# Contributing

Thank you for contributing to plotly.py! We are actively looking for

diverse contributors, with diverse background and skills.

This guide start by a general description of the different ways to contribute

to plotly.py, then we explain some technical aspects of preparing your

contribution.

## Code of Conduct

Please check out the [Code of Conduct](CODE\_OF\_CONDUCT.md). Don't tl:dr; it,

but the general idea is to be nice.

## What are the different ways to contribute?

There are many ways to contribute to plotly.py. It helps to understand first

the structure of the code and of the repository.

- the `codegen` (package in `packages/python/plotly/codegen`): all the code

inside `plotly.graph\_objects` is generated from the plotly javascript API

(the "schema"). The `codegen` package is where the code generation is done.

Most of the codegen code concerns the generation of docstrings. Traces and

Layout classes have a direct correspondence with their Javascript

counterpart. Additional methods are defined for the `Figure` object, such as

`update\_layout`, `add\_trace`, etc.

- the `plotly.express` package (usually imported as `px`) is a high-level

functional API. Its code is in `packages/python/plotly/express`. Most

functions of `plotly.express` call the internal `\_make\_figure` function

in `\_core.py`. More generally, the internals of `px` consist of general

functions taking care of building the figure (defining subplots, traces

or frames, for example), with special cases for different traces handled

within these functions. There is also subsequent code reuse for `px`

docstrings, in particular for documenting parameters.

- the `plotly.figure\_factory` module provides Python "recipes" for building

advanced visualizations, such as Gantt charts, annotated heatmaps, etc.

Figure factories are one of the easiest entry points into plotly.py, since

they consist of Python-only code, with standalone, well-separated functions.

However, please note that some of the figure factories become less relevant

as we are introducing more features into `plotly.express`. Some issues in the

tracker are labeled "figure\_factory" and can be good issues to work on. More

instructions on figure factories are found

[here](packages/python/plotly/plotly/figure\_factory/README.md).

- other pure-Python submodules are: `plotly.io` (low-level interface for

displaying, reading and writing figures), `plotly.subplots` (helper function

for layout of multi-plot figures)

- tests are found in `packages/python/plotly/plotly/tests`. Different

directories correspond to different test jobs (with different dependency sets)

run in continuous integration. These jobs are configured in

`packages/python/plotly/tox.ini`, which itself is used in the Circle CI

configuration file `.circleci/config.yml`. More is explained about tests

in the following "Technical aspects" section.

- the \*\*documentation\*\* is part of this repository. Its structure and some

explanations are described [here](doc/README.md). The documentation, in

particular example-based tutorials, is a great place to start contributing.

The contribution process is also more lightweight, since you can modify

tutorial notebooks without setting up an environment, etc.

We maintain a wishlist of examples to add on

https://github.com/plotly/plotly.py/issues/1965. If you have writing skills,

the wording of existing examples can also be improved in places.

Contributing code or documentation is not the only way to contribute! You can

also contribute to the project by

- reporting bugs (see below).

- submitting feature requests (maybe we'll convince you to contribute it as a

pull request!).

- helping other users on the [community forum](https://community.plot.ly/).

Join the list of [nice people](https://community.plot.ly/u) helping other

plotly users :-).

We also recommend reading the great

[how to contribute to open source](https://opensource.guide/how-to-contribute/)

guide.

## Have a Bug Report?

Open an issue! Go to https://github.com/plotly/plotly.py/issues. It's possible that your issue was already addressed. If it wasn't, open it. We also accept pull requests; take a look at the steps below for instructions on how to do this.

## Have Questions about Plotly?

Check out our Community Forum: https://community.plot.ly/.

## Want to improve the plotly documentation?

Thank you! Instructions on how to contribute to the documentation are given [here](doc/README.md). Please also read the next section if you need to setup a development environment.

## How to contribute - Technical Aspects

Below we explain the technical aspects of contributing. It is not strictly necessary to follow all points (for example, you will not write tests when writing documentation, most of the time), but we want to make sure that you know how to deal with most cases.

Note that if you are modifying a single documentation page, you can do it

directly on Github by clicking on the "Edit this page on GitHub" link, without

cloning the repository.

## Setup a Development Environment

### Fork, Clone, Setup Your Version of the Plotly Python API

First, you'll need to \*get\* our project. This is the appropriate \*clone\* command (if you're unfamiliar with this process, https://help.github.com/articles/fork-a-repo):

\*\*DO THIS (in the directory where you want the repo to live)\*\*

```bash

git clone https://github.com/your\_github\_username/plotly.py.git

cd plotly.py

```

Note: if you're just getting started with git, there exist great resources to

learn and become confident about git, like http://try.github.io/.

### Create a virtual environment for plotly development

You can use either [conda][conda-env] or [virtualenv][virtualenv] to create a virtual environment for plotly development, e.g.

```bash

conda create -n plotly-dev python

conda activate plotly-dev

```

[conda-env]: https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#creating-an-environment-with-commands

[virtualenv]: http://docs.python-guide.org/en/latest/dev/virtualenvs/

### Install requirements - (Non-Windows)

```bash

(plotly\_dev) $ pip install -r packages/python/plotly/requirements.txt

(plotly\_dev) $ pip install -r packages/python/plotly/optional-requirements.txt

```

### Install requirements - (Windows + Conda)

Because Windows requires Visual Studio libraries to compile some of the optional dependencies, follow these steps to

complete installation and avoid gdal-config errors.

```bash

(plotly\_dev) $ pip install -r packages/python/plotly/requirements.txt

(plotly\_dev) $ conda install fiona

(plotly\_dev) $ pip install -r packages/python/plotly/optional-requirements.txt

```

### Editable install of plotly packages

```bash

(plotly\_dev) $ pip install -e packages/python/plotly/

(plotly\_dev) $ pip install -e packages/python/chart-studio/

(plotly\_dev) $ pip install -e packages/python/plotly-geo/

```

This will ensure that the installed packages links to your local development

directory, meaning that all changes you make reflect directly in your

environment (don't forget to restart the Jupyter kernel though!). For more

information see the

[`setuptools`](https://setuptools.readthedocs.io/en/latest/setuptools.html#development-mode)

and

[`pip`](https://pip.pypa.io/en/stable/reference/pip\_install/#install-editable)

documentation on \_development mode\_.

### ipywidgets development install

Run the following commands in your virtual environment to use the

development version of `FigureWidget`,

```bash

(plotly\_dev) $ jupyter nbextension enable --py widgetsnbextension

(plotly\_dev) $ jupyter nbextension install --py --symlink --sys-prefix plotlywidget

(plotly\_dev) $ jupyter nbextension enable --py --sys-prefix plotlywidget

```

To make plotly plots show up in JupyterLab, you also need to [install the plotly jupyterlab extensions][plotly-jl].

[plotly-jl]: https://plot.ly/python/getting-started/#jupyterlab-support-python-35

### Configure black code formatting

This repo uses the [Black](https://black.readthedocs.io/en/stable/) code formatter,

and the [pre-commit](https://pre-commit.com/) library to manage a git commit hook to

run Black prior to each commit. Both pre-commit and black are included in the

`packages/python/plotly/optional-requirements.txt` file, so you should have them

installed already if you've been following along.

To enable the Black formatting git hook, run the following from within your virtual

environment.

```bash

(plotly\_dev) $ pre-commit install

```

Now, whenever you perform a commit, the Black formatter will run. If the formatter

makes no changes, then the commit will proceed. But if the formatter does make changes,

then the commit will abort. To proceed, stage the files that the formatter

modified and commit again.

If you don't want to use `pre-commit`, then you can run black manually prior to making

a PR as follows.

```bash

(plotly\_dev) $ black .

```

### Making a Development Branch

Third, \*don't\* work in the `master` branch. As soon as you get your master branch ready, run:

\*\*DO THIS (but change the branch name)\*\*

```bash

git checkout -b my-dev-branch

```

... where you should give your branch a more descriptive name than `my-dev-branch`

### Pull Request When Ready

Once you've made your changes (and hopefully written some tests, see below for more about testing...),

make that pull request!

## Update to a new version of Plotly.js

First update the version of the `plotly.js` dependency in `packages/javascript/plotlywidget/package.json`.

Then run the `updateplotlyjs` command with:

```bash

$ cd packages/python/plotly

$ python setup.py updateplotlyjs

```

This will download new versions of `plot-schema.json` and `plotly.min.js` from

the `plotly/plotly.js` GitHub repository (and place them in

`plotly/package\_data`). It will then regenerate all of the `graph\_objs`

classes based on the new schema.

For dev branches, it is also possible to use `updateplotlyjsdev --devrepo reponame --devbranch branchname` to update to development versions of `plotly.js`. This will fetch the `plotly.js` in the CircleCI artifact of the branch `branchname` of the repo `reponame`. If `--devrepo` or `--devbranch` are omitted, `updateplotlyjsdev` defaults using `plotly/plotly.js` and `master` respectively. For example, to update to a version from a pull request to the `plotly/plotly.js` repo that is numbered 555, run:

```bash

$ cd packages/python/plotly

$ python setup.py updateplotlyjsdev --devbranch pull/555

```

## Testing

We take advantage of two tools to run tests:

\* [`tox`](https://tox.readthedocs.io/en/latest/), which is both a virtualenv management and test tool.

\* [`pytest`](https://docs.pytest.org/en/latest/), a powerful framework for unit testing.

### Running Tests with `pytest`

Since our tests cover \*all\* the functionality, to prevent tons of errors from showing up and having to parse through a messy output, you'll need to install `optional-requirements.txt` as explained above.

After you've done that, go ahead and run the test suite!

```bash

pytest packages/python/plotly/plotly/tests/

```

Or for more \*verbose\* output:

```bash

pytest -v packages/python/plotly/plotly/tests/

```

Either of those will run \*every\* test we've written for the Python API. You can get more granular by running something like:

```bash

pytest packages/python/plotly/plotly/tests/test\_core/

```

... or even more granular by running something like:

```bash

pytest plotly/tests/test\_plotly/test\_plot.py

```

or for a specfic test function

```bash

pytest plotly/tests/test\_plotly/test\_plot.py::test\_function

```

### Running tests with `tox`

Running tests with tox is much more powerful, but requires a bit more setup.

You'll need to export an environment variable for \*each\* tox environment you wish to test with. For example, if you want to test with `Python 2.7` and

`Python 3.6`, but only care to check the `core` specs, you would need to ensure that the following variables are exported:

```

export PLOTLY\_TOX\_PYTHON\_27=<python binary>

export PLOTLY\_TOX\_PYTHON\_36=<python binary>

```

Where the `<python binary` is going to be specific to your development setup. As a more complete example, you might have this loaded in a `.bash\_profile` (or equivalent shell loader):

```bash

############

# tox envs #

############

export PLOTLY\_TOX\_PYTHON\_27=python2.7

export PLOTLY\_TOX\_PYTHON\_34=python3.4

export TOXENV=py27-core,py34-core

```

Where `TOXENV` is the environment list you want to use when invoking `tox` from the command line. Note that the `PLOTLY\_TOX\_\*` pattern is used to pass in variables for use in the `tox.ini` file. Though this is a little setup, intensive, you'll get the following benefits:

\* `tox` will automatically manage a virtual env for each environment you want to test in.

\* You only have to run `tox` and know that the module is working in both `Python 2` and `Python 3`.

Finally, `tox` allows you to pass in additional command line arguments that are formatted in (by us) in the `tox.ini` file, see `{posargs}`. This is setup to help with our configuration of [pytest markers](http://doc.pytest.org/en/latest/example/markers.html), which are set up in `packages/python/plotly/pytest.ini`. To run only tests that are \*not\* tagged with `nodev`, you could use the following command:

```bash

tox -- -a '!nodev'

```

Note that anything after `--` is substituted in for `{posargs}` in the tox.ini. For completeness, because it's reasonably confusing, if you want to force a match for \*multiple\* `pytest` marker tags, you comma-separate the tags like so:

```bash

tox -- -a '!nodev','!matplotlib'

```

### Writing Tests

You're \*strongly\* encouraged to write tests that check your added functionality.

When you write a new test anywhere under the `tests` directory, if your PR gets accepted, that test will run in a virtual machine to ensure that future changes don't break your contributions!

Test accounts include: `PythonTest`, `PlotlyImageTest`, and `PlotlyStageTest`.