ridgeplot

Release 0.6.0

Tomas Pereira de Vasconcelos

Apr 07, 2026

GETTING STARTED

1 Installing from PyPi 3

2 Installing from source 5

3 Dependencies 7

4 Getting started 9

5 API Reference 17

6 Alternatives 25

7 Release Notes 27

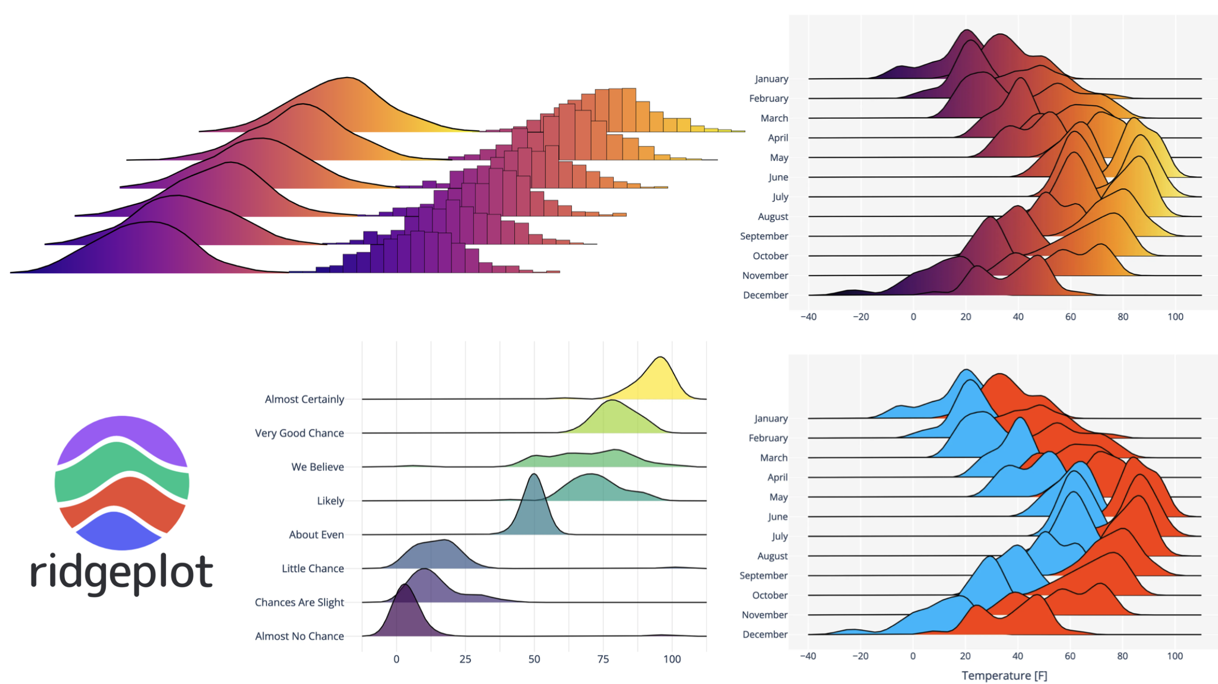
8 Contributing 39

9 Release process 45

Index 47

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.

```python
import numpy as np
from ridgeplot import ridgeplot

my_samples = [np.random.normal(n, size=900) for n in range(6, 0, -2)]
fig = ridgeplot(samples=my_samples)
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, size=900) for n in range(6, 0, -2)]
fig = ridgeplot(samples=my_samples)
fig.show()
```

By default, the *ridgeplot()* function will estimate the samples' probability density functions (PDFs) using kernel density estimation (KDE) and plot them as ridgeline area traces (*trace_type="area"*). If you want to plot histograms instead, you can set the *nbins* parameter to an integer, which will automatically switch the trace type to "bar".

```
fig = ridgeplot(samples=my_samples, nbins=20)
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",
```

(continues on next page)

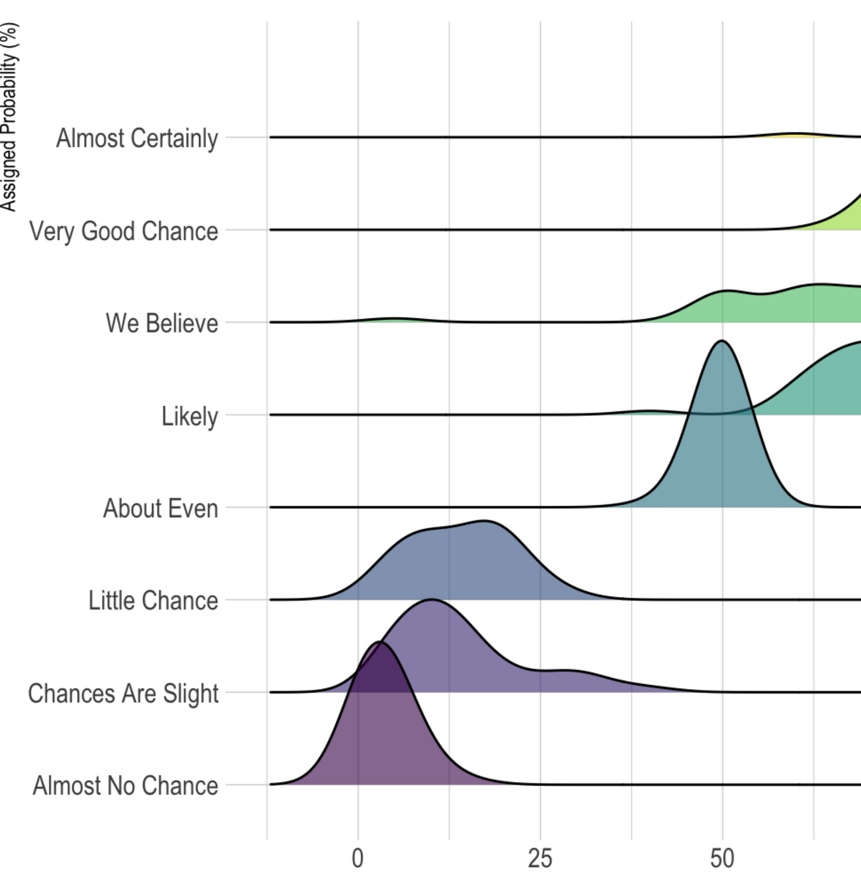
(continued from previous page)

```
    "Very Good Chance",
    "We Believe",
    "Likely",
    "About Even",
    "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",
    opacity=0.6,
    labels=column_names,
    spacing=5 / 9,
)

# And you can still update and extend the final
# Plotly Figure using standard Plotly methods
fig.update_layout(
    height=560,
    width=800,
    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=dict(text="Assigned Probability (%)", font_size=13),
    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

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

```

(continues on next page)

(continued from previous page)

```
# 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=[["Min Temperature [F]", "Max Temperature [F]"]] * len(months),
    row_labels=months,
    colorscale="Inferno",
    bandwidth=4,
    kde_points=np.linspace(-40, 110, 400),
    spacing=0.3,
)
fig.update_layout(
    title="Minimum and maximum daily temperatures in Lincoln, NE (2016)",
    height=600,
    width=800,
    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
```

(continues on next page)

(continued from previous page)

```
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=[["Min Temperature [F]", "Max Temperature [F]"]] * len(months),
```

(continues on next page)

(continued from previous page)

```
    row_labels=months,
    colorscale="Inferno",
    bandwidth=4,
    kde_points=np.linspace(-40, 110, 400),
    spacing=0.3,
)

fig.update_layout(
    title="Minimum and maximum daily temperatures in Lincoln, NE (2016)",
    height=600,
    width=800,
    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()
```

4.4 Coloring options

Note

We are currently investigating the best way to support all color options available in Plotly Express. If you have any suggestions or requests, or just want to track the progress, please check out #226.

The `ridgeplot()` function offers flexible customisation options that help you control the exact coloring of ridgeline traces. Take a look at `colorscale`, `colormode`, `color_discrete_map`, `opacity`, and `line_color` for a detailed description of the available options.

As a simple (but quite common) example, we'll try to adjust the output of the previous example to use different discrete colors for the minimum and maximum temperature traces. Specifically, we'll set all minimum temperature traces to a shade of blue and all maximum temperature traces to a shade of red. To achieve this, we just need to set the `color_discrete_map` parameter to a dictionary that maps the trace labels to the desired colors.

Note

Because the `color_discrete_map` parameter takes precedence over the `colorscale` and `colormode` parameters, we can keep them as they are in the previous example. However, since their behavior will be overridden by `color_discrete_map`, it is a good practice to remove them from the function call to avoid any confusion.

```
fig = ridgeplot(
    # Same options as before, with the
    # addition of `color_discrete_map`
    # ...
    color_discrete_map={
```

(continues on next page)

(continued from previous page)

```
        "Min Temperature [F]": "deepskyblue",
        "Max Temperature [F]": "orangered",
    }
    # ...
)
```

CHAPTER

FIVE

API REFERENCE

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

5.1 ridgeplot.ridgeplot

`ridgeplot.`**`ridgeplot`**(*samples=None*, *densities=None*, *trace_type=None*, *labels=None*, *row_labels=None*, *kernel='gau'*, *bandwidth='normal_reference'*, *kde_points=500*, *nbins=None*, *sample_weights=None*, *norm=None*, *colorscale=None*, *colormode='fillgradient'*, *color_discrete_map=None*, *opacity=None*, *line_color='black'*, *line_width=None*, *spacing=0.5*, *xpad=0.05*, *coloralpha=<MISSING>*, *linewidth=<MISSING>*, *show_yticklabels=<MISSING>*)

Return an interactive ridgeline (Plotly) `Figure`.

Note

You must specify either *`samples`* or *`densities`* to this function, but not both. When specifying *`samples`*, the function will estimate the densities using either Kernel Density Estimation (KDE) or histogram binning. When specifying *`densities`*, the function will skip the density estimation step and use the provided densities directly. See the parameter descriptions below for more details.

Parameters

- **samples** (`Samples` or `ShallowSamples`) – If `samples` data is specified, either Kernel Density Estimation (KDE) or histogram binning will be performed to estimate the underlying densities.

  See *`kernel`*, *`bandwidth`*, and *`kde_points`* for more details on the different KDE parameters. See *`nbins`* for more details on histogram binning. The *`sample_weights`* parameter can be used for both KDE and histogram binning.

  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 density estimation step will be performed over the sample values (S_t) for all traces. The resulting array will be a (4D) *densities* array of shape $(R, T_r, P_t, 2)$ (see *densities* below for more details).

- **densities** (`Densities` or `ShallowDensities`) – If a `densities` array is specified, the density estimation 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)

  See *samples* above for more details.
- **trace_type** (`TraceTypesArray` or `ShallowTraceTypesArray` or `TraceType` or `None`) – The type of trace to display. Choices are `'area'` or `'bar'`. If a single value is passed, it will be used for all traces. If a list of values is passed, it should have the same shape as the samples array. If not specified (default), the traces will be displayed as area plots (`trace_type='area'`) unless histogram binning is used, in which case the traces will be displayed as bar plots (`trace_type='bar'`).

  Added in version 0.3.0.
- **labels** (`LabelsArray` or `ShallowLabelsArray` or `None`) – A collection of string labels for each trace. If not specified (default), the labels will be automatically generated as `"Trace {i}"`, where `i` is the trace's index. If instead a collection of labels is specified, it should have the same shape as the samples array.
- **row_labels** (`Collection[str]` or `None` or `False`) – A collection of string labels for each row in the ridgeline plot. If specified, the length of this collection should match the number of rows in the plot (i.e., the R dimension in the *samples* or *densities* parameter). If not specified (default), the row labels displayed on the y-axis will be automatically generated based on the *labels* argument. If set to `False`, the row labels won't be displayed at all.

  Added in version 0.4.0: Added support for custom row labels, and replaced the deprecated *show_yticklabels* parameter.
- **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 arguments, i.e., fn(x, kern), and return a float, where:
    * x: the clipped input data
    * kern: the kernel instance used

- **kde_points** (KDEPoints) – This parameter 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.
- **nbins** (int or None) – The number of bins to use when applying histogram binning. If not specified (default), KDE will be used instead of histogram binning.

  Added in version 0.3.0.
- **sample_weights** (SampleWeightsArray or ShallowSampleWeightsArray or SampleWeights or None) – An (optional) array of KDE weights corresponding to each sample. The weights should have the same shape as the samples array. If not specified (default), all samples will be weighted equally.
- **norm** (NormalisationOption or None) – The normalisation option to use when normalising the densities. If not specified (default), no normalisation will be applied and the densities will be used *as is*. The following normalisation options are available:
  - "probability" - normalise the densities by dividing each trace by its sum.
  - "percent" - same as "probability", but the normalised values are multiplied by 100.

  Added in version 0.2.0.
- **colorscale** (ColorScale or Collection[Color] or str or None) – A continuous color scale used to color the different traces in the ridgeline plot. It can be represented by a string name (e.g., "viridis"), a ColorScale object, or a list of valid Color objects. If a string name is provided, it must be one of the built-in color scales (see list_all_colorscale_names() and Plotly's built-in color-scales). If a list of colors is provided, it must be a list of valid CSS colors (e.g., ["rgb(255, 0, 0)", "blue", "hsl(120, 100%, 50%)"]). The list will ultimately be converted into a ColorScale object, assuming the colors provided are evenly spaced. If not specified (default), the color scale will be inferred from current Plotly template.
- **colormode** ("fillgradient" or SolidColormode) – This parameter controls the logic used for the coloring of each ridgeline trace.

  The "fillgradient" mode (default) will fill each trace with a gradient using the specified *colorscale*. The gradient normalisation is done using the minimum and maximum x-values over all densities.

  All other modes provide different methods for calculating interpolation values from the specified *colorscale* (i.e., a float value between 0 and 1) for each trace. The interpolated color will be used to color each trace with a solid color. The available modes are:

  - "row-index" - uses the row's index. This is useful when the desired effect is to have the same color for all traces on the same row. e.g., if a ridgeplot has 3 rows of traces, then the color scale interpolation values will be `[[0, ...], [0.5, ...], [1, ...]]`, respectively.
  - "trace-index" - uses the trace's index. e.g., if a ridgeplot has a total of 3 traces (across all rows), then the color scale interpolation values will be 0, 0.5, and 1, respectively, and regardless of each trace's row.
  - "trace-index-row-wise" - uses the row-wise trace index. This is similar to the "trace-index" mode, but the trace index is reset for each row. e.g., if a ridgeplot has a row with only one trace and another with two traces, then the color scale interpolation values will be `[[0], [0, 1]]`, respectively.
  - "mean-minmax" - uses the min-max normalised (weighted) mean of each density to calculate the interpolation values. The normalisation min and max values are the *absolute* minimum and maximum x-values over all densities. This mode is useful when the desired effect is to have the color of each trace reflect the mean of the distribution, while also taking into account the distributions' spread.
  - "mean-means" - similar to the "mean-minmax" mode, but where the normalisation min and max values are the minimum and maximum *mean* x-values over all densities. This mode is useful when the desired effect is to have the color of each trace reflect the mean of the distribution, but without taking into account the entire variability of the distributions.

  Changed in version 0.2.0: The default value changed from "mean-minmax" to "fillgradient".

- **color_discrete_map** (`dict` or `None`) – A mapping from trace labels to specific colors.

  This parameter is useful when you want to have full manual control over the colors assigned to each trace. If specified, the assigned colors are determined by looking up the trace's label as a key in this dictionary. All labels must be present as keys in the dictionary.

  Note that this parameter overrides any value specified for *colorscale* and *colormode*. In this case, the color assigned to each trace will be a solid color, as specified in this dictionary.

  If not specified (default), the colors will be determined using the *colorscale* and *colormode* parameters.

  Added in version 0.5.0.

- **opacity** (`float` or `None`) – If None (default), this parameter will be ignored and the transparency values of the specified color-scale will remain untouched. Otherwise, if a float value is passed, it will be used to overwrite the opacity/transparency of the color-scale's colors.

  Added in version 0.2.0: Replaces the deprecated *coloralpha* parameter.

- **line_color** (`Color` or `"fill-color"`) – The color of the traces' lines. Any valid CSS color is allowed (default: `"black"`). If the value is set to "fill-color", the line color will be the same as the fill color of the traces (see *colormode*). If `colormode='fillgradient'`, the line color will be the mean color of the fill gradient (i.e., equivalent to the fill color when `colormode='mean-minmax'`).

  Added in version 0.2.0.

- **line_width** (`float` or `None`) – The traces' line width (in px). If not specified (default), area plots will have a line width of 1.5 px, and bar plots will have a line width of 0.5 px.

  Added in version 0.2.0: Replaces the deprecated *linewidth* parameter.

  Changed in version 0.2.0: The default value changed from 1 to 1.5

- **spacing** (`float`) – The vertical spacing between density traces, which is defined in units of the highest distribution (i.e., the maximum y-value).
- **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.
- **coloralpha** (`float`) –

  Deprecated since version 0.2.0: Use *`opacity`* instead.
- **linewidth** (`float`) –

  Deprecated since version 0.2.0: Use *`line_width`* instead.
- **show_yticklabels** (`bool`) –

  Deprecated since version 0.4.0: Use *`row_labels`* instead.

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 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.

5.2.1 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:

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

"zonination"

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

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

5.2.2 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`

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].

[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/

[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/

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

References

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

- …

7.2 0.6.0

- Add support for Python 3.14, in accordance with the official Python support policy[1] (#346)

7.2.1 Bug fixes

- Fix the way histogram bin centers are computed (#364)

7.2.2 Documentation

- Update the basic examples throughout the docs (#364)
- Bundle a local copy of `plotly.min.js` as a CDN fallback (#364)

7.2.3 Developer Experience

- Add AGENTS.md, CLAUDE.md, .codex/, and .cursor/ helper files for AI-assisted development (#367)
- Small improvements to the project's Makefile (#367)
- Add a basic version of Cursor's `worktrees.json` config (#364)

7.2.4 CI/CD

- Remove the JPEGs used for visual inspection in regression tests (#364)
- Upgrade ruff's target Python version to 3.10 (#363)
- Bump actions/checkout from 5 to 6 (#357)
- Bump actions/download-artifact from 5 to 6 (#354)
- Bump actions/download-artifact from 6 to 7 (#361)
- Bump actions/upload-artifact from 4 to 5 (#353)

[1] https://devguide.python.org/versions/

- Bump actions/upload-artifact from 5 to 6 (#360)
- Bump sigstore/gh-action-sigstore-python from 3.0.1 to 3.1.0 (#352)
- Bump sigstore/gh-action-sigstore-python from 3.1.0 to 3.2.0 (#359)
- pre-commit autoupdate (#350, #355, #358, #362, and #366)

7.3 0.5.0

7.3.1 Breaking changes

- Dropped support for Python 3.9, in accordance with the official Python support policy[Page 27, 1] (#345)

7.3.2 Features

- Implement a new `color_discrete_map` parameter to allow users to specify custom colors for each trace (#348)

7.3.3 Miscellaneous

- Bump project classification from Pre-Alpha to Alpha (#336)

7.3.4 CI/CD

- Bump actions/github-script from 7 to 8 (#338)
- pre-commit autoupdate (#340)
- Bump peter-evans/find-comment from 3 to 4 (#342)
- pre-commit autoupdate (#341)
- Bump github/codeql-action from 3 to 4 (#344)
- Bump peter-evans/create-or-update-comment from 4 to 5 (#343)

7.4 0.4.0

7.4.1 Features

- Add support for custom row labels via a new `row_labels` argument (#333)

7.4.2 Deprecations

- Deprecated the `show_yticklabels` argument in favor of the new more general and flexible `row_labels` argument (#333)

7.4.3 CI/CD

- Add `./cicd_utils/find-unmentioned-prs.sh` helper script to find merged PRs that were not mentioned in the changelog yet (#334)
- Bump actions/first-interaction from 1 to 3 (#331)
- Bump actions/checkout from 4 to 5 (#330)

- Bump actions/download-artifact from 4 to 5 (#329)
- Bump sigstore/gh-action-sigstore-python from 3.0.0 to 3.0.1 (#326)
- pre-commit autoupdate (#324)

7.4.4 Dependencies

- Remove `importlib_metadata` usage from `conf.py` (#332)

7.5 0.3.2

7.5.1 Internal

- Fix regression tests for Plotly 6.0+ by updating the JSON test artifacts (#313)
- Add an `OFFLINE`-mode option to the Makefile and make `uv` compulsory for local development (#322)

7.5.2 Documentation

- Bump the Plotly JS version used in Sphinx from 2.35.2 to 3.0.0 (#317)

7.6 0.3.1

7.6.1 Internal

- Improve type annotations and use stricter pyright settings (#291)

7.6.2 Documentation

- Use `sphinxcontrib.apidoc` to automatically generate API docs from the source code (#296)
- Update hero image in the docs' landing page (#300)
- Update release process notes (#301 and #303)

7.6.3 CI/CD

- Fix regressions tests by comparing against JSON artifacts instead of flaky JPEGs (#299)

Thanks to @imprvhub for their contributions to this release!

7.7 0.3.0

7.7.1 Features

- Add support for histogram and bar traces (#287)

7.7.2 Documentation

- Small improvements to `ridgeplot()`'s docstring (#284)
- Misc improvements to the API docs and the getting-started and contributing guides (#287)

7.7.3 Internal

- Small improvements to type hints and annotations (#284)
- Introduce an internal `ridgeplot._obj` package to hold object-oriented interfaces (#287)

7.7.4 CI/CD

- Improve type annotations and switch from mypy to pyright with stricter settings (#287)
- Switch from `black` to the new `ruff` formatter (#287)

7.8 0.2.1

7.8.1 Bug fixes

- Fixed `ZeroDivisionError` for index-based colormodes when specifying single-trace or single-row plots (#268)

7.9 0.2.0

After almost 4 years, 30 *"patch"* releases, +200 pull-requests, and close to 1,000 commits, this is ridgeplot's first *minor* release (`v0.1.30 -> v0.2.0`)!

ridgeplot has been downloaded over 400k times (peaking at 102k downloads in a single month), is listed as a dependency in 135 public GitHub repositories, and - perhaps most relevantly - is a dependency of larger projects such as Shiny for Python, Ploomber, and NiMARE which further extends the impact and reach of the project.

This release marks a small milestone for ridgeplot, which we believe has now reached a level of maturity and stability that warrants a stricter and more structured, predictable, and standard release and versioning process. Even though we have managed to never publish breaking changes in the past (if you find any, please let us know!), we will from now on be even more careful and mindful of the impact of any changes that could affect downstream users and their applications.

We will make an effort to standardise and document our versioning policy. For now, we will try to simply adhere to the following general rules:

- We are explicitly **not** going to follow Semantic Versioning, as we believe it is not a good fit for this project yet.
- `MAJOR.MINOR.PATCH` versioning scheme:
  - **MAJOR**: We don't have any plans for this yet… we will probably use this in the future once we settle on a more stable API and feature set
  - **MINOR**: New features, significant improvements, and deprecations
  - **PATCH**: Backwards-compatible bug fixes, small improvements, internal changes, and documentation updates
- **Breaking changes:**
  - **We might introduce breaking changes in minor releases!**

  - However, this will never happen without a proper deprecation period and a clear upgrade path. i.e., we will always first deprecate the old API via a `DeprecationWarning` and provide a clear migration path to the new API.
  - Such instances will be kept to a minimum and will likely only show up in the form of deprecated or renamed parameters or the meaning/behaviour of their arguments/values.

7.9.1 Breaking changes

- Remove support for the deprecated `show_annotations` parameter and `colormode='index'` value (#254)
- The new default colormode is `"fillgradient"` (#244)
- The default value for `line_width` changed from `1` to `1.5` (#253)

7.9.2 Features

- Implement new `"fillgradient"` colormode (#244)
- Add new `line_color` parameter to the `ridgeplot` function (#253)
- Add a `line_color='fill-color'` option which automatically matches the trace's line color to the trace's fill color (#253)
- Add new `norm` parameter to the `ridgeplot` function to allow users to normalize the data before plotting (#255)
- Add `sample_weights` argument to `ridgeplot()` to allow users to pass sample weights to the KDE estimator (#259)

7.9.3 Deprecations

- Rename `coloralpha` to `opacity` for consistently with Plotly Express and deprecate the old parameter name (#245)
- Rename `linewidth` to `line_width` for consistency with Plotly's API and deprecate the old parameter name (#253)
- Deprecated `colorscale='default'` and `list_all_colorscale_names()` in favour or Plotly Express' `px.colors.named_colorscales()` (#262)

7.9.4 Dependencies

- The new minimum version of Plotly is `5.20` to leverage the new `fillgradient` feature (#244)

7.9.5 Optimizations

- Importing statsmodels, scipy, and numpy can be slow, so we now only import the `ridgeplot._kde` module when the user needs this functionality (#242)

7.9.6 Documentation

- Update examples in the getting-started guide to reflect the new default colormode (#244)
- Update the `plotly.min.js` version from `2.27` to `2.35.2` to leverage the `fillgradient` feature (#244)
- Fix the API reference docs for the internal `ridgeplot._color` module (#244)
- Tighten margins in generated examples (#257)
- Add the reference jupyter notebook used to generate the ridgeplot logo (#242)
- Update ridgeplot's logo to use Plotly's official colors (#243)

7.9.7 CI/CD

- Stop sending coverage reports to Codacy (#265)
- Improve local development experience and optimise the CI pipeline (#273)

7.9.8 Internal

- Simplify and refactor `interpolate_color` to not depend on `px.colors.find_intermediate_color` (#253)
- Improve type narrowing using `typing.TypeIs` (#259)
- Refactor community health files (#260)

Thanks to @sstephanyy for their contributions to this release!

7.10 0.1.30

7.10.1 Features

- Add support for named CSS colors (#229)
- Allow users to define color scales as a collection of colors. (`Collection[Color]`) (#231)
- Dynamically infer the default colorscale from the active Plotly template (#237)

7.10.2 Documentation

- Improve the documentation for the `colormode` parameter (#232)

7.10.3 Internal

- Refactor `_figure_factory.py` to use a functional approach (#228)
- Stop using the term "midpoints" to refer to the "interpolation values" when dealing with continuous color scales (#232)
- Refactor color validation logic to use helpers provided by Plotly (#233)
- Drop `colors.json` and use Plotly's `ColorscaleValidator` and `named_colorscales` directly (#234)
- Refactor color utilities into `ridgeplot._color` (#235)

7.11 0.1.29

7.11.1 Features

- Add new `"trace-index-row-wise"` colormode (#224)

7.11.2 Improvements

- Remove duplicated labels when plotting multiple traces on the same y-axis row (#223)

7.11.3 Documentation

- Update and improve the "Contributing" guide (#218 and #221)

7.11.4 Internal

- Eagerly validate input shapes in `RidgeplotFigureFactory` (#222)
- Vendor `_zip_equal()` from more-itertools (#222)
- Improve overall test coverage (#222)

7.11.5 Bug fixes

- Support edge case in `get_collection_array_shape` where the input array is empty or contains nested empty arrays (#222)

7.11.6 CI/CD

- Add new `"Greet new users"` workflow to welcome new contributors to the project (#210)
- Add `concurrency` entries to relevant GitHub workflows (#211)
- Add Dependabot configuration file (#211)
- Add GitHub issue templates (#211)
- Add support for Python 3.13 (#217)
- Add a CodeQL GitHub workflow (#220)

7.12 0.1.28

- Ooops! This release was skipped due to a mistake in the release process. The changes in this release were included in the 0.1.29 release.

7.13 0.1.27

7.13.1 Breaking changes

- Dropped support for Python 3.8, in accordance with the official Python support policy[Page 27, 1] (#204)
- Removed deprecated function `get_all_colorscale_names()` in favor of `list_all_colorscale_names()` (#200)

7.13.2 CI/CD

- Adopt `setuptools-scm` for package versioning (#200)
- Add `actionlint` pre-commit hook (#201)
- Improve logic in `.github/workflows/check-release-notes.yml` to post comments to the PR (#201)
- Handle footnotes in the automatically generated release notes (#209)

7.14 0.1.26

7.14.1 Breaking changes

- Dropped support for `statsmodels==0.14.2` due to import-time issue. See #197 for more details (#198)

7.14.2 CI/CD

- Refactor test coverage logic (#193)
- Replace `pip` and `venv` with `uv` (#189)
- 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)

7.15 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.

7.15.1 Documentation

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

7.15.2 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)

7.15.3 CI/CD

- Add support for Python 3.12 (#182)

7.16 0.1.24

7.16.1 Breaking changes

- Dropped support for Python 3.7, in accordance with the official Python support policy[Page 27, 1] (#154)

7.16.2 Features

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

7.16.3 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 artifacts 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)

7.16.4 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)

7.16.5 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)

7.17 0.1.23

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

7.18 0.1.22

7.18.1 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)

7.18.2 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.

7.18.3 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)

7.18.4 Internal

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

7.19 0.1.21

7.19.1 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)

7.19.2 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)

7.19.3 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)

7.20 0.1.17

- Automate the release process. See .github/workflows/release.yml, 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)

7.21 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)

7.22 0.1.14

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

7.23 0.1.13

- Add tests for example scripts (#14)

7.24 0.1.12

7.24.1 Internal

- Update and standardise CI steps (#6)

7.24.2 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)

7.25 0.1.11

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

7.26 0.1.0

- Initial release!

CHAPTER

EIGHT

CONTRIBUTING

Thank you for your interest in improving ridgeplot!

We really appreciate you taking the time to help make this project better for everyone.

The contribution process for ridgeplot usually starts with filing a GitHub issue. We define two main categories of issues, each with its own issue template

- **Feature request**: Suggest a new idea or enhancement to ridgeplot
- **Bug report**: Report an issue you encountered with ridgeplot

For broader discussions, questions, or general feedback, please head over to our GitHub Discussions page.

Please note that this is a volunteer-run project, and we may not be able to respond to every issue or pull request immediately. Our response time may vary, but we appreciate your patience and will try to get back to you as soon as possible.

After we've triaged your issue and an implementation strategy has been agreed on by a ridgeplot maintainer, the next step is to introduce your changes as a pull request (see *Pull Request Workflow*). Once the pull request is merged, the changes will be automatically included in the next ridgeplot release. Every significant 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 if you think something could be improved.

8.1 Development environment

Here are our guidelines for setting a **working** development environment. Most of the steps have been abstracted away using the make build automation tool. Feel free to peak inside the Makefile to see exactly what is being run, and in which order.

> **Prerequisites**
>
> To follow along with this guide, you will need to have git, make, and uv installed on your system. We recommend using uv to also manage your Python installations. For this project you will need Python 3.10 or higher.

First, you will need to clone this repository. For this, make sure you have a GitHub account, fork ridgeplot 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
```

8.1.1 Bootstrapping the development environment

The following command will:

1. Delete any existing environment artifacts (e.g., `.venv`, `.tox`, `.pytest_cache`, etc.)
2. Create a new virtual environment under `.venv`
3. Install ridgeplot in editable mode along with all it's development dependencies
4. Set up and install all pre-commit hooks.

```
make init
```

The default and **recommended** base Python is `python3.10`. However, if you encounter any issues or don't have this specific version installed on your system, you can change by it exporting the `BASE_PYTHON` environment variable to a valid executable you do have installed. Please note that we no longer support any Python versions lower than 3.10. For example, to use `python3.14`, you should run:

```
BASE_PYTHON=python3.14 make init
```

If for whatever reason you don't have a working internet connection (, , , , etc.), you can try exposing the `OFFLINE=1` environment variable. This will only work if compatible packages have already been cached by `uv` on your system.

```
OFFLINE=1 make init
```

If you need to use jupyter in this environment, run the following command:

```
make jupyter-init
```

Note

Make sure you always work within this virtual environment (*e.g.,* `$ source .venv/bin/activate`). We also recommend that you set up your IDE to always point to this Python interpreter. If you are unsure how to do this, please refer to the documentation of your specific IDE, and get comfortable using virtual environments in Python. You can thank us later!

8.2 Pull Request Workflow

If you're reading this, it means you're probably getting ready to submit a pull request (or at least thinking about it). Either way, **congrats!** and we thank you in advance for putting in the time and effort to contribute to ridgeplot!

1. Always confirm that you have properly configured your name and email address in your git environment. This information will be used to identify you as a contributor in the project's commit history.

   ```
   # e.g., to set your name and email address globally
   git config --global user.name '<Your name>'
   git config --global user.email '<Your email address>'
   ```

2. Branch off the `main` branch.

```
# e.g., to create a new branch named `feat/awesome-feature`
git fetch origin
git switch -c feat/awesome-feature origin/main
```

3. **Implement and commit your changes**
4. Make sure all CI checks are passing locally (see *Continuous Integration* below).

```
tox -m static tests
```

5. Push your changes to your fork

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

6. Create a pull request, and remember to update the pull request's description with relevant notes on the changes implemented, and link to relevant issues (e.g., `fixes #XXX` or `closes #XXX`).
7. At this point, you'll probably also want to 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. ({gh-issue}XXX)
   ```

   where `XXX` should be replaced with your PR's number. If you think that the changes in this pull request do not warrant a changelog entry (e.g., simply fixing a typo), please state it in your pull request's description. In such cases, a maintainer should add a `skip news` label to make the CI checks pass.
8. Wait for all remote CI checks to pass and for a ridgeplot maintainer to review your changes. If you're not done with your changes yet, you can set your pull request to a Draft state, which will signal to the maintainers that you're still working on it. **Just remember to mark it as ready for review when you're done!**
9. Once your pull request is approved, it will be merged into the `main` branch, and your changes will be automatically included in the next ridgeplot release.

8.3 Continuous Integration

> Continuous Integration (CI): automatically builds, tests, and **integrates** code changes within a shared repository.
>
> —GitHub: CI/CD: The what, why, and how

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 astutely 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 changes faster and with greater confidence that we won't be breaking things for users down the line. In order to achieve this, we run a suite of automated tests and coverage reports, as well as a series of linters and type checkers.

8.3.1 Running it locally

Our tool of choice for configuring and reliably run all integration checks is Tox, which allows us to run each step in reproducible isolated virtual environments.

To trigger all checks, simply run:

```
tox -m static tests
```

…yes, it's that simple!

Note that this could 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 and each test environment can be found in tox.ini.

If you need more control over which set of checks is running, take a look at the following two sections.

Tests and coverage reports

We use pytest as our testing framework, and Coverage.py to track and measure code coverage.

To trigger all test suites with coverage reports, run:

```
tox -m tests
```

If you need more control over which tests are running, or which flags are being passed to pytest, you can also invoke `tox -e pytest -- <PYTEST_FLAGS>`. For instance, to run only the tests in the `tests/unit/test_init.py` file without coverage, you could run:

```
tox -e pytest -- tests/unit/test_init.py --no-cov
```

For more details on how these checks are configured, take a look at the pytest.ini, .coveragerc, and/or tox.ini configuration files.

Static checks

This project uses pre-commit hooks to check and automatically fix any linting code formatting issues. These checks are triggered against all staged files before creating any git commit. To manually trigger all pre-commit hooks against all files, run:

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

For more information on all the checks being run here, take a look inside the .pre-commit-config.yaml configuration file.

The only static check that is not run by pre-commit is pyright, which is too expensive to run on every commit. To run pyright against all files, run:

```
tox -e typing
```

Just like with pytest, you can also pass extra positional arguments to pyright by running `tox -e typing -- <PYRIGHT_FLAGS>`.

To trigger all static checks, run:

```
tox -m static
```

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. Coverage data is also uploaded to Codecov here. Check .github/workflows for more details.

Additionally, we use CodeQL to automatically check for security vulnerabilities in the codebase. This check is set to run every day but also on every push or pull request event. Check .github/workflows/codeql.yml for more details.

Finally, we have a small workflow (see .github/workflows/check-release-notes.yml) that checks if the PR author remembered to add an entry to the changelog. If the PR does not warrant a changelog entry, the author can add a `skip news` label to make the CI checks pass.

8.4 Tools and software

Here is a quick overview of all most of the CI tools and software used in this project, along with their respective configuration files. If you have any questions or need help with any of these tools, feel free to ask for help from the community by commenting on your issue or pull request.

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/workflows/ci.yml	Workflow automation for GitHub. We use it to automatically run all integration approval steps on every push or pull request event.
Make	Orchestration	Makefile	A build automation tool that we (mis)use to abstract away some bootstrapping and development environment setup steps.
git	VCS	.gitignore	The project's version control system.
pytest	Testing	pytest.ini	Testing framework for python code.
Coverage.py	Coverage	.coveragerc	The code coverage tool for Python
Codecov	Coverage	.github/workflows/ci.yml	An external services for tracking, monitoring, and alerting on code coverage metrics.
pre-commi	Linting	.pre-commit-config.yaml	Used to to automatically check and fix any formatting rules on every commit.
pyright	Linting	pyrightconfig.json	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.
ruff	Linting	ruff.toml	"An extremely fast Python linter and code formatter, written in Rust." For this project, ruff replaced black, Flake8 (+plugins), isort, pydocstyle, pyupgrade, and autoflake with a single (and faster) tool.
EditorConfig	Linting	.editorconfig	This repository uses the `.editorconfig` standard configuration file, which aims to ensure consistent style across multiple programming environments.
bumpversion	Packaging	.bumpversion.cfg	A small command line tool to simplify releasing software by updating all version strings in your source code by the correct increment.
setuptools	Packaging	pyproject.toml and MANIFEST.in	`MANIFEST.in` tells `setuptools` which files to include in the distribution. `pyproject.toml` is the new standard for defining static package metadata.
readthedocs	Documentation	.readthedocs.yaml and .github/workflows/readth preview.yml	An open-source documentation hosting platform. We use it to automatically build and deploy the documentation for this project.

8.5 Code of Conduct

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

CHAPTER

NINE

RELEASE PROCESS

> **This page is intended for maintainers of the project only**
>
> You need to have push-access to the project's repository to make releases. Therefore, the following release steps are intended to be used as a reference for maintainers or collaborators with push-access to the repository.

1. Review the `## Unreleased changes` section at the top of the docs/reference/changelog.md file and, if necessary, group and/or split entries into relevant subsections (e.g., *Features*, *Docs*, *Bugfixes*, *Security*, etc.). Take a look at previous release notes for guidance and try to keep the format consistent. You can also use the `./cicd_utils/find-unmentioned-prs.sh` helper script to find merged PRs that were not mentioned in the changelog yet.

2. Review new usages of `.. versionadded::`, `.. versionchanged::`, and `.. deprecated::` directives that were added to the documentation since the last release. If necessary, update the version numbers in these directives to reflect the new release version.

   - 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 relevant *MAJOR*, *MINOR*, or *PATCH* numbers.

3. **IMPORTANT:** Remember to switch to the `main` branch and pull the latest changes before proceeding.

   ```
   git switch main
   git pull
   ```

4. Use the bumpversion utility to automatically bump the current version, apply a relevant changes to the repository, and create a new commit and git tag. E.g.,

   ```
   # MAJOR release (e.g., 0.4.2 -> 1.0.0)
   SKIP='no-commit-to-branch' bumpversion major

   # MINOR release (e.g., 0.4.2 -> 0.5.0)
   SKIP='no-commit-to-branch' bumpversion minor

   # PATCH release (e.g., 0.4.2 -> 0.4.3)
   SKIP='no-commit-to-branch' 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]
   ```

5. **DANGER:** Push your changes along with the new git tag to the remote repository.

```
git push --follow-tags
```

6. At this point, a couple of GitHub Actions workflows will be triggered:
   1. .github/workflows/ci.yml: Runs all integration approval checks.
   2. .github/workflows/release.yml: Builds and publishes the new packaged Python distributions to PyPi (and TestPyPi) and publishes a new GitHub Release with relevant release notes and Sigstore-certified built distributions.
7. **Trust but verify!**
   1. Verify that all workflows run successfully;
   2. Verify that the new git tag is present in the remote repository;
   3. Verify that the new release is present in the remote repository;
      1. and that the release notes were correctly parsed
      2. and that the relevant assets were correctly uploaded
   4. Verify that the new package is available in PyPI;
      1. and TestPyPI
   5. Verify that the docs were updated and published to ReadTheDocs.

INDEX

L

load_lincoln_weather() (*in module ridgeplot.datasets*), 22
load_probly() (*in module ridgeplot.datasets*), 21

R

ridgeplot() (*in module ridgeplot*), 17