float` | `np.floating[Any]`

`ridgeplot._types.`**`Int`**

A `TypeAlias` for a int types.

alias of `int` | `np.integer[Any]`

`ridgeplot._types.`**`Numeric`**

A `TypeAlias` for *numeric* types.

alias of `int` | `np.integer[Any]` | `float` | `np.floating[Any]`

`class` `ridgeplot._types.`**`NumericT`**

A `TypeVar` variable bound to `Numeric` types.

alias of TypeVar('NumericT', bound=`int` | `np.integer[Any]` | `float` | `np.floating[Any]`)

`ridgeplot._types.`**`_is_numeric`**(*obj: int | integer[Any] | float | floating[Any]*) → Literal[True]

`ridgeplot._types.`**`_is_numeric`**(*obj: Any*) → bool

Check if the given object is a `Numeric` type.

`ridgeplot._types.`**`XYCoordinate`**

A 2D (x, y) coordinate, represented as a `tuple` of two `Numeric` values.

Example

```
>>> xy_coord = (1, 2)
```

alias of `Tuple[NumericT, NumericT]`

`ridgeplot._types.`**`DensityTrace`**

A 2D line/trace represented as a collection of (x, y) coordinates (i.e. `XYCoordinate`s).

These are equivalent:

- `DensityTrace`
- `CollectionL1[XYCoordinate]`
- `Collection[Tuple[Numeric, Numeric]]`

By convention, the x values should be non-repeating and increasing. For instance, the following is a valid 2D line trace:

Code example

```
>>> density_trace = [(0, 0), (1, 1), (2, 2), (3, 1), (4, 0)]
```

Graphical representation

alias of `Collection`[`Tuple`[`Any`, `Any`]]

`ridgeplot._types.`**`DensitiesRow`**

A *`DensitiesRow`* represents a set of *`DensityTrace`*s that are to be plotted on a given row of a ridgeplot.

These are equivalent:

- `DensitiesRow`
- `CollectionL2[XYCoordinate]`
- `Collection[Collection[Tuple[Numeric, Numeric]]]`

Example

Code example

```
>>> densities_row = [
...     [(0, 0), (1, 1), (2, 0)],                 # Trace 1
...     [(1, 0), (2, 1), (3, 2), (4, 1)],         # Trace 2
...     [(3, 0), (4, 1), (5, 2), (6, 1), (7, 0)], # Trace 3
... ]
```

Graphical representation

---

alias of `Collection`[`Collection`[`Tuple`[`Any`, `Any`]]]

`ridgeplot._types.`**`Densities`**

The *`Densities`* type represents the entire collection of traces that are to be plotted on a ridgeplot.

In a ridgeplot, several traces can be plotted on different rows. Each row is represented by a *`DensitiesRow`* object which, in turn, is a collection of *`DensityTrace`*s. Therefore, the *`Densities`* type is a collection of *`DensitiesRow`*s.

These are equivalent:

- `Densities`
- `CollectionL1[DensitiesRow]`
- `CollectionL3[XYCoordinate]`
- `Collection[Collection[Collection[Tuple[Numeric, Numeric]]]]`

Example

Code example

```
>>> densities = [
...     [                                              # Row 1
...         [(0, 0), (1, 1), (2, 0)],                  # Trace 1
...         [(1, 0), (2, 1), (3, 2), (4, 1)],          # Trace 2
...         [(3, 0), (4, 1), (5, 2), (6, 1), (7, 0)], # Trace 3
...     ],
...     [                                              # Row 2
...         [(-2, 0), (-1, 1), (0, 0)],                # Trace 5
...         [(0, 0), (1, 1), (2, 1), (3, 0)],          # Trace 6
...     ],
... ]
```

Graphical representation

---

alias of `Collection`[`Collection`[`Collection`[`Tuple`[`Any`, `Any`]]]]

`ridgeplot._types.`**`ShallowDensities`**

Shallow type for *`Densities`* where each row of the ridgeplot contains only a single trace.

These are equivalent:

- `Densities`
- `CollectionL1[DensityTrace]`
- `CollectionL2[XYCoordinate]`
- `Collection[Collection[Tuple[Numeric, Numeric]]]`

Example

Code example

```
>>> shallow_densities = [
...     [(0, 0), (1, 1), (2, 0)], # Trace 1
...     [(1, 0), (2, 1), (3, 0)], # Trace 2
...     [(2, 0), (3, 1), (4, 0)], # Trace 3
... ]
```

Graphical representation

---

alias of `Collection`[`Collection`[`Tuple`[`Any`, `Any`]]]

`ridgeplot._types.`**`is_shallow_densities`**(*obj: Collection[Collection[Tuple[Any, Any]]]*) → Literal[True]

`ridgeplot._types.`**`is_shallow_densities`**(*obj: Any*) → bool

Check if the given object is a *`ShallowDensities`* type.

ridgeplot._types.**SamplesTrace**

A *SamplesTrace* is a collection of numeric values representing a set of samples from which a *DensityTrace* can be estimated via KDE.

Example

Code example

```
>>> samples_trace = [0, 1, 1, 2, 2, 2, 3, 3, 4]
```

Graphical representation

alias of `Collection[int | np.integer[Any] | float | np.floating[Any]]`

ridgeplot._types.**SamplesRow**

A *SamplesRow* represents a set of *SamplesTrace*s that are to be plotted on a given row of a ridgeplot.

i.e. a *SamplesRow* is a collection of *SamplesTrace*s and can be converted into a *DensitiesRow* by applying KDE to each trace.

Example

Code example

```
>>> samples_row = [
...     [0, 1, 1, 2, 2, 2, 3, 3, 4], # Trace 1
...     [1, 2, 2, 3, 3, 3, 4, 4, 5], # Trace 2
... ]
```

Graphical representation

alias of `Collection[Collection[int | np.integer[Any] | float | np.floating[Any]]]`

ridgeplot._types.**Samples**

The *Samples* type represents the entire collection of samples that are to be plotted on a ridgeplot.

It is a collection of *SamplesRow* objects. Each row is represented by a *SamplesRow* type which, in turn, is a collection of *SamplesTrace*s which can be converted into *DensityTrace* 's by applying a kernel density estimation algorithm.

Therefore, the *Samples* type can be converted into a *Densities* type by applying a kernel density estimation (KDE) algorithm to each trace.

See *Densities* for more details.

Example

Code example

```
>>> samples = [
...     [                                 # Row 1
...         [0, 1, 1, 2, 2, 2, 3, 3, 4], # Trace 1
...         [1, 2, 2, 3, 3, 3, 4, 4, 5], # Trace 2
...     ],
...     [                                 # Row 2
...         [2, 3, 3, 4, 4, 4, 5, 5, 6], # Trace 3
...         [3, 4, 4, 5, 5, 5, 6, 6, 7], # Trace 4
...     ],
... ]
```

Graphical representation

---

alias of `Collection[Collection[Collection[int | np.integer[Any] | float | np.floating[Any]]]]`

`ridgeplot._types.`**`ShallowSamples`**

Shallow type for *`Samples`* where each row of the ridgeplot contains only a single trace.

Example

Code example

```
>>> shallow_samples = [
...     [0, 1, 1, 2, 2, 2, 3, 3, 4], # Trace 1
...     [1, 2, 2, 3, 3, 3, 4, 4, 5], # Trace 2
... ]
```

Graphical representation

---

alias of `Collection[Collection[int | np.integer[Any] | float | np.floating[Any]]]`

`ridgeplot._types.`**`is_shallow_samples`**(*obj: Collection[Collection[int | integer[Any] | float | floating[Any]]]*) → Literal[True]

`ridgeplot._types.`**`is_shallow_samples`**(*obj: Any*) → bool

Check if the given object is a *`ShallowSamples`* type.

`ridgeplot._types.`**`is_flat_str_collection`**(*obj: Collection[str]*) → Literal[True]

`ridgeplot._types.`**`is_flat_str_collection`**(*obj: Any*) → bool

Check if the given object is a `CollectionL1[str]` type but not a string itself.

`ridgeplot._types.`**`nest_shallow_collection`**(*shallow_collection*)

Convert a *shallow* collection type into a *deep* collection type.

This function should really only be used in the `ridgeplot._ridgeplot` module to normalize user input.

5.4.6 ridgeplot._utils

Miscellaneous utilities and helper functions.

`ridgeplot._utils.`**`normalise_min_max`**(*val*, *min_*, *max_*)

`ridgeplot._utils.`**`get_collection_array_shape`**(*arr*)

Return the shape of a `Collection` array.

Parameters

arr – The `Collection` array.

Returns

The elements of the shape tuple give the lengths of the corresponding array dimensions. If the length of a dimension is variable, the corresponding element is a `set` of the variable lengths. Otherwise, (if the length of a dimension is fixed), the corresponding element is an `int`.

Return type

`Tuple[Union[int, Set[int]], ]`

Examples

```
>>> get_collection_array_shape([1, 2, 3])
(3,)
```

```
>>> get_collection_array_shape([[1, 2, 3], [4, 5]])
(2, {2, 3})
```

```
>>> get_collection_array_shape(
...     [
...         [
...             [1, 2, 3], [4, 5]
...         ],
...         [
...             [6, 7, 8, 9],
...         ],
...     ]
... )
(2, {1, 2}, {2, 3, 4})
```

```
>>> get_collection_array_shape(
...     [
...         [
...             [1], [2, 3], [4, 5, 6],
...         ],
...         [
...             [7, 8, 9, 10, 11],
...         ],
...     ]
... )
(2, {1, 3}, {1, 2, 3, 5})
```

```
>>> get_collection_array_shape(
...     [
...         [
...             [(0, 0), (1, 1), (2, 2), (3, 3)],
...             [(0, 0), (1, 1), (2, 2)],
...             [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)],
...         ],
...         [
...             [(-2, 2), (-1, 1), (0, 1)],
...             [(2, 2), (3, 1), (4, 1)],
...         ],
...     ]
... )
(2, {2, 3}, {3, 4, 5}, 2)
```

```
>>> get_collection_array_shape(
...     [
...         [
...             ["a", "b", "c", "d"], ["e", "f"],
...         ],
...         [
...             ["h", "i", "j", "k", "l"],
...         ],
...     ]
... )
(2, {1, 2}, {2, 4, 5})
```

class ridgeplot._utils.**LazyMapping**(*loader*)

Bases: Mapping[~_KT, ~_VT]

A lazy mapping that loads its contents only when first needed.

Parameters
loader – A callable that returns a mapping.

Examples

```
>>> def my_io_loader() -> dict[str, int]:
...     print("Loading...")
...     return {"a": 1, "b": 2}
...
>>> lazy_mapping = LazyMapping(my_io_loader)
>>> lazy_mapping
Loading...
{'a': 1, 'b': 2}
```

_loader

_inner_mapping: Mapping**[_KT, _VT] | None**

property _mapping: Mapping**[_KT, _VT]**

`_abc_impl = <_abc._abc_data object>`

CHAPTER SIX

ALTERNATIVES

- `plotly` - from examples/galery
- `seaborn` - from examples/galery
- `bokeh` - from examples/galery
- `matplotlib` - from blogpost
- `joypy` - Ridgeplot library using a `matplotlib` backend

CHAPTER SEVEN

RELEASE NOTES

This document outlines the list of changes to ridgeplot between each release. For full details, see the commit logs.

7.1 Unreleased changes

- …

7.1.1 CI/CD

- Move all CI/CD utilities to the `cicd_utils/` directory (#186)
- Publish to PyPi as a Trusted Publisher (#187)
- Add `check-jsonschema` pre-commit hooks and define `timeout-minutes` for all GitHub workflows (#187)

7.2 0.1.25

This release contains a number of improvements to the docs, API reference, CI/CD logic (incl. official support for Python 3.12), and other minor internal changes.

7.2.1 Documentation

- Misc documentation improvements (#180)
- Move changelog to `./docs/reference/changelog.md` (#180)

7.2.2 Internals

- Migrate from `setup.cfg` from `pyproject.toml` (#176)
- Use `importlib.resources` to load data assets from within the package - to be PEP-302 compliant (#176)
- Enforce "strict" mypy mode (mostly improved type annotations for generic types) (#177)

7.2.3 CI/CD

- Add support for Python 3.12 (#182)

7.3 0.1.24

7.3.1 Breaking changes

- Dropped support for Python 3.7. (#154)

7.3.2 Features

- Add hoverinfo by default to the Plotly traces. (#174)

7.3.3 Documentation

- Use the `{raw} html :file:  _static/charts/<PLOT-ID>.html` directive to load the interactive Plotly graphs in the generated Sphinx docs. The generated HTML artefacts only include a `<div>` wrapper block now and the plotly.min.js is now vendored and automatically loaded via the `html_js_files` Sphinx config. (#132)
- Small adjustments to the example plots in the documentation. (#132)
- Reformat markdown files, removing all line breaks. (#132)

7.3.4 Internals

- Define a `ridgeplot._missing.MISSING` sentinel object for internal use (this replaces the multiple module-level `_MISSING = object()` sentinels). (#154)
- Add an internal `extras/` directory to place helper modules and packages used in different CI tasks. (#154 and #161)

7.3.5 CI/CD

- Replace `isort`, `flake8`, and `pyupgrade` with `ruff`. (#131)
- Add regression tests for the figure artifacts generated by the examples in `_ridgeplot_examples`. (#154)
- Remove the Python locked dependency files. (#163)

7.4 0.1.23

- Fix the references to the interactive Plotly IFrames (#129)

7.5 0.1.22

7.5.1 Deprecations

- The `colormode='index'` value has been deprecated in favor of `colormode='row-index'`, which provides the same functionality but is more explicit and allows to distinguish between the `'row-index'` and `'trace-index'` modes. (#114)
- The `show_annotations` argument has been deprecated in favor of `show_yticklabels`. (#114)
- The `get_all_colorscale_names()` function has been deprecated in favor of `list_all_colorscale_names()`. (#114)

7.5.2 Features

- Add functionality to allow plotting of multiple traces per row. (#114)
- Add `ridgeplot.datasets.load_lincoln_weather()` helper function to load the "Lincoln Weather" toy dataset. (#114)
- Add more versions of the *probly* dataset (`"wadefagen"` and `"illinois"`). (#114)
- Add support for Python 3.11.

7.5.3 Documentation

- Major update to the documentation, including more examples, interactive plots, script to generate the HTML and WebP images from the example scripts, improved API reference, and more. (#114)

7.5.4 Internal

- Remove `mdformat` from the automated CI checks. It can still be triggered manually. (#114)
- Improved type annotations and type checking. (#114)

7.6 0.1.21

7.6.1 Features

- Add `ridgeplot.datasets.load_probly()` helper function to load the `probly` toy dataset. The `probly.csv` file is now included in the package under `ridgeplot/datasets/data/`. (#80)

7.6.2 Documentation

- Change to numpydoc style docstrings. (#81)
- Add a robots.txt to the docs site. (#81)
- Auto-generate a site map for the docs site using `sphinx_sitemap`. (#81)
- Change the sphinx theme to `furo`. (#81)
- Improve the internal documentation and some of these internals to the API reference. (#81)

7.6.3 Internal

- Fixed and improved some type annotations, including the introduction of `ridgeplot._types` module for type aliases such as `Numeric` and `NestedNumericSequence`. (#80)
- Add the `blacken-docs` pre-commit hook and add the `pep8-naming`, `flake8-pytest-style`, `flake8-simplify`, `flake8-implicit-str-concat`, `flake8-bugbear`, `flake8-rst-docstrings`, `flake8-rst-docstrings`, etc… plugins to the `flake8` pre-commit hook. (#81)
- Cleanup and improve some type annotations. (#81)
- Update deprecated `set-output` commands (GitHub Actions) (#87)

7.7 0.1.17

- Automate the release process. See .github/workflows/release.yaml, which issues a new GitHub release whenever a new git tag is pushed to the main branch by extracting the release notes from the changelog.
- Fix automated release process to PyPI. (#27)

7.8 0.1.16

- Upgrade project structure, improve testing and CI checks, and start basic Sphinx docs. (#21)
- Implement `LazyMapping` helper to allow `ridgeplot._colors.PLOTLY_COLORSCALES` to lazy-load from `colors.json` (#20)

7.9 0.1.14

- Remove `named_colorscales` from public API (#18)

7.10 0.1.13

- Add tests for example scripts (#14)

7.11 0.1.12

7.11.1 Internal

- Update and standardise CI steps (#6)

7.11.2 Documentation

- Publish official contribution guidelines (`CONTRIBUTING.md`) (#8)
- Publish an official Code of Conduct (`CODE_OF_CONDUCT.md`) (#7)
- Publish an official release/change log (`CHANGES.md`) (#6)

7.12 0.1.11

- `colors.json` was missing from the final distributions (#2)

7.13 0.1.0

- Initial release!

CHAPTER

EIGHT

CONTRIBUTING

Thank you for your interest in contributing to ridgeplot!

The contribution process for ridgeplot should start with filing a GitHub issue. We define three main categories of issues, and each category has its own GitHub issue template

- Feature requests
- Bug reports
- Documentation fixes

After the implementation strategy has been agreed on by a ridgeplot contributor, the next step is to introduce your changes as a pull request (see *Pull Request Workflow*) against the ridgeplot repository. Once your pull request is merged, your changes will be automatically included in the next ridgeplot release. Every change should be listed in the ridgeplot *Changelog*.

The following is a set of (slightly opinionated) rules and general guidelines for contributing to ridgeplot. Emphasis on **guidelines**, not *rules*. Use your best judgment, and feel free to propose changes to this document in a pull request.

8.1 Development environment

Here are some guidelines for setting up your development environment. Most of the steps have been abstracted away using the make build automation tool. Feel free to peak inside Makefile at any time to see exactly what is being run, and in which order.

First, you will need to clone this repository. For this, make sure you have a GitHub account, fork ridgeplot to your GitHub account by clicking the Fork button, and clone the main repository locally (e.g. using SSH)

```
git clone git@github.com:tpvasconcelos/ridgeplot.git
cd ridgeplot
```

You will also need to add your fork as a remote to push your work to. Replace `{username}` with your GitHub username.

```
git remote add fork git@github.com:{username}/ridgeplot.git
```

The following command will 1) create a new virtual environment (under `.venv`), 2) install ridgeplot in editable mode (along with all it's dependencies), and 3) set up and install all pre-commit hooks. Make sure you always work within this virtual environment (i.e., `$ source .venv/bin/activate`). On top of this, you should also set up your IDE to always point to this python interpreter. In PyCharm, open `Preferences -> Project:  ridgeplot -> Project Interpreter` and point the python interpreter to `.venv/bin/python`.

```
make init
```

The default and **recommended** base python is `python3.8`. You can change this by exporting the `BASE_PYTHON` environment variable:

```
BASE_PYTHON=python3.12 make init
```

If you need to use jupyter-lab, you can install all extra requirements, as well as set up the environment and jupyter kernel with

```
make init-jupyter
```

8.2 Pull Request Workflow

1. Always confirm that you have properly configured your Git username and email.

   ```
   git config --global user.name 'Your name'
   git config --global user.email 'Your email address'
   ```

2. Each release series has its own branch (i.e. `MAJOR.MINOR.x`). If submitting a documentation or bug fix contribution, branch off of the latest release series branch.

   ```
   git fetch origin
   git checkout -b <YOUR-BRANCH-NAME> origin/x.x.x
   ```

   Otherwise, if submitting a new feature or API change, branch off of the `main` branch

   ```
   git fetch origin
   git checkout -b <YOUR-BRANCH-NAME> origin/main
   ```

3. Apply and commit your changes.
4. Include tests that cover any code changes you make, and make sure the test fails without your patch.
5. Add an entry to docs/reference/changelog.md summarising the changes in this pull request. The entry should follow the same style and format as other entries, i.e.

   ```
   - Your summary here. (#XXX)
   ```

   where `#XXX` should link to the relevant pull request. If you think that the changes in this pull request do not warrant a changelog entry, please state it in your pull request's description. In such cases, a maintainer should add a `skip news` label to make CI pass.

6. Make sure all integration approval steps are passing locally (i.e., `tox`).
7. Push your changes to your fork

   ```
   git push --set-upstream fork <YOUR-BRANCH-NAME>
   ```

8. Create a pull request. Remember to update the pull request's description with relevant notes on the changes implemented, and to link to relevant issues (e.g., `fixes #XXX` or `closes #XXX`).
9. Wait for all remote CI checks to pass and for a ridgeplot contributor to approve your pull request.

8.3 Continuous Integration

From GitHub’s Continuous Integration and Continuous Delivery (CI/CD) Fundamentals:

> *Continuous Integration (CI) automatically builds, tests, and* ***integrates*** *code changes within a shared repository.*

The first step to Continuous Integration (CI) is having a version control system (VCS) in place. Luckily, you don’t have to worry about that! As you have already noticed, we use Git and host on GitHub.

On top of this, we also run a series of integration approval steps that allow us to ship code changes faster and more reliably. In order to achieve this, we run automated tests and coverage reports, as well as syntax (and type) checkers, code style formatters, and dependency vulnerability scans.

8.3.1 Running it locally

Our tool of choice to configure and reliably run all integration approval steps is Tox, which allows us to run each step in reproducible isolated virtual environments. To trigger all checks in parallel, simply run:

```
tox -p -m static tests
```

It’s that simple !! Note only that this will take a while the first time you run the command, since it will have to create all the required virtual environments (along with their dependencies) for each CI step.

The configuration for Tox can be found in tox.ini.

Tests and coverage reports

We use pytest as our testing framework, and pytest-cov to track and measure code coverage. You can find all configuration details in tox.ini. To trigger all tests, simply run

```
tox -p -m tests
```

If you need more control over which tests are running, or which flags are being passed to pytest, you can also invoke `pytest` directly which will run on your current virtual environment. Configuration details can be found in tox.ini.

Linting

This project uses pre-commit hooks to check and automatically fix any formatting rules. These checks are triggered before creating any git commit. To manually trigger all linting steps (i.e., all pre-commit hooks), run

```
pre-commit run --all-files
```

For more information on which hooks will run, have a look inside the .pre-commit-config.yaml configuration file. If you want to manually trigger individual hooks, you can invoke the `pre-commit`script directly. If you need even more control over the tools used you could also invoke them directly (e.g., `isort .`). Remember however that this is **not** the recommended approach.

8.3.2 GitHub Actions

We use GitHub Actions to automatically run all integration approval steps defined with Tox on every push or pull request event. These checks run on all major operating systems and all supported Python versions. Finally, the generated coverage reports are uploaded to Codecov and Codacy. Check .github/workflows/ci.yaml for more details.

8.3.3 Tools and software

Here is a quick overview of all CI tools and software in use, some of which have already been discussed in the sections above.

Tool	Category	config files	Details
Tox	Orchestration	tox.ini	We use Tox to reliably run all integration approval steps in reproducible isolated virtual environments.
GitHub Actions	Orchestration	.github/work	Workflow automation for GitHub. We use it to automatically run all integration approval steps defined with Tox on every push or pull request event.
Git	VCS	.gitignore	Projects version control system software of choice.
pytest	Testing	tox.ini	Testing framework for python code.
pytest-cov	Coverage	tox.ini	Coverage plugin for pytest.
Codecov and Codacy	Coverage		Two great services for tracking, monitoring, and alerting on code coverage and code quality.
pre-commit hooks	Linting	.pre-commit-config.yaml	Used to to automatically check and fix any formatting rules on every commit.
mypy	Linting	mypy.ini	A static type checker for Python. We use quite a strict configuration here, which can be tricky at times. Feel free to ask for help from the community by commenting on your issue or pull request.
black	Linting	pyproject.toml	"The uncompromising Python code formatter". We use `black` to automatically format Python code in a deterministic manner. We use a maximum line length of 100 characters.
flake8	Linting	setup.cfg	Used to check the style and quality of python code.
isort	Linting	setup.cfg	Used to sort python imports.
EditorConfig	Linting	.editorconfig	This repository uses the `.editorconfig` standard configuration file, which aims to ensure consistent style across multiple programming environments.

8.4 Project structure

8.4.1 Community health files

GitHub's community health files allow repository maintainers to set contributing guidelines to help collaborators make meaningful, useful contributions to a project. Read more on this official reference.

- CODE_OF_CONDUCT.md - A CODE_OF_CONDUCT file defines standards for how to engage in a community. For more information, see "Adding a code of conduct to your project."
- CONTRIBUTING.md - A CONTRIBUTING file communicates how people should contribute to your project. For more information, see "Setting guidelines for repository contributors."

8.4.2 Configuration files

For more context on some of the tools referenced below, refer to the sections on *Continuous Integration*.

- .github/workflows/ci.yaml - Workflow definition for our CI GitHub Actions pipeline.
- .pre-commit-config.yaml - List of pre-commit hooks.
- .editorconfig - EditorConfig standard configuration file.
- mypy.ini - Configuration for the `mypy` static type checker.
- pyproject.toml -
- build system requirements (probably won't need to touch these!) and black configurations.
- setup.cfg - Here, we specify the package metadata, requirements, as well as configuration details for flake8 and isort.
- tox.ini - Configuration for tox, pytest, and coverage.

8.5 Release process

You need push access to the project's repository to make releases. The following release steps are here for reference only.

1. Review the `## Unreleased changes` section in docs/reference/changelog.md by checking for consistency in format and, if necessary, refactoring related entries into relevant subsections (e.g. *Features*, *Docs*, *Bugfixes*, *Security*, etc.). Take a look at previous release notes for guidance and try to keep it consistent.
2. Submit a pull request with these changes only and use the `"Cleanup release notes for X.X.X release"` template for the pull request title. ridgeplot uses the SemVer (`MAJOR.MINOR.PATCH`) versioning standard. You can determine the latest release version by running `git describe --tags --abbrev=0` on the `main` branch. Based on this, you can determine the next release version by incrementing the MAJOR, MINOR, or PATCH. More on this on the next section. For now, just make sure you merge this pull request into the `main` branch before continuing.
3. Use the bumpversion utility to bump the current version. This utility will automatically bump the current version, and issue a relevant commit and git tag. E.g.,

```
# Bump MAJOR version (e.g., 0.4.2 -> 1.0.0)
bumpversion major
```

(continues on next page)

(continued from previous page)

```
# Bump MINOR version (e.g., 0.4.2 -> 0.5.0)
bumpversion minor

# Bump PATCH version (e.g., 0.4.2 -> 0.4.3)
bumpversion patch
```

You can always perform a dry-run to see what will happen under the hood.

```
bumpversion --dry-run --verbose [--allow-dirty] [major,minor,patch]
```

4. Push your changes along with all tag references:

```
git push && git push --tags
```

5. At this point a couple of GitHub Actions workflows will be triggered:
   1. `.github/workflows/ci.yaml`: Runs all CI checks with Tox against the new changes pushed to `main`.
   2. `.github/workflows/release.yaml`: Issues a new GitHub release triggered by the new git tag pushed in the previous step.
   3. `.github/workflows/publish-pypi.yaml`: Builds, packages, and uploads the source and wheel package to PyPI (and test PyPI). This is triggered by the new GitHub release created in the previous step.
6. **Trust but verify!**
   1. Verify that all three workflows passed successfully: https://github.com/tpvasconcelos/ridgeplot/actions
   2. Verify that the new git tag is present in the remote repository: https://github.com/tpvasconcelos/ridgeplot/tags
   3. Verify that the new release is present in the remote repository and that the release notes were correctly parsed: https://github.com/tpvasconcelos/ridgeplot/releases
   4. Verify that the new package is available in PyPI: https://pypi.org/project/ridgeplot/
   5. Verify that the docs were updated and published to https://ridgeplot.readthedocs.io/en/stable/

8.6 Code of Conduct

Please remember to read and follow our standard Code of Conduct.

PYTHON MODULE INDEX

INDEX