ridgeplot

Tomas Pereira de Vasconcelos

Jun 20, 2022

GETTING STARTED

The `ridgeplot` python library aims at providing a simple API for plotting beautiful ridgeline plots within the extensive Plotly interactive graphing environment.

Bumper stickers:

- Do one thing, and do it well!
- Use sensible defaults, but allow for extensive configuration!

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

Install and update using pip:

```
pip install -U ridgeplot
```

CHAPTER

ONE

DEPENDENCIES

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

CHAPTER

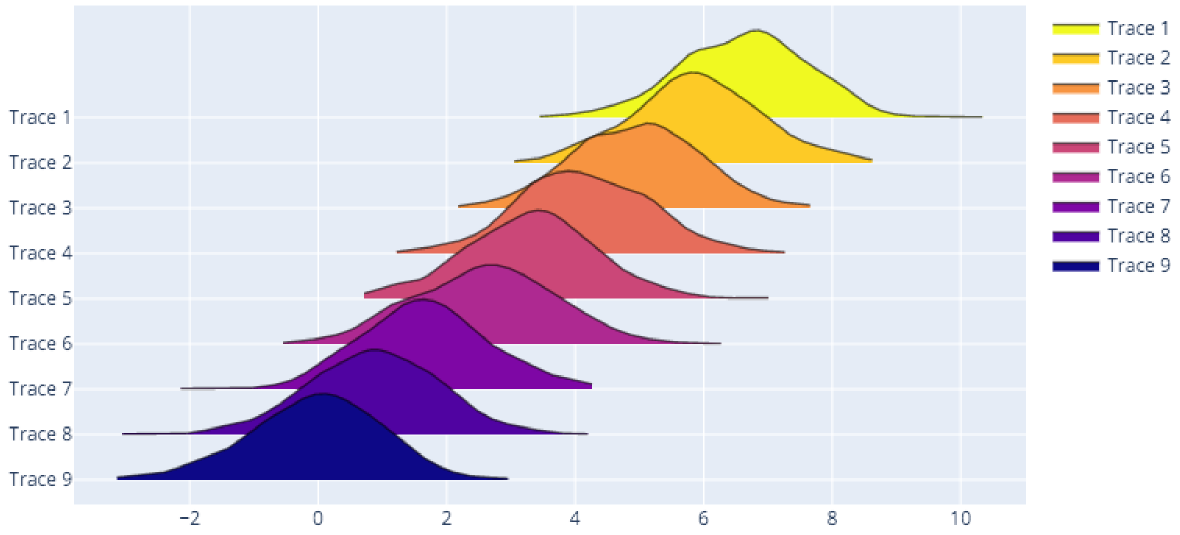
TWO

GETTING STARTED

2.1 Sensible defaults

Get started with a simple function call to *ridgeplot()* with sensible (plotly) defaults.

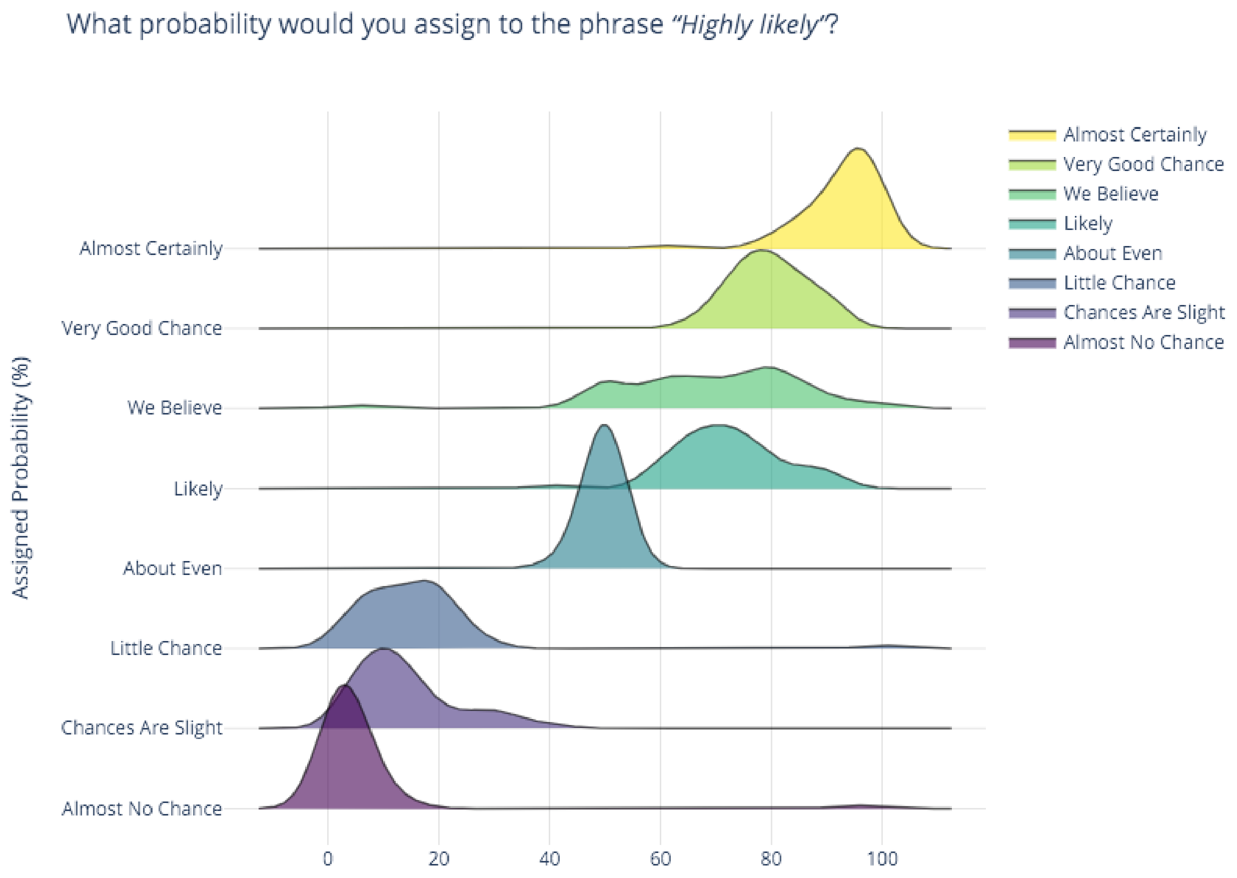
```
import numpy as np

from ridgeplot import ridgeplot

# Put your real samples here...
np.random.seed(0)
synthetic_samples = [np.random.normal(n / 1.2, size=600) for n in range(9, 0, -1)]

# Call the `ridgeplot()` helper, packed with sensible defaults
fig = ridgeplot(samples=synthetic_samples)

# The returned Plotly `Figure` is still fully customizable
fig.update_layout(height=500, width=800)

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



2.2 Fully configurable

In this example, we will be replicating the first ridgeline plot example in this *from Data to Viz* post, which uses the *probly* dataset. You can find the *plobly* dataset on multiple sources like in the bokeh python interactive visualization library. I'll be using the same source used in the original post.

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


# Get the raw data
df = pd.read_csv("https://raw.githubusercontent.com/bokeh/bokeh/main/bokeh/sampledata/_
↪data/probly.csv")

# Let's grab only the subset of columns displayed in the example
column_names = [
    "Almost Certainly", "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.values.T,
    bandwidth=4,
    kde_points=np.linspace(-12.5, 112.5, 400),
    colorscale="viridis",
    colormode="index",
    coloralpha=0.6,
    labels=column_names,
    spacing=5 / 9,
)

# Again, update the figure layout to your liking here
fig.update_layout(
    title="What probability would you assign to the phrase <i>“Highly likely”</i>?",
    height=650,
    width=800,
    plot_bgcolor="rgba(255, 255, 255, 0.0)",
    xaxis_gridcolor="rgba(0, 0, 0, 0.1)",
    yaxis_gridcolor="rgba(0, 0, 0, 0.1)",
    yaxis_title="Assigned Probability (%)",
)
fig.show()
```


What probability would you assign to the phrase *"Highly likely"*?
Assigned Probability (%)
Almost Certainly
Very Good Chance
We Believe
Likely
About Even
Little Chance
Chances Are Slight
Almost No Chance
0
20
40
60
80
100

CHAPTER

THREE

API REFERENCE

ridgeplot.**ridgeplot**(*samples=None*, *densities=None*, *kernel: str = 'gau'*, *bandwidth='normal_reference'*, *kde_points=500*, *colorscale: Union[str, Iterable[Tuple[float, str]]] = 'plasma'*, *colormode: str = 'mean-means'*, *coloralpha: Optional[float] = None*, *labels=None*, *linewidth: float = 1.4*, *spacing: float = 0.5*, *show_annotations: bool = True*, *xpad: float = 0.05*) → plotly.graph_objs._figure.Figure

Creates and returns a Plotly figure with a beautiful ridgeline plot.

You have to specify one of: `samples` or `densities`. If you specify both `samples` and `densities` arguments, a `ValueError` exception will be raised! One of these arguments should always remain set to None. See `samples` and `densities` bellow for more information.

Args:

samples 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.

densities If `densities` arrays are specified instead, the KDE step will be skipped and all associated arguments ignored. Each density array should have shape $(2, N)$, but N may vary with each array.

kernel 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 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 This argument controls the points at which KDE is computed. If an int value is passed (default), the densities will be evaluated at `kde_points` evenly spaced points between the min and max of each set of samples. However, you may also specify a custom range by instead passing an array of points. This array should be one-dimensional.

colorscale Any valid Plotly color-scale or a str with a valid named color-scale. Use *`get_all_colorscale_names()`* to see which names are available or check out Plotly's built-in color-scales.

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:

- `"index"` - uses the trace's index. e.g. if 3 traces are specified, then the midpoints will be [0, 0.5, 1].
- `"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 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 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 The traces' line width (in px).

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

show_annotations If True (default), it will show the label names as "y-tick-labels".

xpad 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.

.._cs: https://plotly.com/python/builtin-colorscales/

Returns:

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

Raises:

`ValueError` If both `samples` and `densities` arguments are not None, or if neither `samples` or `densities` are specified.

`ridgeplot.`**`get_all_colorscale_names`**`()` → Tuple[str]

Returns a tuple with all available colorscale names.

CHAPTER FOUR

CHANGELOG

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

4.1 Unreleased changes

- ...

4.2 0.1.16

- Fix automated release process to PyPi. (#27)

4.3 0.1.15

- 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)

4.4 0.1.14

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

4.5 0.1.13

- Add tests for example scripts (#14)

4.6 0.1.12

4.6.1 Internal

- Update and standardise CI steps (#6)

4.6.2 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)

4.7 0.1.11

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

4.8 0.1.0

- Initial release!

CHAPTER

FIVE

ALTERNATIVES

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

CHAPTER

SIX

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.

6.1 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 `python3.7` . You can change this by exporting the `BASE_PYTHON` environment variable. For instance, you could instead run:

```
BASE_PYTHON=python3.8 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
```

Bonus: If you need to use plotly inside a jupyter-lab notebook, just run

```
make jupyter-plotly
```

6.2 Pull Request Workflow

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

   ```
   git config --global user.name 'Your real 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/2.0.x
   ```

   Otherwise, if submitting a new feature or API change, branch off of the "master" 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 CHANGES.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.

6.3 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.

6.3.1 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, simply run

```
tox
```

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 -e py
```

You can also run your tests against any other supported python versions (e.g., `tox -e py38`). 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

```
tox -e lint
```

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.

6.3.2 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](https://github.com/tpvasconcelos/ridgeplot/blob/master/ .github/workflows/ci.yaml) for more details.

6.3.3 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/workflows/ci.yaml	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 120 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.

6.4 Project structure

6.4.1 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."

6.4.2 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.

6.5 Release process

- 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 master branch.

6.5.1 Release steps

1. Review the `## Unreleased changes` section in CHANGES.md by checking for consistency in format and, if necessary, refactoring related entries into relevant subsections (e.g. Features, Docs, Bugfixes, Security, etc).
   - Submit a pull request with these changes. You may use the `"Update release notes for X.X.X release"` template for the pull request title.
2. 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

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

(continues on next page)

(continued from previous page)

```
# 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]
```

3. Push your changes along with all tag references:

```
git push --tags
```

4. Open a pull request titled `"Release version X.X.X"`
5. Wait for all CI checks to pass.
6. A ridgeplot main contributor should sign off and merge this pull requests.
7. Create a new release using the GitHub UI.
   - Copy the raw markdown section in `CHANGES.md` corresponding to this release and use it as the description of the GitHub Release.
   - Use the same `X.X.X` tag used in the release.
8. At this point a GitHub Actions workflow will be triggered which will build and publish new wheels to PyPI. Be sure to check whether all workflows passed successfully.

6.6 Code of Conduct

Please remember to read and follow our Code of Conduct.

INDEX