# Contents

1 Documentation ............................................. 3
  1.1 About psyplot ........................................... 3
    1.1.1 Why psyplot? ...................................... 3
    1.1.2 About the author .................................. 3
    1.1.3 License ............................................ 4
  1.2 Installation ............................................ 4
    1.2.1 How to install ..................................... 4
      Installation using conda ................................ 4
      Installation using pip .................................. 4
      Installation from source ................................. 4
    1.2.2 Dependencies ...................................... 5
      Required dependencies .................................. 5
      Optional dependencies .................................. 5
    1.2.3 Running the tests .................................. 5
    1.2.4 Building the docs .................................. 5
    1.2.5 Uninstallation ...................................... 6
      Uninstallation via conda ................................. 6
      Uninstallation via pip .................................. 6
  1.3 Getting started ........................................ 7
    1.3.1 Initialization and interactive usage ................. 7
    1.3.2 Choosing the dimension ............................. 11
    1.3.3 Configuring the appearance of the plot .............. 12
    1.3.4 Controlling the update ............................. 15
      Automatic update ...................................... 16
      Direct control on formatoption update .................. 16
    1.3.5 Creating and managing multiple plots ................. 16
      Creating multiple plots ................................ 16
      Slicing and filtering the project ....................... 18
    1.3.6 Saving and loading your project ...................... 19
    1.3.7 Adding your own script: The post formatoption ...... 20
  1.4 Configuration .......................................... 20
    1.4.1 The rcParams ...................................... 20
    1.4.2 Default formatoptions ............................... 22
  1.5 Subprojects ............................................ 23
    1.6 xarray Accessors ....................................... 23
      1.6.1 The DatasetAccessor dataset accessor ............... 23
      1.6.2 The InteractiveArray dataarray accessor .......... 24
    1.7 Psyplot plugins ....................................... 28
1.12 API Reference .......................................................... 70
1.12.1 Subpackages ......................................................... 70
   psyplot.compat package ............................................. 70
   Submodules .......................................................... 70
   psyplot.compat.pycompat module ................................. 70
   psyplot.config package .......................................... 70
   Submodules .......................................................... 71

1.11 Contributing to psyplot ........................................... 64
1.11.1 Code of Conduct ................................................ 65
1.11.2 What should I know before I get started? .................... 65
   The psyplot framework .......................................... 65
1.11.3 How Can I Contribute? ......................................... 66
   Reporting Bugs .................................................... 66
   How Do I Submit A (Good) Bug Report? .......................... 66
   Suggesting Enhancements ....................................... 66
   How Do I Submit A (Good) Enhancement Suggestion? .......... 67
   Pull Requests ..................................................... 67
   Adding new examples ............................................. 67
1.11.4 Styleguides ....................................................... 67
   Git Commit Messages ............................................ 67
   Documentation Styleguide ..................................... 68
   Example ............................................................ 68

1.10 Developers guide .................................................. 49
1.10.1 The psyplot framework ........................................ 50
   The project() function ......................................... 51
   The InteractiveBase and the Plotter classes .................... 54
      Interactive data objects ...................................... 54
      Visualization objects ....................................... 55
   Formatoptions .................................................... 57
1.10.2 How to implement your own plotters and plugins .......... 57
   Creating plotters ................................................ 57
   Interface for the plotter ....................................... 58
      Interface to the data ......................................... 59
      Interfacing to other formatoptions .......................... 59
      First solution: Use a dict ................................... 59
      Second solution: Interact with other formatoptions ......... 60
   Creating new plugins ............................................ 62
   rcParams handling in plugins .................................. 63

1.9 Example Gallery .................................................... 38
1.9.1 Usage of Climate Data Operators .............................. 39
1.9.2 Sharing formatoptions ........................................ 40
1.9.3 Applying your own post processing ............................ 43
   Usage .............................................................. 43
   Timing .............................................................. 47
1.9.4 Saving and loading .............................................. 48

1.8 Command line usage ............................................... 36
1.8.1 Positional Arguments .......................................... 36
1.8.2 Named Arguments ............................................. 36
1.8.3 Info options .................................................. 37
1.8.4 Output options ............................................... 37
1.8.1 Positional Arguments .......................................... 36
1.8.2 Named Arguments ............................................. 36
1.8.3 Info options .................................................. 37
1.8.4 Output options ............................................... 37

1.7 Command line usage ............................................... 29
1.7.1 Existing plugins ............................................... 29
1.7.2 How to exclude plugins ....................................... 35
   The PSYPLOT_PLUGINS environment variable ................ 35
   The PSYPLOT_PLOTMETHODS environment variable ......... 35

1.6 Project() function ................................................ 22
1.6.1 InteractiveBase ............................................... 24
1.6.2 Plotter classes ................................................ 24
1.6.3 Interactive data objects ..................................... 25
1.6.4 Visualization objects ........................................ 27

1.1 Introduction ........................................................ 4
1.1.1 What is psyplot? .............................................. 4
1.1.2 What’s in a psyplot plot? ..................................... 4
1.1.3 What’s in a psyplot data object? .............................. 4
1.1.4 What’s in a psyplot visualization object? ..................... 5
1.1.5 What’s in a data operator? ................................... 5
1.1.6 What’s in a post processing statement? ...................... 5
1.1.7 What’s in a formatoption? .................................... 6
1.1.8 What’s in a parameter? ....................................... 7
1.1.9 What’s in a formatoption with a default? .................... 7
1.1.10 What’s in a formatoption with a default value? .......... 9
1.1.11 What’s in a formatoption key? ............................... 10
1.1.12 What’s in a formatoption value? ............................. 10

1.5 Using psyplot ........................................................ 2
1.5.1 How to install psyplot ......................................... 2
1.5.2 How to test psyplot ............................................ 2
1.5.3 How to run psyplot ............................................ 2
1.5.4 How to use psyplot ............................................ 3
Welcome! Looking for a fast and flexible visualization software? Here we present psyplot, an open source python project that mainly combines the plotting utilities of matplotlib and the data management of the xarray package and integrates them into a software that can be used via command-line and via a GUI!

The main purpose is to have a framework that allows a fast, attractive, flexible, easily applicable, easily reproducible and especially an interactive visualization of your data.

The ultimate goal is to help scientists in their daily work by providing a flexible visualization tool that can be enhanced by their own visualization scripts. psyplot can be used via command line and with the graphical user interface (GUI) from the psyplot-gui module.

If you want more motivation: Have a look into the About psyplot section.

The package is very new and there are many features that will be included in the future. So we are very pleased for feedback! Please simply raise an issue on GitHub.
1.1 About psyplot

1.1.1 Why psyplot?

When visualizing data, one always has to choose:

- Either create the plot with an intuitive graphical user interface (GUI) (e.g. panoply) but less options for customization and difficult to script
- or create the plot from the command line, e.g. via NCL, R or python with more possibilities for customization and scripting but also less intuitive

psyplot wants to combine these two worlds: create a well-documented and easy accessible framework to visualize data from a GUI and the command line (and of course through a script).

There exists nothing like that. Of course you can also work with software like Paraview via the built-in python shell. But, if you really want to explore your data it is totally not straightforward to access and explore it from within such a software using numeric functions from numpy, scipy, etc.

Therefore I developed this modular framework that can create and customize plots efficiently with short and comprehensive commands, that can be accessed through a GUI (see Subprojects) and where you have always a comprehensive API to access your data.

Different from the usual use with matplotlib, which in the end most of the time results in copy-pasting parts of your code, this software is build on the don’t repeat yourself principle. Each of the small parts that make up a visualization, whether it is part of the data evaluation or of the appearance of the plot, psyplot puts it into a format option can be reused when it is needed.

Nevertheless, it’s again a new piece of software. Therefore, if you want to use it, for sure you need a bit of time to get comfortable with the framework. I promise to you, it’s worth it. So get started and please let me know if you have a different opinion.

1.1.2 About the author

I, (Philipp Sommer), work as a PhD student for climate modeling in the Atmosphere-Regolith-Vegetation (ARVE) group in the Institute of Earth Surface Dynamics (IDYST) at the University of Lausanne. During my time at the Max-Planck-Institute for Meteorology I worked a lot with the Max-Planck-Institute Earth-System-Model (MPI-ESM) and the ICON model in the working group on Terrestrial Hydrology of Stefan Hagemann. This included a lot of evaluation of climate model data. It finally gave the motivation for the visualization framework psyplot.
1.3 License

psyplot is published under the GNU General Public License v2.0

1.2 Installation

1.2.1 How to install

There basically three different methodologies for the installation. You should choose the one, which is the most appropriate solution concerning your skills and your usage:

The recommended installation  We recommend to use anaconda for installing python and psyplot (see Installation using conda). If you however already have python installed on your system, you can also use pip (see Installation using pip).

The developer installation  Install it from source (see Installation from source)

Installation using conda

We highly recommend to use conda for installing psyplot. After having downloaded the installer from anaconda or miniconda, you can install psyplot and the optional plugins (see Optional dependencies) via:

```
$ conda install -c conda-forge psy-maps psyplot-gui psy-reg netCDF4
```

If you only want to install the core, i.e. the raw framework, run:

```
$ conda install -c conda-forge psyplot
```

If you want to be able to read GeoTiff Raster files, you will need to have gdal installed:

```
$ conda install gdal
```

Please also visit the xarray installation notes for more informations on how to best configure the xarray package for your needs.

Installation using pip

If you do not want to use conda for managing your python packages, you can also use the python package manager pip and install via:

```
$ pip install psyplot
```

However to be on the safe side, make sure you have the Dependencies installed.

Installation from source

To install it from source, make sure you have the Dependencies installed, clone the github repository via:

```
git clone https://github.com/Chilipp/psyplot.git
```

and install it via:
1.2.2 Dependencies

Required dependencies

Psyplot supports officially python 3.6 and 3.7. Previous versions are also available for python 2.7. Furthermore the package is built upon multiple other packages, mainly

- xarray>=0.8: Is used for the data management in the psyplot package
- matplotlib>=1.4.3<3.1: The python visualization package
- PyYAML: Needed for the configuration of psyplot
- docrep: A package for efficient documentation processing for large APIs
- funcargparse: A package to create command line parsers from function docstrings

Optional dependencies

We furthermore recommend to use

- psyplot-gui: A graphical user interface to psyplot
- psy-simple: A psyplot plugin to make simple plots
- psy-maps: A psyplot plugin for visualizing data on a map
- psy-reg: A psyplot plugin for visualizing fits to your data
- cdo: The python bindings for cdos (see also the cdo example)

1.2.3 Running the tests

We use pytest to run our tests. So you can either run clone out the github repository and run:

```bash
$ python setup.py test
```

or install pytest by yourself and run:

```bash
$ py.test
```

To also test the plugin functionality, install the psyplot_test module in tests/test_plugin via:

```bash
$ cd tests/test_plugin && python setup.py install
```

and run the tests via one of the above mentioned commands.

1.2.4 Building the docs

To build the docs, check out the github repository and install the requirements in 'docs/environment.yml'. The easiest way to do this is via anaconda by typing:

```bash
$ python setup.py install
```
Then build the docs via:

```bash
$ cd docs
$ make html
```

**Note:** The building of the docs always preprocesses the examples. You might want to disable that by setting `process_examples = False`. Otherwise please note that the examples are written as python3 notebooks, hence you may have to install a python3 kernel through `ipykernel`. Just create a new environment `py37` and install it via:

```bash
conda create -n py37 python=3.7
source activate py37
conda install notebook ipykernel
ipython kernel install --user
```

You then have to install the necessary modules for each of the examples in the new `py37` environment.

### 1.2.5 Uninstallation

The uninstallation depends on the system you used to install psyplot. Either you did it via *conda* (see *Uninstallation via conda*), via *pip* or from the *source files* (see *Uninstallation via pip*).

Anyway, if you may want to remove the psyplot configuration files. If you did not specify anything else (see `psyplot.config.rcsetup.psyplot_fname()`), the configuration files for psyplot are located in the user home directory. Under linux and OSX, this is `$HOME/.config/psyplot`. On other platforms it is in the .psyplot directory in the user home.

**Uninstallation via conda**

If you installed psyplot via *conda*, simply run:

```bash
conda remove psyplot
```

If you however installed it via a preconfigured environment (see install-conda-env), you can simply remove the entire virtual environment via:

```bash
conda env remove -n psyplot
```

**Uninstallation via pip**

Uninstalling via pip simply goes via:

```bash
pip uninstall psyplot
```

Note, however, that you should use *conda* if you also installed it via conda.
1.3 Getting started

1.3.1 Initialization and interactive usage

This section shall introduce you how to read data from a netCDF file and visualize it via psyplot. For this, you need to have netCDF4 and the psy-maps psyplot plugin to be installed (see Installation).

Furthermore we use the demo.nc netCDF file for our demonstrations.

Note: We recommend to either run this example using our GUI. However, you can also either use IPython from the terminal via

```bash
conda install ipython  # or pip install ipython
ipython  # starts the ipython console
```

and copy-paste the commands in this example, or you use a jupyter notebook via

```bash
conda install jupyter  # or pip install jupyter
jupyter notebook  # starts the notebook server
```

Then create a new notebook in the desired location and copy-paste the examples below. If you want, we also recommend to include the following commands in the notebook

```python
import psyplot.project as psy
# show the figures inline in the notebook and not in a separate window
%matplotlib inline
# don't close the figures after showing them, because than the update
# would not work
%config InlineBackend.close_fi```ures = False
# show the figures after they are drawn or updated. This is useful
# for the visualization in the jupyter notebook
psy.rcParams['auto_show'] = True
```

After you installed psyplot, you can import the package via

```python
In [1]: import psyplot
```

Psyplot has several modules and subpackages. The main module for the use of psyplot is the project module.

```python
In [2]: import psyplot.project as psy
```

Plots can be created using the attributes of the plot instance of the ProjectPlotter.

Each new plugin defines several plot methods. In case of the psy-maps package, those are

```python
In [3]: psy.plot.show_plot_methods()
barplot
    Make a bar plot of one-dimensional data
combined
    Plot a 2D scalar field with an overlying vector field
density
    Make a density plot of point data
fldmean
    Calculate and plot the mean over x- and y-dimensions
lineplot
```

(continues on next page)
So to create a simple 2D plot of the temperature field 't2m', you can type

```
In [4]: p = psy.plot.mapplot('demo.nc', name='t2m')
```

![Image of a map plot with temperature data]

**Note:** If you’re not using the GUI, you have to call the `show()` method to display the plot, i.e. just run

```
p.show()
```

Now you created your first project

```
In [5]: p
Out[5]: psyplot.project.Project(
  arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00)
```

which contains the `xarray.DataArray` that stores the data and the corresponding plotter that visualizes it

```
In [6]: p[0]
Out[6]: <xarray.DataArray 't2m' (lat: 96, lon: 192)>
array([[251.41689, 251.454, 251.48915, ..., 251.29774, 251.33876, 251.37978],
       ...])
```
The visualization and data handling within the psyplot framework is designed to be as easy, flexible and interactive as possible. The appearance of a plot is controlled by the formatoptions of the plotter. In our case, they are the following:

```
In [8]: p.keys()
```

they can be investigated through the `Project.keys()`, `summarizes()` and `docs()`, or the corresponding low level methods of the `Plotter` class, `show_keys()`, `show_summaries()` and `show_docs()`.

1.3. Getting started
Updating a formatoption is straight forward. Each formatoption accepts a certain type of data. Let’s say, we want to have a different projection. Then we can look at the types this formatoption accepts using the `Project.docs()` method.

```python
In [9]: p.docs('projection')
projection
=======
Specify the projection for the plot

This formatoption defines the projection of the plot

Possible types
-------------
cartopy.crs.CRS
  A cartopy projection instance (e.g. :class:`cartopy.crs.PlateCarree`)
str
  A string specifies the projection instance to use. The centered longitude and latitude are determined by the :attr:`clon` and :attr:`clat` formatoptions.
Possible strings are (each standing for the specified projection)

+---------------------+----------------------------------+
| cyl                 | :class:`cartopy.crs.PlateCarree` |
| robin               | :class:`cartopy.crs.Robinson`    |
| moll                | :class:`cartopy.crs.Mollweide`   |
| geo                 | :class:`cartopy.crs.Geostationary` |
| northpole           | :class:`cartopy.crs.NorthPolarStereo` |
| southpole           | :class:`cartopy.crs.SouthPolarStereo` |
| ortho               | :class:`cartopy.crs.Orthographic` |
| stereo              | :class:`cartopy.crs.Stereographic` |
| near                | :class:`cartopy.crs.NearsidedPerspective` |
+---------------------+----------------------------------+
```

Warnings
--------
An update of the projection clears the axes

Let’s use an orthogonal projection. The update goes via the `Project.update()` method which goes all the way down to the `psyplot.plotter.Plotter.update()` and the `psy_maps.plotters.Projection.update()` method of the formatoption.

```python
In [10]: p.update(projection='ortho')
```
Our framework also let’s us update the dimensions of the data we show. For example, if we want to display the field for February, we can type

```python
# currently we are displaying january
In [11]: p[0].time.values
Out[11]: numpy.datetime64('1979-01-31T18:00:00.000000000')

In [12]: p.update(time='1979-02', method='nearest')
# now its february
In [13]: p[0].time.values
Out[13]: numpy.datetime64('1979-02-28T18:00:00.000000000')
```

which is in our case equivalent for choosing the second index in our time coordinate via

```python
In [14]: p.update(time=1)
```

So far for the first quick introduction. If you are interested you are welcomed to visit our example galleries or continue with this guide.

In the end, don’t forget to close the project in order to delete the data from the memory and close the figures

```python
In [15]: p.close(True, True, True)
```

### 1.3.2 Choosing the dimension

As you saw already above, the scalar variable 't2m' has multiple time steps and we can control what is shown via the `update()` method. By default, the `mapplot()` plot method chooses the first time step and the first vertical level (if those dimensions exist).

However, you can also specify the exact data slice for your visualization based upon the dimensions in you dataset. When doing that, you basically do not have to care about the exact dimension names in the netCDF files, because those
are decoded following the CF Conventions. Hence each of the above dimensions are assigned to one of the general dimensions 't' (time), 'z' (vertical dimension), 'y' (horizontal North-South dimension) and 'x' (horizontal East-West dimension). In our demo file, the dimensions are therefore decoded as 'time' \( \rightarrow 't', 'lev' \rightarrow 'z', 'lon' \rightarrow 'x', 'lat' \rightarrow 'y'.

Hence it is equivalent if you type

```python
In [16]: psy.plot.mapplot('demo.nc', name='t2m', t=1)
Out[16]: psyplot.project.Project([ arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00])
```

or

```python
In [17]: psy.plot.mapplot('demo.nc', name='t2m', time=1)
Out[17]: psyplot.project.Project([ arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00])
```

Finally you can also be very specific using the `dims` keyword via

```python
In [18]: psy.plot.mapplot('demo.nc', name='t2m', dims={'time': 1})
Out[18]: psyplot.project.Project([ arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00])
```

You can also use the `method` keyword from the plotting function to use the advantages of the `xarray.DataArray.sel()` method. E.g. to plot the data corresponding to March 1979 you can use

```python
In [19]: psy.plot.mapplot('demo.nc', name='t2m', t='1979-03', method='nearest', z=100000)
```

**Note:** If your netCDF file does (for whatever reason) not follow the CF Conventions, we interprete the last dimension as the x-dimension, the second last dimension (if existent) as the y-dimension, the third last dimension as the z-dimension. The time dimension however has to have the name 'time'. If that still does not fit your netCDF files, you can specify the correct names in the `rcParams`, namely

```python
In [20]: psy.rcParams.find_all('decoder.(x|y|z|t)')
Out[20]: RcParams({'decoder.t': {'time'},
                'decoder.x': set(),
                'decoder.y': set(),
                'decoder.z': set()})
```

### 1.3.3 Configuring the appearance of the plot

psyplot is build upon the great and extensive features of the matplotlib package. Hence, our framework can in principle be seen as a high-level interface to the matplotlib functionalities. However you can always access the basic matplotlib objects like figures and axes if you need.

In the psyplot framework, the communication to matplotlib is done via `formatoptions` that control the appearance of a plot. Each plot method (i.e. each attribute of `psyplot.project.plot`) has several a set of them and they set up the corresponding plotter.
Formatoptions are all designed for an interactive usage and can usually be controlled with very simple commands. They range from simple formatoptions like choosing the title to choosing the latitude-longitude box of the data.

The formatoptions depend on the specific plotting method and can be seen via the methods

| keys(*args, **kwargs) | Classmethod to return a nice looking table with the given formatoptions |
| summaries(*args, **kwargs) | Method to print the summaries of the formatoptions |
| docs(*args, **kwargs) | Method to print the full documentations of the formatoptions |

For example to look at the formatoptions of the `mapplot` method in an interactive session, type

```
In [21]: psy.plot.mapplot.keys(grouped=True) # to see the fmt keys
```

---

### Color coding formatoptions

<table>
<thead>
<tr>
<th>bounds</th>
<th>cbar</th>
<th>cbar spacing</th>
<th>cmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctickprops</td>
<td>cticksize</td>
<td>ctickweight</td>
<td>extend</td>
</tr>
</tbody>
</table>

---

### Label formatoptions

<table>
<thead>
<tr>
<th>clabel</th>
<th>clabelprops</th>
<th>clabelsize</th>
<th>clabelweight</th>
</tr>
</thead>
<tbody>
<tr>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
</tr>
</tbody>
</table>

---

### Miscellaneous formatoptions

<table>
<thead>
<tr>
<th>clat</th>
<th>clip</th>
<th>clon</th>
<th>datagrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid_color</td>
<td>grid_labels</td>
<td>grid_labelsize</td>
<td>grid_settings</td>
</tr>
<tr>
<td>interp_bounds</td>
<td>lonlatbox</td>
<td>lsm</td>
<td>map_extent</td>
</tr>
<tr>
<td>projection</td>
<td>stock_img</td>
<td>transform</td>
<td>xgrid</td>
</tr>
<tr>
<td>ygrid</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Axis tick formatoptions

---
**psyplot Documentation, Release 1.2.1**

### Masking formatoptions

<table>
<thead>
<tr>
<th>maskbetween</th>
<th>maskgeq</th>
<th>maskgreater</th>
<th>maskleq</th>
<th>maskless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Plot fmt options

<table>
<thead>
<tr>
<th>plot</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

### Post processing fmt options

<table>
<thead>
<tr>
<th>post</th>
<th>post_timing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Axes fmt options

<table>
<thead>
<tr>
<th>tight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

In [22]: psy.plot.mapplot.summaries(['title', 'cbar'])  # to see the fmt summaries

Show the title

cbar Specify the position of the colorbars

In [23]: psy.plot.mapplot.docs('title')  # to see the full fmt docs

====

Show the title

Set the title of the plot.

You can insert any meta key from the :attr:`xarray.DataArray.attrs` via a string like ```%(key)s``` . Furthermore there are some special cases:

- Strings like ```'%Y'``` , ```'%b'``` , etc. will be replaced using the
  :meth:`datetime.datetime.strftime` method as long as the data has a time
  coordinate and this can be converted to a :class:`~datetime.datetime`
  object.
- ```'%(x)s'``` , ```'%(y)s'``` , ```'%(z)s'``` , ```'%(t)s'``` will be replaced
  by the value of the x-, y-, z- or time coordinate (as long as this
  coordinate is one-dimensional in the data)
- any attribute of one of the above coordinates is inserted via
psyplot Documentation, Release 1.2.1

``axis + key`` (e.g. the name of the x-coordinate can be inserted via 
``'%(xname)s'``).
- Labels defined in the :class:`psyplot.rcParams` ``'texts.labels'`` key are also replaced when enclosed by '{}'. The standard labels are
  - tinfo: ``'%H:%M'``
  - dtinfo: ``'%B %d, %Y. %H:%M'``
  - dinfo: ``'B %d, %Y'``
  - desc: ``'(%(long_name)s [%(units)s]''
  - sdesc: ``'(%(name)s [%(units)s]''

Possible types
--------------
str
   The title for the :func:`~matplotlib.pyplot.title` function.

Notes
-----
This is the title of this specific subplot! For the title of the whole figure, see the :attr:`figtitle` formatoption.

See Also
--------
figtitle, titlesize, titleweight, titleprops

But of course you can also use the online documentation of the method your interested in.

To include a formatoption from the beginning, you can simply pass in the key and the desired value as keyword argument, e.g.

```python
In [24]: psy.plot.mapplot('demo.nc', name='t2m', title='my title',
   cbar='r')
```

```python
Out[24]: psyplot.project.Project([ arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00])
```

This works generally well as long as there are no dimensions in the desired data with the same name as one of the passed in formatoptions. If you want to be really sure, use the `fmt` keyword via

```python
In [25]: psy.plot.mapplot('demo.nc', name='t2m', fmt={'title': 'my title', 'cbar': 'r'})
```

```python
Out[25]: psyplot.project.Project([ arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00])
```

The same methodology works for the interactive usage, i.e. you can use

```python
In [26]: p.update(title='my title', cbar='r')
```

```python
# or
In [27]: p.update(fmt={'title': 'my title', 'cbar': 'r'})
```

1.3.4 Controlling the update
Automatic update

By default, a call of the `update()` method forces an automatic update and redrawing of all the plots. There are however several ways to modify this behavior:

1. Changing the behavior of one single project
   
   1. in the initialization of a project using the `auto_update` keyword
      
      ```python
      In [28]: p = psy.plot.mapplot('demo.nc', name='t2m', auto_update=False)
      ```
   
   2. setting the `no_auto_update` attribute
      
      ```python
      In [29]: p.no_auto_update = True
      ```

2. Changing the default configuration in the 'lists.auto_update' key in the `rcParams`

   ```python
   In [30]: psy.rcParams['lists.auto_update'] = False
   ```

3. Using the `no_auto_update` attribute as a context manager

   ```python
   In [31]: with p.no_auto_update:
      ....:     p.update(title='test')
      ....:
   ```

If you disabled the automatical update via one of the above methods, you have to start the registered updates manually via

```python
In [32]: p.update(auto_update=True)
# or
In [33]: p.start_update()
```

Direct control on formatoption update

By default, when updating a formatoption, it is checked for each plot whether the formatoption would change during the update or not. If not, the formatoption is not updated. However, sometimes you may want to do that and for this, you can use the `force` keyword in the `update()` method.

1.3.5 Creating and managing multiple plots

Creating multiple plots

One major advantage of the psyplot framework is the systematic management of multiple plots at the same time. To create multiple plots, simply pass in a list of dimension values and/or names. For example

```python
In [34]: psy.plot.mapplot('demo.nc', name='t2m', time=[0, 1])
Out[34]: psyplot.project.Project(
   arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
   arr1: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00])
```
created two plots: one for the first and one for the second time step.

Furthermore

```
In [35]: psy.plot.mapplot('demo.nc', name=['t2m', 'u'], time=[0, 1])
Out[35]:
psyplot.project.Project({
    arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr1: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00,
    arr2: 2-dim DataArray of u, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr3: 2-dim DataArray of u, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00})
```

created four plots. By default, each plot is made in an own figure but you can also use the `ax` keyword to setup how the plots will be arranged. The `sort` keyword allows you to sort the plots.

As an example we plot the variables 't2m' and 'u' for the first and second time step into one figure and sort by time. This will produce

```
In [36]: psy.plot.mapplot(
    ....:     'demo.nc', name=['t2m', 'u'], time=[0, 1], ax=(2, 2), sort=['time'],
    ....:     title='%(long_name)s, %b')
Out[36]:
psyplot.project.Project({
    arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr1: 2-dim DataArray of u, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr2: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00,
    arr3: 2-dim DataArray of u, with (lat, lon)=(96, 192), lev=100000.0, time=1979-02-28T18:00:00})
```
**Warning:** As the xarray package, the slicing is based upon positional indexing with lists (see the xarray documentation on positional indexing). Hence you might think of choosing your data slice via `psy.plot.mapplot(...)`, `x=[1, 2, 3, 4, 5]`, ...). However this would result in 5 different plots! Instead you have to write `psy.plot.mapplot(..., x=[[1, 2, 3, 4, 5]], ...)`. The same is true for plotting methods like the `mapvector` method. Since this method needs two variables (one for the latitudinal and one for the longitudinal direction), typing

```python
In [37]: psy.plot.mapvector('demo.nc', name=['u', 'v'])
```

results in a `ValueError`. Instead you have to write

```python
In [38]: psy.plot.mapvector('demo.nc', name=['u', 'v'])
```

Please have a look into the documentations of the `mapvector` and `mapcombined` for getting examples on how to use this methods.

### Slicing and filtering the project

Managing a whole lot of plots is basically the same as managing a single plot. However, you can always get the single array and handle it separately.

You can either get it through the usual list slicing (the `Project` class actually is a simple list subclass) or you can use meta attributes, dimensions and the specific `arr_name` attribute. For the latter one, just call the project with your filtering attributes

This behavior is especially useful if you want to address only some arrays with your update. For example, let’s consider we want to choose a ‘winter’ colormap for the zonal wind variable and a colormap ranging from blue to red for the temperature. Then we could do this via

```python
In [39]: p(name='t2m').update(cmap='RdBu_r')
In [40]: p(name='u').update(cmap='winter')
```

**Note:** When doing so, we recommend to temporarily disable the automatic update because then the figure will only be drawn once and the update will be done in parallel.

Hence, it is better to use the context manager `no_auto_update` (see Automatic update)

```python
In [41]: with p.no_auto_update:
    p(name='t2m').update(cmap='RdBu_r')
    p(name='u').update(cmap='winter')
    p.start_update()
```

Finally you can access the plots created by a specific plotting method through the corresponding attribute in the `Project` class. In this case this is of course useless because all plots in maps were created by the same plotting method, but it may be helpful when having different plotters in one project (see The psyplot framework). Anyway, the plots created by the `mapplot` method could be accessed via

```python
In [42]: p.mapplot
```

(continues on next page)
1.3.6 Saving and loading your project

Within the psyplot framework, you can also save and restore your plots easily and flexibel.

To save your project, use the `save_project()` method:

```python
In [43]: p.save_project('my_project.pkl')
```

This saves the plot-settings into the file 'my_project.pkl', a simple pickle file that you could open by yourself using

```python
In [44]: import pickle
In [45]: with open('my_project.pkl', 'rb') as f:
    d = pickle.load(f)
```

In order to not avoid large project files, we do not store the data but only the filenames of the datasets. Hence, if you want to load the project again, make sure that the datasets are accessible through the path as they are listed in the `dsnames` attribute.

Otherwise you have several options to avoid wrong paths:

1. Use the `alternative_paths` parameter and provide for each filename a specific path when you save the project

```python
In [47]: p.dsnames
Out[47]: {'demo.nc'}
In [48]: p.save_project('test.pkl', alternative_paths={'demo.nc': 'other_path.nc'})
```

2. pack the whole data to the place where you want to store the project file

```python
In [49]: p.save_project('target-folder/test.pkl', pack=True)
```

3. specify where the datasets can be found when you load the project:

```python
In [50]: p = psy.Project.load_project('test.pkl', alternative_paths={'demo.nc': 'other_path.nc'})
```

4. Save the data in the pickle file, too

```python
In [51]: p.save_project('test.pkl', ds_description={'arr'})
```

To restore your project, simply use the `load_project()` method via
In [52]: maps = psy.Project.load_project('test.pkl')

Note: Saving a project stores the figure informations like axes positions, background colors, etc. However only the axes informations from from the axes within the project are stored. Other axes in the matplotlib figures are not considered and will not be restored. You can, however, use the alternative_axes keyword in the Project.load_project() method if you want to restore your settings and/or customize your plot with the post formatoption (see Adding your own script: The post formatoption)

1.3.7 Adding your own script: The post formatoption

Very likely, you will face the problem that not all your needs are satisfied by the formatoptions in one plotter. You then have two choices:

1. define your own plotter with new formatoptions (see How to implement your own plotters and plugins)

   Pros
   • more structured approach
   • you can enhance the plotter with other formatoptions afterwards and reuse it

   Cons
   • more complicated
   • you always have to ship the module where you define your plotter when you want to save and load your project
   • can get messy if you define a lot of different plotters

2. use the post formatoption

   Pros
   • fast and easy
   • easy to save and load

   Cons
   • may get complicated for large scripts
   • has to be enabled manually by the user

For most of the cases, the post formatoptions is probably what you are looking for (the first option is described in our developers guide).

This formatoption is designed for applying your own postprocessing script to your plot. It accepts a string that is executed using the built-in exec() function and is executed at the very end of the plotting. In this python script, the formatoption itself (and therefore the plotter and axes) can be accessed inside the script through the self variable. An example how to handle this formatoption can be found in our example gallery.

1.4 Configuration

1.4.1 The rcParams
Psyplot, and especially its plugins, have a lot of configuration values. Our rcParams handling is motivated by matplotlib although we extended the possibilities of its matplotlib.RcParams class. Our rcParams are stored in the psyplot.rcParams object. Without any plugins, this looks like

```python
In [1]: from psyplot import rcParams

In [2]: print(rcParams.dump(exclude_keys=[]))
# Configuration parameters of the psyplot module
# You can copy this file (or parts of it) to another path and save it as psyplotrc.yml. The directory should then be stored in the PSYPLOTCONFIGDIR environment variable.
# Created with python
# 3.7.3 (default, Mar 27 2019, 22:11:17)
# [GCC 7.3.0]

# Automatically draw the figures if the draw keyword in the update and start_update methods is None
auto_draw: true
# Automatically show the figures after the update and start_update methods
auto_show: false
# path for supplementary data
datapath: null
# interpolation method to calculate 2D-bounds (see the `kind` parameter in the :meth:`psyplot.data.CFDecoder.get_plotbounds` method)
decoder.interp_kind: linear
# names that shall be interpreted as the time dimension
decoder.t: !!set
  time: null
# names that shall be interpreted as the longitudinal x dim
decoder.x: !!set {}
# names that shall be interpreted as the latitudinal y dim
decoder.y: !!set {}
# names that shall be interpreted as the vertical z dim
decoder.z: !!set {}
# Boolean flag to control whether CDOs (Climate Data Operators) should be used to calculate grid weights. If None, they are tried to be used.
gridweights.use_cdo: null
# default value (boolean) for the auto_update parameter in the initialization of Plotter, Project, etc. instances
lists.auto_update: true
# formatoption keys and values that are defined by the user to be used by the specified plotters. For example to modify the title of all :class:`psyplot.plotter.maps.FieldPlotter` instances, set
defaults: {"plotter.fieldplotter.title": "my title"}
plotter.user: {}
# boolean controlling whether all plotters specified in the project.plotters item will be automatically imported when importing the psyplot.project module
project.auto_import: false
# boolean controlling whether the seaborn module shall be imported when importing the project module. If None, it is only tried to import the module.
```

(continues on next page)
You can use this object like a dictionary and modify the default values. For example, if you do not want, that the seaborn package is imported when the psyplot.project module is imported, you can simply do this via:

```
In [3]: rcParams['project.import_seaborn'] = False
```

Additionally, you can make these changes permanent. At every first import of the psyplot module, the rcParams are updated from a yaml configuration file. On Linux and OS X, this is stored under $HOME/.config/psyplot/psyplotrc.yml, under Windows it is stored at $HOME/.psyplot/psyplotrc.yml. But use the psyplot.config.rcsetup.psyplot_fname() function, to get the correct location.

To make our changes from above permanent, we could just do:

```
In [4]: import yaml
    ...: from psyplot.config.rcsetup import psyplot_fname
    ...:
In [5]: with open(psyplot_fname(if_exists=False), 'w') as f:
    ...:     yaml.dump({'project.import_seaborn': False}, f)
...:
# or we use the dump method
In [6]: rcParams.dump(psyplot_fname(if_exists=False),
    ...:     overwrite=True,  # update the existing file
    ...:     include_keys=['project.import_seaborn'])
...:
```

### 1.4.2 Default formatoptions

The psyplot plugins, (psy_simple.plugin, psy_maps.plugin, etc.) define their own rcParams instance. When the plugins are loaded at the first import of psyplot, these instances update psyplot.rcParams.

The update mainly defines the default values for the plotters defined by that plugin. However, it is not always obvious, which key in the psyplot.rcParams belongs to which formatoption. For this purpose, however, you can use the default_key attribute. For example, the title formatoption has the default_key

```
In [7]: import psyplot.project as psy
In [8]: plotter = psy.plot.lineplot.plotter_cls()
    ...: plotter.title.default_key
    ...:
Out[8]: 'plotter.baseplotter.title'
```

As our plotters are based on inheritance, the default values use it, too. Therefore, the FieldPlotter, the underlying plotter for the mapplot plot method, uses the same configuration value in the psyplot.rcParams:
1.5 Subprojects

psyplot is only the over-arching framework. Its capabilities are split into several subprojects. Each of them is accessible via https://psyplot.readthedocs.io/projects/<project-name>

- the psyplot_gui package: The GUI to psyplot
- the psy-simple package: A plugin for simple visualization
- the psy-maps package: A psyplot plugin for visualizing data on a map
- the psy-reg package: A psyplot plugin for visualizing and calculating regression fits

See Psyplot plugins for more informations on the plugins.

1.6 xarray Accessors

psyplot defines a DataArray and a Dataset accessor. You can use these accessors (see xarray Internals) to visualize your data and to update your plots. The following sections will show you how to make and update plots with these accessors. The plotmethods of the accessors are the same as for the psyplot.project.plot object.

1.6.1 The DatasetAccessor dataset accessor

Importing the psyplot package registers a new dataset accessor (see xarray.register_dataset_accessor()), the DatasetAccessor. You can access it via the psy attribute of the Dataset class, i.e.

```python
xarray.Dataset.psy alias of psyplot.data.DatasetAccessor
```

It can be used to visualize the variables in the dataset directly from the dataset itself, e.g.

```python
In [1]: import psyplot
In [2]: ds = psyplot.open_dataset('demo.nc')
In [3]: sp = ds.psy.plot.mapplot(name='t2m', cmap='Reds')
```
The variable `sp` is a psyplot subproject of the current main project.

```
In [4]: print(sp)
psycopg.project.Project({  arr0: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), ...
  →lev=100000.0, time=1979-01-31T18:00:00})
```

Hence, it would be completely equivalent if you type

```
In [5]: import psyplot.project as psyplot
In [6]: sp = psy.plot.mapplot(ds, name='t2m', cmap='Reds')
```

Note that the `DatasetAccessor.plot` attribute has the same plotmethods as the `psyplot.project.plot` instance.

### 1.6.2 The InteractiveArray dataarray accessor

More advanced then the `dataset accessor` is the registered DataArray accessor, the `InteractiveArray`. As well as the `DatasetAccessor`, it is registered as the `psy` attribute of any `DataArray`, i.e.

```
xarray.DataArray.psy alias of psyplot.data.InteractiveArray
```

You can use it for two things:

1. create plots of the array
2. update the plots and the array

#### Creating plots with the dataarray accessor

Just use the `plot` attribute the accessor.

```
In [7]: import psyplot
```
In [8]: ds = psyplot.open_dataset('demo.nc')

In [9]: da = ds.t2m[0, 0]

# this is a two dimensional array
In [10]: print(da)
<xarray.DataArray 't2m' (lat: 96, lon: 192)
array([[251.41689, 251.454 , 251.48915, ..., 251.29774, 251.33876, 251.37978],  
       [255.86024, 256.3114 , 256.72742, ..., 254.40712, 254.90517, 255.42665],  
...,
       [263.70984, 263.6454 , 263.58875, ..., 263.96375, 263.86804, 263.78406],  
       [262.4989 , 262.48718, 262.47742, ..., 262.5536 , 262.5321 , 262.51453],  
       dtype=float32)

Coordinates:
* lon (lon) float64 0.0 1.875 3.75 5.625 7.5 ... 352.5 354.4 356.2 358.1
* lat (lat) float64 88.57 86.72 84.86 83.0 ... -83.0 -84.86 -86.72 -88.57
lev float64 1e+05
time datetime64[ns] 1979-01-31T18:00:00

Attributes:
  long_name: Temperature
  units: K
  code: 130
  table: 128
  grid_type: gaussian

# and we can plot it using the mapplot plot method
In [11]: plotter = da.psy.plot.mapplot()

The resulting plotter, an instance of the psyplot.plotter.Plotter class, is the object that visualizes the data array. It can also be accessed via the da.psy.plotter attribute. Note that the creation of such a plotter overwrites any previous plotter in the da.psy.plotter attribute.

This methodology does not only work for DataArrays, but also for multiple DataArrays in a InteractiveList. This data structure is, for example, used by the psyplot.project.plot.lineplot plot method to visualize multiple lines. Consider the following example:
In [12]: ds0 = ds.isel(lev=0)  # select a subset of the dataset

# create a list of arrays at different longitudes
In [13]: l = psyplot.InteractiveList([
        ....: ds0.t2m.sel(lon=2.35, lat=48.86, method='nearest'),  # Paris
        ....: ds0.t2m.sel(lon=13.39, lat=52.52, method='nearest'),  # Berlin
        ....: ds0.t2m.sel(lon=-74.01, lat=40.71, method='nearest'),  # NYC
        ....: ])

In [14]: l.arr_names = ['Paris', 'Berlin', 'NYC']

# plot the list
In [15]: plotter = l.psy.plot.lineplot(xticks='data', xticklabels='%B')

Note that for the InteractiveList, the psy attribute is just the list itself. So it would have been equivalent to call

In [16]: l.plot.lineplot()

### Updating plots and arrays with the dataarray accessor

The InteractiveArray accessor is designed for interactive usage of, not only the matplotlib figures, but also of the data. If you selected a subset of a dataset, e.g. via

In [17]: da = ds.t2m[0, 0]
        ....: print(da.time)  # January 1979
        ....:
<xarray.DataArray 'time' ()>
array('1979-01-31T18:00:00.000000000', dtype='datetime64[ns]')
Coordinates:
  lev       float64 1e+05
  time      datetime64[ns] 1979-01-31T18:00:00
Attributes:
  standard_name:  time
You can change to a different slice using the `InteractiveArray.update()` method.

```
In [18]: da.psy.base = ds  # tell psyplot the source of the dataarray
In [19]: da.psy.update(time=2)
    ....: print(da.time)  # changed to March 1979
    ....:
<xarray.DataArray 'time' ()>
array('1979-03-31T18:00:00.000000000', dtype='datetime64[ns]')
Coordinates:
  lev   float64 1e+05
  time  datetime64[ns] 1979-03-31T18:00:00
Attributes:
  standard_name: time
```

The `da.psy.base = ds` command hereby tells the dataarray, where it is coming from, since this information is not known in the standard xarray framework.

**Hint:** You can avoid this, using the `DatasetAccessor.create_list()` method of the dataset accessor

```
In [20]: da = ds.psy.create_list(time=0, lev=0, name='t2m')[0]
    ....: print(da.psy.base is ds)
    ....:
True
```

If you plotted the data, you can also change the formatoptions using the `update()` method, e.g.

```
# create plot
In [21]: da.psy.plot.mapplot()
```

![Map plot example](image.png)
The same holds for the Interactive list

In [23]: l.update(time=slice(1, 4), # change the data by selecting a subset of the
    # timeslice
    ....:       title='Subset',   # change a formatoption, the title of the plot
    ....: )

1.7 Psyplot plugins

psyplot only provides the abstract framework on how to make the interactive visualization and data analysis. The real work is implemented in plugins to this framework. Each plugin is a separate package that has to be installed independent of psyplot and each plugin registers new plot methods for psyplot.project.plot.
1.7.1 Existing plugins

**psy_simple.plugin** A psyplot plugin for simple visualization tasks. This plugin provides a base for all the other plugins

- Examples Gallery
- plot methods

  `psyplot.project.plot.density` Make a density plot of point data
  `psyplot.project.plot.plot2d` Make a simple plot of a 2D scalar field
  `psyplot.project.plot.combined` Plot a 2D scalar field with an overlying vector field
  `psyplot.project.plot.violinplot` Make a violin plot of your data
  `psyplot.project.plot.lineplot` Make a line plot of one-dimensional data
  `psyplot.project.plot.vector` Make a simple plot of a 2D vector field
  `psyplot.project.plot.barplot` Make a bar plot of one-dimensional data

![Fig. 1: Bar plot demo](image)

**psy_maps.plugin** A psyplot plugin for visualizing data on a map

- Examples Gallery
- plot methods

  `psyplot.project.plot.mapplot` Plot a 2D scalar field on a map
  `psyplot.project.plot.mapvector` Plot a 2D vector field on a map
  `psyplot.project.plot.mapcombined` Plot a 2D scalar field with an overlying vector field on a map
Fig. 2: Line plot demo

Fig. 3: 2D plots
Fig. 4: Vector plot

Fig. 5: Violin plot demo
Fig. 6: Visualizing circumpolar data

Fig. 7: Basic data visualization on a map
psyreg.plugin  A psyplot plugin for visualizing and calculating regression fits

- Examples Gallery
- plot methods

  psyplot.project.plot.densityreg  Make a density plot and draw a fit from x to y of points

  psyplot.project.plot.linreg  Draw a fit from x to y

If you have new plugins that you think should be included in this list, please do not hesitate to open an issue on the github project page of psyplot or implement it by yourself in this file and make a pull request.

**Note:** Because psyplot plugins are imported right at the startup time of psyplot but nevertheless use the psyplot.config.rcsetup.RcParams class, you always have to import psyplot first if you want to load a psyplot plugin. In other words, if you want to import one of the above mentioned modules manually, you always have to type

```python
import psyplot
import PLUGIN_NAME.plugin
```

instead of

```python
import PLUGIN_NAME.plugin
import psyplot
```

where PLUGIN_NAME is any of psy_simple, psy_maps, etc.
Fig. 9: Creating and accessing a fit

Fig. 10: Plot a fit over a density plot
1.7.2 How to exclude plugins

The psyplot package loads all plugins right when the psyplot is imported. In other words, the statement

```
import psyplot
```

already includes that all the psyplot plugin packages are loaded.

You can however exclude plugins from the automatic loading via the PSYPLOT_PLUGINS environment variable and exclude specific plot methods of a plugin via the PSYPLOT_PLOTMETHODS variable.

The PSYPLOT_PLUGINS environment variable

This environment variable is a :: separated string with plugin names. If a plugin name is preceded by a no:, this plugin is excluded. Otherwise, only this plugin is included.

To show this behaviour, we can use psyplot --list-plugins which shows the plugins that are used. By default, all plugins are included

In [1]: !psyplot --list-plugins
- plugin = psy_simple.plugin
- plugin = psy_maps.plugin

Excluding psy-maps works via

In [2]: !PSYPLOT_PLUGINS=no:psy_maps.plugin psyplot --list-plugins
- plugin = psy_simple.plugin

Including only psy-maps works via

In [3]: !PSYPLOT_PLUGINS='yes:psy_maps.plugin' psyplot --list-plugins
- plugin = psy_maps.plugin

The PSYPLOT_PLOTMETHODS environment variable

The same principle is used when the plot methods are loaded from the plugins. If you want to manually exclude a plot method from loading, you include it via no:<plugin-module>::<plotmethod>. For example, to exclude the :attr:mapplot <psy_maps:psyplot.project.plot.mapplot> plot method from the psy-maps plugin, you can use

In [4]: !PSYPLOT_PLOTMETHODS=no:psy_maps.plugin:mapplot psyplot --list-plot-methods

barplot: Make a bar plot of one-dimensional data
density: Make a density plot of point data
fldmean: Calculate and plot the mean over x- and y-dimensions
lineplot: Make a line plot of one-dimensional data
mapcombined: Plot a 2D scalar field with an overlying vector field on a map
mapvector: Plot a 2D vector field on a map
plot2d: Make a simple plot of a 2D scalar field
violinplot: Make a violin plot of your data

and the same if you only want to include the :attr:mapplot <psy_maps:psyplot.project.plot.mapplot> and the :attr:lineplot <psy_simple:psyplot.project.plot.lineplot> methods
1.8 Command line usage

The `psyplot.__main__` module defines a simple parser to parse commands from the command line to make a plot of data in a netCDF file. Note that the arguments change slightly if you have the `psyplot-gui` module installed (see `psyplot-gui` documentation).

It can be run from the command line via:

```
python -m psyplot [options] [arguments]
```

or simply:

```
psyplot [options] [arguments]
```

Load a dataset, make the plot and save the result to a file

```
               [-n [variable_name [variable_name ...]]]
               [-d dim,val1,val2[,...]] [dim,val1,val2[,...]] ...
               [-pm {'mapcombined', 'vector', 'fldmean', 'plot2d', 'barplot',
               'combined', 'lineplot', 'violinplot', 'mapplot', 'mapvector', 'density'}]
               [-o str or list of str] [-p str] [-engine str] [-fmt FILENAME]
               [-op str] [-cd str]
               [-chname [project-variable,variable-to-use [project-variable,variable-to-use ...
               [str [str ...]]]
```

1.8.1 Positional Arguments

**str**

Either the filenames to show, or, if the `project` parameter is set, the a list of `-separated filenames to make a mapping from the original filename to a new one

Default: []

1.8.2 Named Arguments

- **-n, --name**
  
The variable names to plot if the `output` parameter is set

  Default: []

- **-d, --dims**
  
  A mapping from coordinate names to integers if the `project` is not given

- **-pm, --plot-method**
  
  Possible choices: mapcombined, vector, fldmean, plot2d, barplot, combined, lineplot, violinplot, mapplot, mapvector, density

  The name of the plot_method to use

- **-p, --project**
  
  If set, the project located at the given file name is loaded
-engine
The engine to use for opening the dataset (see `psyplot.data.open_dataset()`) 

-fmt, --formatoptions
The path to a yaml (`.yml` or `.yaml`) or pickle file defining a dictionary of formatoption that is applied to the data visualized by the chosen `plot_method`

-rc, --rc-file
The path to a yaml configuration file that can be used to update the `rcParams`

-e, --encoding
The encoding to use for loading the project. If None, it is automatically determined by pickle. Note: Set this to 'latin1' if using a project created with python2 on python3.

--enable-post
Enable the post processing formatoption. If True/set, post processing scripts are enabled in the given project. Only set this if you are sure that you can trust the given project file because it may be a security vulnerability.

Default: False

-sns, --seaborn-style
The name of the style of the seaborn package that can be used for the `seaborn.set_style()` function

-cd, --concat-dim
The concatenation dimension if multiple files in `fnames` are provided

Default: <inferred>

-chname
A mapping from variable names in the project to variable names in the datasets that should be used instead. Variable names should be separated by a comma.

Default: {}

### 1.8.3 Info options

Options that print informations and quit afterwards

-`-V, --version` show program's version number and exit

-`-aV, --all-versions` Print the versions of all plugins and requirements and exit

-`-lp, --list-plugins` Print the names of the plugins and exit

-`-lpm, --list-plot-methods` List the available plot methods and what they do

-`-lds, --list-datasets` List the used dataset names in the given project.

### 1.8.4 Output options

Options that only have an effect if the `-o` option is set.

-`-o, --output` If set, the data is loaded and the figures are saved to the specified filename and now graphical user interface is shown

-`-t, --tight` If True/set, it is tried to figure out the tight bbox of the figure and adjust the paper size of the `output` to it

Default: False

-`-op, --output-project` The name of a project file to save the project to
Examples

Here are some examples on how to use psyplot from the command line.

Plot the variable 't2m' in a netCDF file 'myfile.nc' and save the plot to 'plot.pdf':

```
$ psyplot myfile.nc -n t2m -pm mapplot -o test.pdf
```

Create two plots for 't2m' with the first and second timestep on the second vertical level:

```
$ psyplot myfile.nc -n t2m -pm mapplot -o test.pdf -d t,0,1 z,1
```

If you have saved a project using the `psyplot.project.Project.save_project()` method into a file named 'project.pkl', you can replay this via:

```
$ psyplot -p project.pkl -o test.pdf
```

If you use a different dataset than the one you used in the project (e.g. 'other_ds.nc'), you can replace it via:

```
$ psyplot other_dataset.nc -p project.pkl -o test.pdf
```

or explicitly via:

```
$ psyplot old_ds.nc,other_ds.nc -p project.pkl -o test.pdf
```

You can also load format options from a configuration file, e.g.:

```
$ echo 'title: my title' > fmt.yaml
$ psyplot myfile.nc -n t2m -pm mapplot -fmt fmt.yaml -o test.pdf
```

1.9 Example Gallery

The example gallery provides you with some examples on the general usage of the psyplot framework and shows you some applications of the different plotter classes in the psyplot package. You can either download the examples as a Jupyter Notebook or as a converted python script.

After downloading the jupyter-notebook, you can open it by typing:

```
$ jupyter notebook
```

into the terminal and navigate to the file you downloaded.

Note that the examples are python3 notebooks. If you are using python2, you might either open the notebook in an editor and rename 'python3' in each of the files to 'python3', or you create a new conda environment via:

```
conda create -n py35 python=3.5
source activate py35
conda install notebook ipykernel
ipython kernel install --user
```

and install the necessary modules into that environment.

There are lot’s of more examples out there for the

- psy-maps plugin
- psy-simple plugin
• psy-reg plugin

1.9.1 Usage of Climate Data Operators

This example shows you how CDOs are binded in the psyplot package.

It requires the 'demo.nc' netCDF file and the psy-maps plugin.

```python
import psyplot.project as psy

cdo = psy.Cdo()
lines = cdo.fldmean(input='-sellevidx,1 demo.nc', plot_method='lineplot', name='t2m',
fmt=dict(xticks='month', xticklabels='%b %Y'))

datetime_index = DatetimeIndex(['1979-01-31 18:00:00', '1979-02-28 18:00:00',
                               '1979-03-31 18:00:00', '1979-04-30 18:00:00',
                               '1979-05-31 18:00:00'],
dtype='datetime64[ns]', name='time', freq=\texttt{None})

maps = cdo.timmean(input='demo.nc', name='t2m', plot_method='mapplot', fmt=dict(cmap='RdBu_r'))
```

1.9. Example Gallery
1.9.2 Sharing formatoptions

This example shows you the capabilities of sharing formatoptions and what it is all about.

Within the psyplot framework you can easily manage multiple plots and even have interaction between them. This is especially useful if want to compare different variables.

This example requires the file 'sharing_demo.nc' which contains one variable for the temperature and the psy-maps plugin.

```python
import psyplot.project as psy
defname = 'sharing_demo.nc'

First we create 4 plots into one figure, one for each time step

maps = psy.plot.mapplot(fname, name='t2m', title='{dinfo}', ax=(2, 2), time=range(4))
```

---

```
RecursionError Traceback (most recent call last)
<ipython-input-3-f64617b189f6> in <module>
----> 1 maps = psy.plot.mapplot(fname, name='t2m', title='{dinfo}', ax=(2, 2),

~/.checkouts/readthedocs.org/user_builds/psyplot/conda/latest/lib/python3.7/site-packages/psyplot/project.py in __call__(self, *args, **kwargs)
    1747     return self._project_plotter._add_data(
    1748     ```self.plotter_cls, *args, **dict(chain(
--> 1749     ('prefer_list', self._prefer_list),
    1750     ```)}
```
As you see, they have slightly different boundaries which can be very annoying if we want to compare them. Therefore we can share the boundaries of the colorbar. The corresponding formatoption is the `bounds` formatoption.

```
maps.share(keys='bounds')
maps.show()
```
Now the very first array (January 31st) shares the boundaries with all the other. Furthermore it uses their data as well to calculate the range.

The sharing of formatoptions works for every formatoption key and formatoption groups.

maps[0].psy.plotter.groups

```
{'axes': 'Axes formatoptions',
 'colors': 'Color coding formatoptions',
 'labels': 'Label formatoptions',
 'masking': 'Masking formatoptions',
 'misc': 'Miscellaneous formatoptions',
 'plot': 'Plot formatoptions',
 'ticks': 'Axis tick formatoptions'}
```

Suppose for example, we want to work with only the last array but have the color settings kept equal throughout each plot. For this we can share the 'colors' group of the formatoption. To do this, we should first unshare the formatoptions currently the first one shares the boundaries with the others.

```
maps.unshare(keys='bounds')
# Now we share the color settings of the last one
arr = maps[-1]
maps[:-1].share(arr, keys='colors')
```

If we now update any of the color formatoptions of the last array, we update them for all the others. However, the other formatoptions (in this example the projection) keep untouched.
1.9.3 Applying your own post processing

This demo shows you how to use the `post` formatoption to apply your own post processing scripts. It requires the 'demo.nc' netCDF file, netCDF4 and the psy-maps plugin.

```
import psyplot.project as psy
```

**Usage**

The `post` formatoption let's you apply your own script to modify your data. Let's start with a simple plot of the wind speed:

```
sp = psy.plot.mapvector('demo.nc', name=[['u', 'v']], plot='stream',
                          color='red', title='%b %Y')
print(sp)
```
It is hard to see the continents below this amount of arrows. So we might want to enhance our plot with cartopy’s `stock_img`.

But since there is now formatoption for it, we can now either define a new plotter and add the formatoption, or we use the `post` formatoption.

```python
sp.update(enable_post=True,
          post='self.stock_img = self.ax.stock_img()')
sp.show()
```
The first parameter `enable_post=True` sets the `enable_post` attribute of the plotter to `True`. This attribute is by default set to `False` because it is always a security vulnerability to use the built-in `exec` function which is used by the `post` formatoption. We could, however, already have included this in our first definition of the project via

```python
sp = psy.plot.mapvector('demo.nc', name=[['u', 'v']], plot='stream', color='red', enable_post=True)
```

The second parameter `post='self.ax.stock_img()'` updates the `post` formatoption. It accepts an executable python script as a string. Note that we make use of the `self` variable, the only variable that is given to the script. It is the `Formatoption` instance that performs the update. Hence you can access all the necessary attributes and informations:

- the axes through `self.ax`
- the figure through `self.ax.figure`
- the data that is plotted through `self.data`
- the raw data from the dataset through `self.raw_data`
- the plotter through `self.plotter`
- any other formatoption in the plotter, e.g. the `title` through `self.plotter.title`

For example, let’s add another feature that adds the mean of the plotted variables to the plot. For this, let’s first have a look into the `plot_data` attribute of the plotter which includes the data that is visualized and that is accessible through the `data` attribute of the `post` formatoption:

```python
sp.plotTERS[0].plot_data
```

```python
<xarray.DataArray (variable: 2, lat: 96, lon: 192)>
array([[[-5.548854, -5.470729, ..., -5.680202, -5.618678],
    [-4.17483 , -4.246608, ..., -3.992702, -4.089869],
    ..., 
    [-2.664088, -2.547389, ..., -2.911159, -2.785182],
    [-3.48733 , -3.371119, ..., -3.708034, -3.599635]],
   [[ 2.322004, 2.529036, ..., 1.901594, 2.112532],
    [ 1.343489, 1.620344, ..., 0.820051, 1.075911],
    ...,
```

(continues on next page)
It is a 3-dimensional array, where the first dimension consists of the zonal wind speed 'u' and the meridional wind speed 'v'. So let's add their means as a text to the plot:

```python
sp.update(post=""
    self.stock_img = self.ax.stock_img()
    umean = self.data[0].mean().values
    vmean = self.data[1].mean().values
    abs_mean = (np.sqrt(self.data[0]**2 + self.data[1]**2)).mean().values
    self.text = self.ax.text(0., -0.15,
        'u: %1.4f m/s, v: %1.4f m/s, wind speed: %1.4f m/s' % (umean, vmean, abs_mean),
        transform=self.ax.transAxes)"")
sp.show()
```
Timing

psyplot is intended to work interactively. By default, the `post` format option is only updated when you personally update it. However, you can modify this timing using the `post_timing` format option. It can be either

- 'never': The default which requires a manual update
- 'replot': To update it when the data changes
- 'always': To always update it.

For example, in the current setting, when we change the data to the second time step via

```python
sp.update(time=1)
print(sp)
sp.show()
```

```python
copy
psyplot.project.Project({
    'arr0': 3-dim DataArray of u, v, with (variable, lat, lon)=(2, 96, 192), lev=100000.0, time=1979-02-28T18:00:00})
```

Our means are not updated, for this, we have to

1. set the `post_timing` to 'replot'
2. slightly modify our `post` script to not plot two texts above each other

```python
sp.update(post_timing='replot', post=""
    self.stock_img = self.ax.stock_img()
    umean = self.data[0].mean().values
    vmean = self.data[1].mean().values
    abs_mean = ((self.data[0]**2 + self.data[1]**2)**0.5).mean().values
    if hasattr(self, 'text'):
        text = self.text
    else:
        text = self.ax.text(0., -0.15, '',
            transform=self.ax.transAxes)
    text.set_text('u: %1.4f m/s, v: %1.4f m/s, wind speed: %1.4f m/s' % (
```

(continues on next page)
Now, if we update to the third timestep, our means are also calculated

```python
sp.update(time=2)
sp.show()
```

1.9.4 Saving and loading

Saving a project is straight forward via the `save_project` method

```python
d = sp.save_project()
```
However, when loading the project, the `enable_post` attribute is (for security reasons) again set to `False`. So if you are sure you can trust the post processing scripts in the `post` format option, load your project with `enable_post=True`

```python
psy.close('all')
sp = psy.Project.load_project(d, enable_post=True)
```

1.10 Developers guide

In this section we provide a deeper overview and introduction in the psyplot frameworks that is necessary for creating new plugins.
The main module we used so far, was the `psyplot.project` module. It is the end of a whole framework that is setup by the psyplot package.

This framework is designed in analogy to matplotlib's figure - axes - artist setup, where one figure controls multiple axes, an axes is the manager of multiple artists (e.g. a simple line) and each artist is responsible for visualizing one or more objects on the plot. The psyplot framework instead is defined through the `Project - (InteractiveBase - Plotter) - Formatoption` relationship.

The last to parts in this framework, the `Plotter` and `Formatoption`, are only defined through abstract base classes in this package. They are filled with contents in plugins such as the `psy-simple` or the `psy-maps` plugin (see `Psyplot plugins`).
The `project()` function

The `psyplot.project.Project` class (in analogy to matplotlib's `Figure` class) is basically a list that controls multiple plot objects. It comprises the full functionality of the package and packs it into one class, the `Project` class.

In analogy to `pyplot.figure()` function, a new project can simply be created via

```
In [1]: import psyplot.project as psy
In [2]: p = psy.project()
```

This automatically sets `p` to be the current project which can be accessed through the `gcp()` method. You can also set the current project by using the `scp()` function.

**Note:** We highly recommend to use the `project()` function to create new projects instead of creating projects from the `Project`. This ensures the right numbering of the projects of old projects.

The project uses the plotters from the `psyplot.plotter` module to visualize your data. Hence you can add new plots and new data to the project by using the `Project.plot` attribute or the `psyplot.project.plot` attribute which targets the current project. The return types of the plotting methods are again instances of the `Project` class, however we consider them as subprojects in contrast main projects that are created through the `project()` function. There is basically no difference but the result of the `Project.is_main` attribute which is `False` for subprojects. Hence, each new plot creates a subproject but also stores the data array in the corresponding main project of the `Project` instance from which the plot method has been called. The newly created subproject can be accessed via

```
In [3]: sp = psy.gcp()
```

whereas the current main project can be accessed via

```
In [4]: p = psy.gcp(main=True)
```

Plots created by a specific method of the `Project.plot` attribute may however be accessed via the corresponding attribute of the `Project` class. The following example creates three subprojects, two with the `mapplot` and `mapvector` methods from the `psy-maps` plugin and one with the simple `lineplot` method from the `psy-simple` plugin to visualize simple lines.

```
In [5]: import matplotlib.pyplot as plt
In [6]: import cartopy.crs as ccrs

# the subplots for the maps (need cartopy projections)
In [7]: ax = list(psy.multiple_subplots(2, 2, n=3, for_maps=True))

# the subplot for the line plot
In [8]: ax.append(plt.gcf().add_subplot(2, 2, 4))

# scalar field of the zonal wind velocity in the file demo.nc
In [9]: psy.plot.mapplot('demo.nc', name='u', ax=ax[0], clabel='{desc}')
Out[9]:

```

```

# a second scalar field of temperature
In [10]: psy.plot.mapplot('demo.nc', name='t2m', ax=ax[1], clabel='{desc}')

```

(continues on next page)
# a vector plot projected on the earth

In [11]: psy.plot.mapvector('demo.nc', name=['u', 'v'], ax=ax[2],
    .....:     attrs={'long_name': 'Wind speed'})
    .....:

Out[11]:

In [12]: psy.plot.lineplot('demo.nc', name='t2m', x=0, y=0, z=range(4),
    .....:     ax=ax[3], xticklabels='%b %d', ylabel='{desc}',
    .....:     legendlabels='%(zname)s = %(z)s %(zunits)s')
    .....:

The latter is now the current subproject we could access via `psy.gcp()`.
However we can access all of them through the main project

In [13]: mp = psy.gcp(True)

In [14]: mp  # all arrays
Out[14]:
2 Main psyplot.project.Project(
    arr0: 2-dim DataArray of u, with (lat, lon)=(96, 192),
    .....:     lev=100000.0, time=1979-01-31T18:00:00,
    arr1: 2-dim DataArray of t2m, with (lat, lon)=(96, 192),
    .....:     lev=100000.0, time=1979-01-31T18:00:00,
    arr2: 3-dim DataArray of u, v, with (variable, lat, lon)=(2, 96, 192),
    .....:     lev=100000.0, time=1979-01-31T18:00:00)

(continues on next page)
arr2: 3-dim DataArray of u, v, with (variable, lat, lon)=(2, 96, 192), lev=100000.0, time=1979-01-31T18:00:00,
arr3: psyplot.data.InteractiveList([arr0: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=100000.0,
arr1: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=85000.0,
arr2: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=50000.0,
arr3: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=20000.0])

In [15]: mp.mapplot # all scalar fields

psyplot.project.Project({
    arr0: 2-dim DataArray of u, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr1: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00})

In [16]: mp.mapvector # all vector plots

psyplot.project.Project({arr2: 3-dim DataArray of u, v, with (variable, lat, lon)=(2, 96, 192), lev=100000.0, time=1979-01-31T18:00:00})

In [17]: mp.maps # all data arrays that are plotted on a map

psyplot.project.Project({
    arr0: 2-dim DataArray of u, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr1: 2-dim DataArray of t2m, with (lat, lon)=(96, 192), lev=100000.0, time=1979-01-31T18:00:00,
    arr2: 3-dim DataArray of u, v, with (variable, lat, lon)=(2, 96, 192), lev=100000.0, time=1979-01-31T18:00:00})

In [18]: mp.lineplot # the simple plot we created

psyplot.project.Project({arr3: psyplot.data.InteractiveList([arr0: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=100000.0,
arr1: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=85000.0,
arr2: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=50000.0,
arr3: 1-dim DataArray of t2m, with (time)=(5,), lon=0.0, lat=88.57216851400727, lev=20000.0])})

The advantage is, since every plotter has different formatoptions, we can now update them very easily. For example lets update the arrowsize to 1 (which only works for the mapvector plots), the projection to an orthogonal (which only works for maps), the simple plots to use the 'viridis' colormap for color coding the lines and for all we choose their title corresponding to the variable names

In [19]: p.maps.update(projection='ortho')
The **InteractiveBase** and the **Plotter** classes

Interactive data objects

The next level are instances of the **InteractiveBase** class. This abstract base class provides an interface between the data and the visualization. Hence a plotter (that’s how we call instances of the **Plotter** class) will deal with the subclasses of the **InteractiveBase**:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InteractiveArray</strong>(xarray_obj, *args, **kwargs)</td>
<td>Interactive psyplot accessor for the data array</td>
</tr>
<tr>
<td><strong>InteractiveList</strong>(args, **kwargs)</td>
<td>List of <strong>InteractiveArray</strong> instances that can be plotted itself</td>
</tr>
</tbody>
</table>

Those classes (in particular the **InteractiveArray**) keep the reference to the base dataset to allow the update of the data slice you are plotting. The **InteractiveList** class can be used in a plotter for the visualization of multiple **InteractiveArray** instances (see for example the psyplot.plotter.simple.LinePlotter and psyplot.plotter.maps.CombinedPlotter classes). Furthermore those data instances have a **plotter** attribute that is usually occupied by an instance of a **Plotter** subclass.

**Note:** The **InteractiveArray** serves as a **DataArray** accessor. After you imported psyplot, you can access it via the **psy** attribute of a **DataArray**, i.e. via

```python
In [23]: import xarray as xr
In [24]: xr.DataArray([]).psy
Out[24]: <psypplot.data.InteractiveArray at 0x7f495b87e080>
```
Visualization objects

Each plotter class is the coordinator of several visualization options. Thereby the `Plotter` class itself contains only the structural functionality for managing the formatoptions that do the real work. The plotters for the real usage are defined in plugins like the `psy-simple` or the `psy-maps` package.

Hence each `InteractiveBase` instance is visualized by exactly one `Plotter` class. If you don’t want to use the `project framework`, the initialization of such an instance nevertheless straight forward. Just open a dataset, extract the right data array and plot it

```python
In [25]: from psyplot import open_dataset
In [26]: from psy_maps.plotters import FieldPlotter
In [27]: ds = open_dataset('demo.nc')
In [28]: arr = ds.t2m[0, 0]
In [29]: plotter = FieldPlotter(arr)
```

Now we created a plotter with all it’s formatoptions:

```python
In [30]: type(plotter), plotter
Out[30]:
(psy_maps.plotters.FieldPlotter,
 {'levels': None,
  'interp_bounds': None,
  'plot': 'mesh',
  'miss_color': None,
  'projection': 'cyl',
  'transform': 'cyl',
  'clon': None,
  'clat': None,
  'lonlatbox': None,
  ...
  }
)
```

(continues on next page)
You can use the show_keys(), show_summaries() and show_docs() methods to have a look into the documentation into the formatoptions or you simply use the builtin help() function for it:

```python
>>> help(plotter.clabel)
```

The update methods are the same as for the Project class. You can use the psyplot.data.InteractiveArray.update() via arr.psy.update() which updates the data and forwards the formatoptions to the Plotter.update() method.

**Note:** Plotters are subclasses of dictionaries where each item represents the key-value pair of one formatoption. Anyway, although you could now simply set a formatoption like you set an item for a dictionary via
In [31]: plotter['clabel'] = 'my label'

or equivalently

In [32]: plotter.clabel = 'my label'

this would not change the plot! Instead you have to use the `psyplot.plotter.Plotter.update()` method, i.e.

In [33]: plotter.update(clabel='my label')

**Formatoptions**

Formatoptions are the core of the visualization in the psyplot framework. They conceptually correspond to the basic `matplotlib.artist.Artist` and inherit from the abstract `Formatoption` class. Each plotter is set up through it’s formatoptions where each formatoption has a unique formatoption key inside the plotter. This formatoption key (e.g. ‘title’ or ‘clabel’) is what is used for updating the plot etc. You can find more information in *How to implement your own plotters and plugins*.

### 1.10.2 How to implement your own plotters and plugins

New plotters and plugins to the psyplot framework are highly welcomed. In this guide, we present *how to create new plotters* and explain to you how you can *include them as a plugin in psyplot*.

#### Creating plotters

Implementing new plotters can be very easy or quite an effort depending on how sophisticated you want to do it. In principle, you only have to implement the `Formatoption.update()` method and a default value. I.e., one simple formatoption would be

In [1]: from psyplot.plotter import Formatoption, Plotter

In [2]: class MyFormatoption(Formatoption):
   ...:     default = 'my text'
   ...:     def update(self, value):
   ...:         self.ax.text(0.5, 0.5, value, fontsize='xx-large')

...:

together with a plotter

In [3]: class MyPlotter(Plotter):
   ...:     my_fmt = MyFormatoption('my_fmt')

and your done. Now you can make a simple plot

In [4]: from psyplot import open_dataset

In [5]: ds = open_dataset('demo.nc')

In [6]: plotter = MyPlotter(ds.t2m)
However, if you’re using the psyplot framework, you probably will be a bit more advanced so let’s talk about attributes and methods of the Formatoption class.

If you look into the documentation of the Formatoption class, you find quite a lot of attributes and methods which probably is a bit depressing and confusing. But in principle, we can group them into 4 categories, the interface to the data, to the plotter and to other formatoptions. Plus an additional category for some Formatoption internals you definitely have to care about.

**Interface for the plotter**

The first interface is the one, that interfaces to the plotter. The most important attributes in this group are the `key`, `priority`, `plot_fmt`, `initialize_plot()` and most important the `update()` method.

The `key` is the unique key for the formatoption inside the plotter. In our example above, we assign the 'my_fmt' key to the MyFormatoption class in MyPlotter. Hence, this key is defined when the plotter class is defined and will be automatically assigned to the formatoption.

The next important attribute is the `priority` attribute. There are three stages in the update of a plotter:

1. The stage with data manipulation. If formatoptions manipulate the data that shall be visualized (the `data` attribute), those formatoptions are updated first. They have the `psyplot.plotter.START` priority.
2. The stage of the plot. Formatoptions that influence how the data is visualized are updated here (e.g. the colormap or formatoptions that do the plotting). They have the `psyplot.plotter.BEFOREPLOTTING` priority.
3. The stage of the plot where additional informations are inserted. Here all the labels are updated, e.g. the title, xlabel, etc. This is the default priority of the `Formatoption.priority` attribute, the `psyplot.plotter.END` priority.

If there is any formatoption updated within the first two groups, the plot of the plotter is updated. This brings us to the third important attribute, the `plot_fmt`. This boolean tells the plotter, whether the corresponding formatoption is assumed to make a plot at the end of the second stage (the BEFOREPLOTTING stage). If this attribute is True, then the plotter will call the `Formatoption.make_plot()` method of the formatoption instance.

Finally, the `initialize_plot()` and `update()` methods, this is were your contribution really is required. The `initialize_plot()` method is called when the plot is created for the first time, the `update()` method when it is updated (the default implementation of the `initialize_plot()` simply calls the `update()` method). Implement these methods in your formatoption and thereby make use of the interface to the `data` and other formatoptions.
Interface to the data

The next set of attributes help you to interface to the data. There are two important parts in this section the interface to the data and the interpretation of the data.

The first part is mainly represented to the Formatoption.data and Formatoption.raw_data attributes. The first part is mainly represented to the Formatoption.data and Formatoption.raw_data attributes. The plotter that contains the formatoption often creates a copy of the data because the data for the visualization might be modified (see for example the psy_reg.plotter.LinRegPlotter). This modified data can be accessed through the Formatoption.data and should be the standard approach to access the data within a formatoption. Nevertheless, the original data can be accessed through the Formatoption.raw_data attribute. However, it only makes sense to access this data for formatoption with START priority.

The result of these two attributes depend on the Formatoption.index_in_list attribute. The data objects in the psyplot framework are either a xarray.DataArray or a list of those in a psyplot.data.InteractiveList. If the index_in_list attribute is not None, and the data object is an InteractiveList, then only the array at the specified position is returned. To completely avoid this issue, you might also use the iter_data or iter_raw_data attributes.

The second part in this section is the interpretation of the data and here, the formatoption can use the Formatoption.decoder attribute. This subclass of the psyplot.data.CFDecoder helps you to identify the x- and y-variables in the data.

Interfacing to other formatoptions

A formatoption is the lowest level in the psyplot framework. It is represented at multiple levels:

1. at the lowest level through the subclass of the Formatoption class
2. at the Plotter class level which includes the formatoption class as a descriptor (in our example above it's MyPlotter.my_fmt)
3. at the Plotter instance level through
   i. a personalized instance of the corresponding Formatoption class (i.e. plotter = MyPlotter(); plotter.my_fmt is not MyPlotter.my_fmt)
   ii. an item in the plotter (i.e. plotter = MyPlotter(); plotter['my_fmt'])
4. In the update methods of the Plotter, psyplot.data.InteractiveBase and psyplot.data.ArrayList as a keyword (i.e. plotter = MyPlotter(); plotter.update(my_fmt='new value'))

Hence, there is one big to the entire framework, that is: the functionality of a new formatoption has to be completely defined through exactly one argument, i.e. it must be possible to assign a value to the formatoption in the plotter.

For complex formatoption, this might indeed be quite a challenge for the developer and there are two solutions to it:

1. The simple solution for the developer: Allow a dictionary as a formatoption, here we also have the psyplot.plotter.DictFormatoption to help you.
2. Interface to other formatoptions

First solution: Use a dict

That said, to implement a formatoption that inserts a custom text and let the user define the size of the text, you either create a formatoption that accepts a text via
class CustomText(DictFormatoption):
    default = {'text': ''}

    text = None

    def validate(self, value):
        if not isinstance(value, dict):
            return {'text': value}
        return value

    def initialize_plot(self, value):
        self.text = self.ax.text(0.2, 0.2, value['text'],
                                  fontsize=value.get('size', 'large'))

    def update(self, value):
        self.text.set_text(value['text'])
        self.text.set_fontsize(value.get('size', 'large'))

class MyPlotter(Plotter):
    my_fmt = CustomText('my_fmt')

and then you could create and update a plotter via

```python
p = MyPlotter(xarray.DataArray([]))
p.update(my_fmt='my text')  # updates the text
p.update(my_fmt={'size': 14})  # updates the size
p.update(my_fmt={'size': 14, 'text': 'Something'})  # updates text and size
```

This solution has the several advantages:

- The user does not get confused through too many formatoptions
- It is easy to allow more keywords for this formatoption

Indeed, the psy_simple.plotter.Legend formatoption uses this framework since the matplotlib.pyplot.legend() function accepts that many keywords that it would be not informative to create a formatoption for each of them.

Otherwise you could of course avoid the DictFormatoption and just force the user to always provide a new dictionary.

Second solution: Interact with other formatoptions

Another possibility is to implement a second formatoption for the size of the text. And here, the psyplot framework helps you with several attributes of the Formatoption class:

- **the children attribute** Forces the listed formatoptions in this list to be updated before the current formatoption is updated
- **the dependencies attributes** Same as children but also forces an update if one of the named formatoptions are updated
- **the parents attribute** Skip the update if one of the parents is updated
- **the connections attribute** just provides connections to the listed formatoptions
Each of those attributes accept a list of strings that represent the formatoption keys of other formatoptions. Those formatoptions are then accessible within the formatoption via the usual `getattr()`. I.e. if you list a formatoption in the `children` attribute, you can access it inside the formatoption (`self`) via `self.other_formatoption`.

In our example of the `CustomText`, this could be implemented via

```python
class CustomTextSize(Formatoption):
    """
    Set the fontsize of the custom text
    Possible types
    --------------
    int
        The fontsize of the text
    """
    default = int
    def validate(self, value):
        return int(value)
    # this text has not to be updated if the custom text is updated
    children = ['text']
    def update(self, value):
        self.text.text.set_fontsize(value)

class CustomText(Formatoption):
    """
    Place a text
    Possible types
    -----------
    str
        The text to display"
    ""
    def initialize_plot(self, value):
        self.text = self.ax.text(0.2, 0.2, value['text'])
    def update(self, value):
        self.text.set_text(value)

class MyPlotter(Plotter):
    my_fmt = CustomText('my_fmt')
    my_fmtsize = CustomTextSize('my_fmtsize', text='my_fmt')
```

the update in that sense would be like

and then you could create and update a plotter via

```python
p = MyPlotter(xarray.DataArray([]))
p.update(my_fmt='my text')  # updates the text
p.update(my_fmtsize=14)    # updates the size
p.update(my_fmt='Something', my_fmtsize=14)  # updates text and size
```

The advantages of this methodology are basically:
• The user straight away sees two formatoptions that can be interpreted easily
• The formatoption that controls the font size could easily be subclassed and replaced in a subclass of MyPlotter. In the first framework using the DictFormatoption, this would mean that the entire process has to be rewritten.

As you see in the above definition `my_fmtsize = CustomTextSize('my_fmtsize', text='my_fmt')`, we provide an additional text keyword. That is because we explicitly named the text key in the children attribute of the CustomTextSize formatoption. In that way we can tell the my_fmtsize formatoption how to find the necessary formatoption. That works for all keys listed in the children, dependencies, parents and connections attributes.

Creating new plugins

Now that you have created your plotter, you may want to include it in the plot methods of the Project class such that you can do something like

```python
import psyplot.project as psy
psy.plot.my_plotter('netcdf-file.nc', name='varname')
```

There are three possibilities how you can do this:

1. The easy and fast solution for one session: register the plotter using the psyplot.project.register_plotter() function
2. The easy and steady solution: Save the calls you used in step 1 in the 'project.plotter.user' key of the rcParams
3. The steady and shareable solution: Create a new plugin

The third solution has been used for the psy-maps and psy-simple plugins. To create a skeleton for your plugin, you can use the psyplot-plugin command that is installed when you install psyplot.

For our demonstration, let’s create a plugin named my-plugin. This is simply done via

```bash
In [7]: !psyplot-plugin my-plugin
In [8]: import glob
In [9]: glob.glob('my-plugin/**', recursive=True)
```

The following files are created in a directory named 'my-plugin':

'setup.py' The installation script
'my_plugin/plugin.py' The file that sets up the configuration of our plugin. This file should define the rcParams for the plugin (see also rcParams handling in plugins)

'my_plugin/plotters.py' The file in which we define the plotters. This file should define the plotters and formatoptions.

If you want to see more, look into the comments in the created files.

rcParams handling in plugins

Every formatoption does have default values. In our example above, we simply set it via the default attribute. This is a hard-coded, but easy, stable and quick solution.

However, your formatoption could also be used in different plotters, each requiring a different default value. Or you want to give the user the possibility to set his own default value. For this, we implemented the

```
psycopg.plotter.Plotter. _rcparams_string List of base strings in the psycopg.rcParams
dictionary
```

attribute. Here you can specify a string for this plotter which is used to get the default value of the formatoptions in this plotter from the rcParams. The expected default_key for one formatoption would then be the_chosen_string + fmt_key.

The following example illustrates this:

```
In [10]: from psycopg.config.rcsetup import rcParams
       ....: from psycopg.plotter import Plotter, Formatoption
       ....:
First we define our defaultParams, a mapping from default key to the default value, a validation function, and a description (see the psycopg.config.rcsetup.defaultParams dictionary).

In [11]: defaultParams = {
       ....:     'plotter.example_plotter.fmt1': [
       ....:         1, lambda val: int(val), 'Example formatoption']
       ....: }
       ....:
Then we update the defaultParams of the psycopg.rcParams and set the value

In [12]: rcParams.defaultParams.update(defaultParams)
In [13]: rcParams.update_from_defaultParams(defaultParams)
       ....: print(rcParams['plotter.example_plotter.fmt1'])
       ....: 1
```

Now we define a formatoption for our new plotter class and implement it in a new plotter object.

```
In [14]: class ExampleFmt(Formatoption):
       ....:     def update(self, value):
       ....:         pass
       ....:
In [15]: class ExamplePlotter(Plotter):
       ....:     # we use our base string, 'plotter.example_plotter.'
       ....:     _rcparams_string = ['plotter.example_plotter. ']
```

(continues on next page)
If we now create a new instance of this `ExamplePlotter`, the `fmt1` formatoption will have a value of 1, as we defined it in the above `defaultParams`:

```
In [16]: plotter = ExamplePlotter()
In [17]: print(plotter['fmt1'])
1
```

Changing the value in the `rcParams`, also changes the default value for the plotter

```
In [20]: rcParams['plotter.example_plotter.fmt1'] = 2
In [21]: print(plotter.fmt1.default)
2
```

Also, if we subclass this plotter, the `default_key` will not change

```
In [22]: class SecondPlotter(ExamplePlotter):
    ...:    # we set a new `_rcparams_string`
    ...:    _rcparams_string = ['plotter.another_plotter.']
    ...:
In [23]: plotter = SecondPlotter()
```

If you’re developing a new plugin you would then have to define the `rcParams` and `defaultParams` in the plugin.py script (see *Creating new plugins*) and they will then be automatically implemented in `psyplot.rcParams`.

## 1.11 Contributing to psyplot

First off, thanks for taking the time to contribute!

The following is a set of guidelines for contributing to psyplot and its packages, which are hosted on GitHub. These are mostly guidelines, not rules. Use your best judgment, and feel free to propose changes to this document in a pull request.
1.11.1 Code of Conduct

This project and everyone participating in it is governed by the psyplot Code of Conduct. By participating, you are expected to uphold this code.

1.11.2 What should I know before I get started?

The psyplot framework

psyplot is just the framework that allows interactive data analysis and visualization. Much of the functionality however is implemented by other packages. What package is the correct one for your bug report/feature request, can be determined by the following list

- psyplot-gui: Everything specific to the graphical user interface
- psy-simple: Everything concerning, e.g. the lineplot, plot2d, density or vector plot methods
- psy-maps: Everything concerning, e.g. the mapplot, mapvector mapcombined plot methods
- psy-reg: Everything concerning, e.g. the linreg or densityreg plot methods
- psyplot: Everything concerning the general framework, e.g. data handling, parallel update, etc.

Concerning plot methods, you can simply find out which module implemented it via

```python
import psyplot.project as psy
print(psy.plot.name_of_your_plot_method._plugin)
```

If you still don’t know, where to open the issue, just go for psyplot.
1.11.3 How Can I Contribute?

Reporting Bugs

This section guides you through submitting a bug report for psyplot. Following these guidelines helps maintainers and the community understand your report, reproduce the behavior, and find related reports.

Before creating bug reports, please check existing issues and pull requests as you might find out that you don’t need to create one. When you are creating a bug report, please include as many details as possible. Fill out the required template, the information it asks for helps us resolve issues faster.

Note: If you find a Closed issue that seems like it is the same thing that you’re experiencing, open a new issue and include a link to the original issue in the body of your new one.

How Do I Submit A (Good) Bug Report?

Bugs are tracked as GitHub issues. After you’ve determined which repository your bug is related to, create an issue on that repository and provide the following information by filling in the template.

Explain the problem and include additional details to help maintainers reproduce the problem:

- Use a clear and descriptive title for the issue to identify the problem.
- Describe the exact steps which reproduce the problem in as many details as possible. For example, start by explaining how you started psyplot, e.g. which command exactly you used in the terminal, or how you started psyplot otherwise. When listing steps, don’t just say what you did, but explain how you did it. For example, did you update via GUI or console and what?
- Provide specific examples to demonstrate the steps. Include links to files or GitHub projects, or copy/pasteable snippets, which you use in those examples. If you’re providing snippets in the issue, use Mark-down code blocks.
- Describe the behavior you observed after following the steps and point out what exactly is the problem with that behavior.
- Explain which behavior you expected to see instead and why.
- Include screenshots and animated GIFs which show you following the described steps and clearly demonstrate the problem. You can use this tool to record GIFs on macOS and Windows, and this tool or this tool on Linux.
- If the problem is related to your data structure, include a small example how a similar data structure can be generated.

Include details about your configuration and environment:

- Which version of psyplot are you using? You can get the exact version by running psyplot -aV in your terminal, or by starting the psyplot-gui and open Help->Dependencies.
- What’s the name and version of the OS you’re using?

Suggesting Enhancements

This section guides you through submitting an enhancement suggestion for psyplot, including completely new features and minor improvements to existing functionality.

If you want to change an existing feature, use the change feature template, otherwise fill in the new feature template.
How Do I Submit A (Good) Enhancement Suggestion?

Enhancement suggestions are tracked as GitHub issues. After you’ve determined which repository your enhancement suggestion is related to, create an issue on that repository and provide the following information:

- **Use a clear and descriptive title** for the issue to identify the suggestion.
- **Provide a step-by-step description of the suggested enhancement** in as many details as possible.
- **Provide specific examples to demonstrate the steps.** Include copy/pasteable snippets which you use in those examples, as Markdown code blocks.
- **Describe the current behavior** and explain which behavior you expected to see instead and why.
- **Include screenshots and animated GIFs** which help you demonstrate the steps or point out the part of psyplot which the suggestion is related to. You can use this tool to record GIFs on macOS and Windows, and this tool or this tool on Linux.
- **Explain why this enhancement would be useful** to most psyplot users.
- **List some other analysis software or applications where this enhancement exists.**
- **Specify which version of psyplot you’re using.** You can get the exact version by running `psyplot -aV` in your terminal, or by starting the psyplot-gui and open Help->Dependencies.
- **Specify the name and version of the OS you’re using.**

### Pull Requests

- Fill in the required template
- Do not include issue numbers in the PR title
- Include screenshots and animated GIFs in your pull request whenever possible.
- Document new code based on the *Documentation Styleguide*
- End all files with a newline and follow the PEP8, e.g. by using `flake8`

### Adding new examples

You have new examples? Great! If you want to add them to the documentation, please just fork the correct github repository and add a jupyter notebook in the `examples` directory, together with all the necessary data files.

To build our docs, we use the `sphinx-nbexamples` package to convert the examples to reStructuredText. Therefore please make sure that your supplementary data files are correctly implemented in the meta data of the notebook.

We are, however, also willing to help you finalizing incomplete pull requests.

### 1.11.4 Styleguides

**Git Commit Messages**

- Use the present tense ("Add feature" not "Added feature")
- Use the imperative mood ("Move cursor to..." not "Moves cursor to...")
- Limit the first line (summary) to 72 characters or less
- Reference issues and pull requests liberally after the first line
• When only changing documentation, include \[\text{ci skip}\] in the commit title

Documentation Styleguide

• Follow the numpy documentation guidelines.
• Use reStructuredText.
• Try to not repeat yourself and make use of the psyplot.docstring.docstrings

Example

```python
@docstrings.get_sectionsf('new_function')
def new_function(a=1):
    """Make some cool new feature

    This function implements a cool new feature

    Parameters
    ----------
    a: int
        First parameter

    Returns
    -------
    something awesome
        The result"
...

@docstrings.dedent
def another_new_function(a=1, b=2):
    """Make another cool new feature

    Parameters
    ----------
    %(new_function.parameters)s
    b: int
        Another parameter

    Returns
    -------
    Something even more awesome"
...
```

Note: This document has been inspired by the contribution guidelines of Atom

1.12 API Reference

psyplot visualization framework

Functions
**get_versions**((requirements, key))

Get the version information for psyplot, the plugins and its requirements

**Data**

**with_gui**

Boolean that is True, if psyplot runs inside the graphical user interface

**psyplot.get_versions**(requirements=True, key=None)

Get the version information for psyplot, the plugins and its requirements

**Parameters**

- **requirements** (*bool*) – If True, the requirements of the plugins and psyplot are investigated
- **key** (*func*) – A function that determines whether a plugin shall be considered or not. The function must take a single argument, that is the name of the plugin as string, and must return True (import the plugin) or False (skip the plugin). If None, all plugins are imported

**Returns**

A mapping from 'psyplot'/'the plugin names to a dictionary with the 'version' key and the corresponding version is returned. If requirements is True, it also contains a mapping from 'requirements' a dictionary with the versions

**Return type**

dict

**Examples**

Using the built-in JSON module, we get something like

```python
import json
print(json.dumps(psyplot.get_versions(), indent=4))
{
    "psy_simple.plugin": {
        "version": "1.0.0.dev0"
    },
    "psyplot": {
        "version": "1.0.0.dev0",
        "requirements": {
            "matplotlib": "1.5.3",
            "numpy": "1.11.3",
            "pandas": "0.19.2",
            "xarray": "0.9.2"
        }
    },
    "psy_maps.plugin": {
        "version": "1.0.0.dev0",
        "requirements": {
            "cartopy": "0.15.0"
        }
    }
}
```

**psyplot.with_gui = False**

Boolean that is True, if psyplot runs inside the graphical user interface by the psyplot_gui module
1.12.1 Subpackages

psyplot.compat package

Submodules

psyplot.compat.pycompat module

Compatibility module for different python versions

That's a test

Classes

DictMethods

Functions

get_default_value(func, arg)
getcwd(*args, **kwargs)
isstring(s)

class psyplot.compat.pycompat.DictMethods
    Bases: object

    static iteritems(d)
    static iterkeys(d)
    static itervalues(d)

psyplot.compat.pycompat.get_default_value(func, arg)
psyplot.compat.pycompat.getcwd(*args, **kwargs)
psyplot.compat.pycompat.isstring(s)

psyplot.config package

Configuration module of the psyplot package

This module contains the module for managing rc parameters and the logging. Default parameters are defined in
the `rcsetup.defaultParams` dictionary, however you can set up your own configuration in a yaml file (see `psyplot.load_rc_from_file()`)

Data

<table>
<thead>
<tr>
<th>config_path</th>
<th>str or None. Path to the yaml configuration file (if found).</th>
</tr>
</thead>
<tbody>
<tr>
<td>logcfg_path</td>
<td>str. Path to the yaml logging configuration file</td>
</tr>
</tbody>
</table>

```
psyplot.config.config_path = '/home/docs/.config/psyplot/psyplotrc.yml'
```

str or None. Path to the yaml configuration file (if found). See `psyplot_fname()` for further information

Type: class

```
psyplot.config.logcfg_path = '/home/docs/checkouts/readthedocs.org/user_builds/psyplot/checkouts/latest/psyplot/config/logging.yml'
```

str. Path to the yaml logging configuration file

Submodules

**psyplot.config.logsetup module**

Logging configuration module of the psyplot package

This module defines the essential functions for setting up the `logging.Logger` instances that are used by the psyplot package.

Functions

```
setup_logging([default_path, default_level, ...])  # Setup logging configuration
```

```
psyplot.config.logsetup.setup_logging(default_path=None, default_level=20, env_key='LOG_PSYPLOT')
```

Setup logging configuration

Parameters

- **default_path** (str) – Default path of the yaml logging configuration file. If None, it defaults to the ‘logging.yaml’ file in the config directory
- **default_level** (int) – Default: `logging.INFO`. Default level if default_path does not exist
- **env_key** (str) – environment variable specifying a different logging file than `default_path` (Default: ‘LOG_CFG’)

Returns path – Path to the logging configuration file

Return type str

Notes

Function taken from http://victorlin.me/posts/2012/08/26/good/logging-practice-in-python
psyplot Documentation, Release 1.2.1

psyplot.config.rcsetup module

Default management of the psyplot package

This module defines the necessary classes, data and functions for the default configuration of the module. The structure is motivated and to larger parts taken from the matplotlib package.

Classes

- RcParams(*args, **kwargs)
  A dictionary object including validation
- SubDict(base, base_str[, pattern, ...])
  Class that keeps weak reference to the base dictionary

Functions

- get_configdir([name, env_key])
  Return the string representing the configuration directory.
- psyplot_fname([env_key, fname, if_exists])
  Get the location of the config file.
- safe_list(l)
  Function to create a list
- validate_bool(b)
  Convert b to a boolean or raise
- validate_bool_maybe_none(b)
  Convert b to a boolean or raise
- validate_dict(d)
  Validate a dictionary
- validate_path_exists(s)
  If s is a path, return s, else False
- validate_str(s)
  Validate a string
- validate_stringlist(s)
  Validate a list of strings
- validate_stringset(*args, **kwargs)
  Validate a set of strings

Data

- defaultParams
dict with default values and validation functions
- rcParams
  RcParams instance that stores default

```python
class psyplot.config.rcsetup.RcParams(*args, **kwargs)
Bases: dict

A dictionary object including validation

validating functions are defined and associated with rc parameters in defaultParams

This class is essentially the same as in maplotlib RcParams but has the additional find_and_replace() method.

Parameters
defaultParams (dict) – The defaultParams to use (see the defaultParams attribute). By default, the psyplot.config.rcsetup.defaultParams dictionary is used

Other Parameters

**args, **kwargs – Any key-value pair for the initialization of the dictionary
```

Attributes

- HEADER
  str(object='') -> str
- defaultParams
  The description of each keyword in the rcParams dictionary
- msg_depr
  str(object='') -> str

Continued on next page
Table 17 – continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg_depr_ignore</td>
<td>str(object='') -&gt; str</td>
</tr>
<tr>
<td>validate</td>
<td>Dictionary with validation methods as values</td>
</tr>
</tbody>
</table>

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect(key, func)</td>
<td>Connect a function to the given formatoption</td>
</tr>
<tr>
<td>copy()</td>
<td>Make sure, the right class is retained</td>
</tr>
<tr>
<td>disconnect([key, func])</td>
<td>Disconnect the connections to the an rcParams key</td>
</tr>
<tr>
<td>dump([fname, overwrite, include_keys, ...])</td>
<td>Dump this instance to a yaml file</td>
</tr>
<tr>
<td>find_all(pattern)</td>
<td>Return the subset of this RcParams dictionary whose keys match, using <code>re.search()</code>, the given pattern.</td>
</tr>
<tr>
<td>find_and_replace(*args, **kwargs)</td>
<td>Like <code>find_all()</code> but the given strings are replaced</td>
</tr>
<tr>
<td>keys()</td>
<td>Return sorted list of keys.</td>
</tr>
<tr>
<td>load_from_file([fname])</td>
<td>Update rcParams from user-defined settings</td>
</tr>
<tr>
<td>load_plugins([raise_error])</td>
<td>Load the plotters and defaultParams from the plugins</td>
</tr>
<tr>
<td>remove(key, func)</td>
<td>Disconnect the connections to the an rcParams key</td>
</tr>
<tr>
<td>update([E, ]**F)</td>
<td>If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]</td>
</tr>
<tr>
<td>update_from_defaultParams([defaultParams, ...])</td>
<td>Update from the a dictionary like the defaultParams</td>
</tr>
<tr>
<td>values()</td>
<td>Return values in order of sorted keys.</td>
</tr>
</tbody>
</table>

HEdGER = 'Configuration parameters of the psyplot module
n
You can copy this file (or parts of it) to another path and save it as psyplotrc.yml. The directory should then be stored in the PSYPLOTCONFIGDIR environment variable.'

**connect (key, func)**

Connect a function to the given formatoption

**Parameters**

- **key (str)** – The rcParams key
- **func (function)** – The function that shall be called when the rcParams key changes. It must accept a single value that is the new value of the key.

**copy ()**

Make sure, the right class is retained

**defaultParams**

**descriptions**

The description of each keyword in the rcParams dictionary

**disconnect (key=None, func=None)**

Disconnect the connections to the an rcParams key

**Parameters**

- **key (str)** – The rcParams key. If None, all keys are used
- **func (function)** – The function that is connected. If None, all functions are connected

**dump (fname=None, overwrite=True, include_keys=None, exclude_keys=['project.plotters'], include_descriptions=True, **kwargs)**

Dump this instance to a yaml file

**Parameters**
• **fname** (str or None) – file name to write to. If None, the string that would be written to a file is returned

• **overwrite** (bool) – If True and **fname** already exists, it will be overwritten

• **include_keys** (None or list of str) – Keys in the dictionary to be included. If None, all keys are included

• **exclude_keys** (list of str) – Keys from the RcParams instance to be excluded

Other Parameters ‘**kwargs’ – Any other parameter for the yaml.dump() function

Returns if **fname** is None, the string is returned. Otherwise, None is returned

Return type str or None

Raises IOError – If **fname** already exists and **overwrite** is False

See also:

load_from_file()

**find_all** (pattern)

Return the subset of this RcParams dictionary whose keys match, using re.search(), the given pattern.

Parameters **pattern** (str) – pattern as suitable for re.compile

Returns RcParams instance with entries that match the given **pattern**

Return type RcParams

Notes

Changes to the returned dictionary are (different from **find_and_replace()** are not propagated to the parent RcParams dictionary.

See also:

**find_and_replace()**

**find_and_replace**(*args, **kwargs)

Like **find_all()** but the given strings are replaced

This method returns a dictionary-like object that keeps weak reference to this rcParams instance. The resulting SubDict instance takes the keys from this rcParams instance but leaves away what is found in **base_str**.

**args** and **kwargs** are determined by the SubDict class, where the **base** dictionary is this one.

Parameters

• **base_str** (str or list of str) – Strings that are used as to look for keys to get and set keys in the base dictionary. If a string does not contain '%(key)s', it will be appended at the end. '%(key)s' will be replaced by the specific key for getting and setting an item.

• **pattern** (str) – Default: '.+'. This is the pattern that is inserted for %(key)s in a base string to look for matches (using the re module) in the base dictionary. The default pattern matches everything without white spaces.

• **pattern_base** (str or list or str) – If None, the whatever is given in the **base_str** is used. Those strings will be used for generating the final search patterns.
can specify this parameter by yourself to avoid the misinterpretation of patterns. For example for a `base_str` like 'my.str' it is recommended to additionally provide the `pattern_base` keyword with 'my\.str'. Like for `base_str`, the `%{key}s` is appended if not already in the string.

- **trace (bool)** – Default: False. If True, changes in the SubDict are traced back to the `base` dictionary. You can change this behaviour also afterwards by changing the `trace` attribute

- **replace (bool)** – Default: True. If True, everything but the ‘%(key)s’ part in a base string is replaced (see examples below)

**Returns** SubDict with this rcParams instance as reference.

**Return type** `SubDict`

### Examples

The syntax is the same as for the initialization of the `SubDict` class:

```python
>>> from psyplot import rcParams
>>> d = rcParams.find_and_replace(["plotter.baseplotter.", ...
                                         "plotter.vector."])
>>> print(d['title'])
None
>>> print(d['arrowsize'])
1.0
```

**See also:**

`find_all()`, `SubDict()`

**keys()**

Return sorted list of keys.

**load_from_file (fname=None)**

Update rcParams from user-defined settings

This function updates the instance with what is found in `fname`

**Parameters**

- **fname (str)** – Path to the yaml configuration file. Possible keys of the dictionary are defined by `config.rcsetup.defaultParams`. If None, the `config.rcsetup.psyplot_fname()` function is used.

**See also:**

`dump_to_file()`, `psyplot_fname()`

**load_plugins (raise_error=False)**

Load the plotters and defaultParams from the plugins

This method loads the `plotters` attribute and `defaultParams` attribute from the plugins that use the entry point specified by `group`. Entry points must be objects (or modules) that have a `defaultParams` and a `plotters` attribute.

**Parameters**

- **raise_error (bool)** – If True, an error is raised when multiple plugins define the same plotter or rcParams key. Otherwise only a warning is raised

  ```python
  msg_depr = '%s is deprecated and replaced with %s; please use the latter.'
  msg_depr_ignore = '%s is deprecated and ignored. Use %s'
  ```
remove \( (key, \text{func}) \) → None. Update \( D \) from dict/iterable \( E \) and \( F \).

If \( E \) is present and has a .keys() method, then does: for \( k \in E: D[k] = E[k] \) If \( E \) is present and lacks a .keys() method, then does: for \( k, v \in E: D[k] = v \) In either case, this is followed by: for \( k \in F: D[k] = F[k] \)

update \( \{E, **F\} \) → None. Update \( D \) from dict/iterable \( E \) and \( F \).

\( \text{update} \) from \( \text{defaultParams} \) (defaultParams=None, plotters=True)

Update from the a dictionary like the \text{defaultParams}

Parameters

- \text{defaultParams} (dict) – The \text{defaultParams} like dictionary. If None, the \text{defaultParams} attribute will be updated
- \text{plotters} (bool) – If True, 'project.plotters' will be updated too

validate

Dictionary with validation methods as values

values()

Return values in order of sorted keys.

class psyplot.config.rcsetup.SubDict (base, base_str='.+', pattern_base=None, trace=False, replace=True)

Bases: collections.UserDict, dict

Class that keeps week reference to the base dictionary

This class is used by the \text{RcParams.find_and_replace()} method to provide an easy handable instance that keeps reference to the base rcParams dictionary.

Parameters

- \text{base} (dict) – base dictionary
- \text{base_str} (str or list of str) – Strings that are used as to look for keys to get and set keys in the \text{base} dictionary. If a string does not contain '%(key)s', it will be appended at the end. '%(key)s' will be replaced by the specific key for getting and setting an item.
- \text{pattern} (str) – Default: '.+'. This is the pattern that is inserted for '%(key)s' in a base string to look for matches (using the re module) in the \text{base} dictionary. The default \text{pattern} matches everything without white spaces.
- \text{pattern_base} (str or list or str) – If None, the whatever is given in the \text{base_str} is used. Those strings will be used for generating the final search patterns. You can specify this parameter by yourself to avoid the misinterpretation of patterns. For example for a \text{base_str} like 'my.str' it is recommended to additionally provide the \text{pattern_base} keyword with 'my\.str'. Like for \text{base_str}, the '%(key)s' is appended if not already in the string.
- \text{trace} (bool) – Default: False. If True, changes in the SubDict are traced back to the \text{base} dictionary. You can change this behaviour also afterwards by changing the \text{trace} attribute
- \text{replace} (bool) – Default: True. If True, everything but the '%(key)s' part in a base string is replaced (see examples below)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_base_str(base_str[, pattern, ...])</td>
<td>Add further base string to this instance</td>
</tr>
<tr>
<td>iteritems()</td>
<td>Unsorted iterator over items</td>
</tr>
<tr>
<td>iterkeys()</td>
<td>Unsorted iterator over keys</td>
</tr>
</tbody>
</table>
### Table 19 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>itervalues()</code></td>
<td>Unsorted iterator over values</td>
</tr>
<tr>
<td><code>update(*args, **kwargs)</code></td>
<td>Update the dictionary</td>
</tr>
</tbody>
</table>

### Attributes

- **base**: `dict`. Reference dictionary
- **base_str**: list of strings. The strings that are used to set and get a specific key
- **data**: Dictionary representing this `SubDict` instance
- **patterns**: list of compiled patterns from the `base_str` attribute, that
- **replace**: `bool`. If True, matching strings in the `base_str` replace
- **trace**: `bool`. If True, changes are traced back to the `base` dict

### Notes

- If a key of matches multiple strings in `base_str`, the first matching one is used.
- the `SubDict` class is (of course) not that efficient as the `base` dictionary, since we loop multiple times through it’s keys

### Examples

Initialization example:

```python
>>> from psyplot import rcParams
>>> d = rcParams.find_and_replace(['plotter.baseplotter.', ...
  'plotter.vector.'])
>>> print d['title']
my title
>>> print d['arrowsize']
1.0
```

To convert it to a usual dictionary, simply use the `data` attribute:

```python
>>> d.data
{'title': None, 'arrowsize': 1.0, ...}
```

Note that changing one keyword of your `SubDict` will not change the `base` dictionary, unless you set the `trace` attribute to True:

```python
>>> d['title'] = 'my title'
>>> print(d['title'])
my title
>>> print(rcParams['plotter.baseplotter.title'])
>>> d.trace = True
>>> d['title'] = 'my second title'
>>> print(d['title'])
my second title
```

(continues on next page)
Furthermore, changing the `replace` attribute will change how you can access the keys:

```python
>>> d.replace = False
# now setting d['title'] = 'anything' would raise an error (since
# d.trace is set to True and 'title' is not a key in the rcParams
data dictionary. Instead we need
>>> d['plotter.baseplotter.title'] = 'anything'
```

See also:

`RcParams.find_and_replace`

`add_base_str` *(base_str, pattern='.+', pattern_base=None, append=True)*

Add further base string to this instance

**Parameters**

- `base_str (str or list of str)` – Strings that are used as to look for keys to get and set keys in the `base` dictionary. If a string does not contain '%(key)s', it will be appended at the end. '%(key)s' will be replaced by the specific key for getting and setting an item.

- `pattern (str)` – Default: '.+'. This is the pattern that is inserted for '%(key)s' in a base string to look for matches (using the `re` module) in the `base` dictionary. The default pattern matches everything without white spaces.

- `pattern_base (str or list or str)` – If None, the whatever is given in the `base_str` is used. Those strings will be used for generating the final search patterns. You can specify this parameter by yourself to avoid the misinterpretation of patterns. For example for a `base_str` like 'my.str' it is recommended to additionally provide the `pattern_base` keyword with 'my\.str'. Like for `base_str`, the '%(key)s' is appended if not already in the string.

- `append (bool)` – If True, the given `base_str` are appended (i.e. it is first looked for them in the `base` dictionary), otherwise they are put at the beginning

```python
dict = {}

base = {}

base_str = []

data = {}

See also:

`iteritems`

`iteritems` *

Unsorted iterator over items

`iterkeys` *

Unsorted iterator over keys
**itervalues()**
Unsorted iterator over values

**patterns = []**
list of compiled patterns from the `base_str` attribute, that are used to look for the matching keys in `base`

**replace**
`bool`. If True, matching strings in the `base_str` attribute are replaced with an empty string.

**trace = False**
`bool`. If True, changes are traced back to the `base` dict

**update(** `*args, **kwargs`)**
Update the dictionary

`psyplot.config.rcsetup.defaultParams`
dict with default values and validation functions

`psyplot.config.rcsetup.get_configdir**(name='psyplot', env_key='PSYPLOTCONFDIR')**
Return the string representing the configuration directory.

The directory is chosen as follows:
1. If the `env_key` environment variable is supplied, choose that.
2a. On Linux and osx, choose `$HOME/.config/` + name.
2b. On other platforms, choose `$HOME/.` + name.
3. If the chosen directory exists, use that as the configuration directory.
4. A directory: return None.

**Parameters**
- **name (str)** – The name of the program
- **env_key (str)** – The environment variable that can be used for the configuration directory

**Notes**
This function is motivated by the `matplotlib.matplotlib_fname()` function

`psyplot.config.rcsetup.psyplot_fname**(env_key='PSYPLOTRC', fname='psyplotrc.yml', if_exists=True)**`
Get the location of the config file.

The file location is determined in the following order
- `$PWD/psyplotrc.yml`
- environment variable `PSYPLOTRC` (pointing to the file location or a directory containing the file `psyplotrc.yml`)
- `SPSYPLOTCONFDIR/psyplot`
- On Linux and osx,
  - `$HOME/.config/psyplot/psyplotrc.yml`
- On other platforms,
  - `$HOME/psyplot/psyplotrc.yml` if `$HOME` is defined.
• Lastly, it looks in $PSYPLOTTDATA/psyplotrc.yml for a system-defined copy.

Parameters

- **env_key** *(str)* – The environment variable that can be used for the configuration directory
- **fname** *(str)* – The name of the configuration file
- **if_exists** *(bool)* – If True, the path is only returned if the file exists

Returns None, if no file could be found and if_exists is True, else the path to the psyplot configuration file

Return type None or str

Notes

This function is motivated by the `matplotlib.matplotlib_fname()` function

**psyplot.config.rcsetup.rcParams**

RcParams instance that stores default format options and configuration settings.

**psyplot.config.rcsetup.safe_list(l)**

Function to create a list

Parameters `l` *(iterable or anything else)* – Parameter that shall be converted to a list.

- If string or any non-iterable, it will be put into a list
- If iterable, it will be converted to a list

Returns `l` put (or converted) into a list

Return type list

**psyplot.config.rcsetup.validate_bool(b)**

Convert `b` to a boolean or raise

**psyplot.config.rcsetup.validate_bool_maybe_none(b)**

Convert `b` to a boolean or raise

**psyplot.config.rcsetup.validate_dict(d)**

Validate a dictionary

Parameters `d` *(dict or str)* – If str, it must be a path to a yaml file

Returns

Return type dict

Raises **ValueError** –

**psyplot.config.rcsetup.validate_path_exists(s)**

If `s` is a path, return `s`, else False

**psyplot.config.rcsetup.validate_str(s)**

Validate a string

Parameters `s` *(str)* –

Returns

Return type str

 Raises **ValueError** –
psyplot.config.rcsetup.validate_stringlist(s)
Validate a list of strings

Parameters val (iterable of strings)
Returns list of str
Return type list
Raises ValueError

psyplot.config.rcsetup.validate_stringset(*args, **kwargs)
Validate a set of strings

Parameters val (iterable of strings)
Returns set of str
Return type set
Raises ValueError

psyplot.sphinxext package
Sphinx extension package of the psyplot module

Submodules

psyplot.sphinxext.extended_napoleon module
Sphinx extension module to provide additional sections for numpy docstrings
This extension extends the sphinx.ext.napoleon package with an additional Possible types section in order to document possible types for descriptors.

Notes
If you use this module as a sphinx extension, you should not list the sphinx.ext.napoleon module in the extensions variable of your conf.py. This module has been tested for sphinx 1.3.1.

Classes

<table>
<thead>
<tr>
<th>DocstringExtension</th>
<th>Class that introduces a “Possible Types” section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExtendedGoogleDocstring(docstring[, config, ...])</td>
<td>sphinx.ext.napoleon.GoogleDocstring with more sections</td>
</tr>
<tr>
<td>ExtendedNumpyDocstring(docstring[, config, ...])</td>
<td>sphinx.ext.napoleon.NumpyDocstring with more sections</td>
</tr>
</tbody>
</table>

Functions

| process_docstring(app, what, name, obj, ...) | Process the docstring for a given python object. |
| setup(app) | Sphinx extension setup function |

class psyplot.sphinxext.extended_napoleon.DocstringExtension
Bases: object
Class that introduces a “Possible Types” section

This class serves as a base class for sphinx.ext.napoleon.NumpyDocstring and sphinx.ext.napoleon.GoogleDocstring to introduce another section names Possible types

Examples

The usage is the same as for the NumpyDocstring class, but it supports the Possible types section:

```python
>>> from sphinx.ext.napoleon import Config
>>> from psyplot.sphinxext.extended_napoleon import (...
... ExtendedNumpyDocstring)
>>> config = Config(napoleon_use_param=True,
... napoleon_use_rtype=True)
>>> docstring = '''...
... Possible types
... --------------
... type1
... Description of `type1`
... type2
... Description of `type2`'''
>>> print(ExtendedNumpyDocstring(docstring, config))
.. rubric:: Possible types
   *
   type1  --
   Description of `type1`
   type2  --
   Description of `type2`
```

```python
class psyplot.sphinxext.extended_napoleon.ExtendedGoogleDocstring
document (docstring, config= None, app=None, what= None, name=
  None, obj=None, options=None)

  extended_napoleon.DocstringExtension
  sphinx.ext.napoleon.GoogleDocstring with more sections

class psyplot.sphinxext.extended_napoleon.ExtendedNumpyDocstring
document (docstring, config= None, app=None, what= None, name=
  None, obj=None, options=None)

  extended_napoleon.DocstringExtension
  sphinx.ext.napoleon.NumpyDocstring with more sections
```
psyplot.sphinxext.extended_napoleon.process_docstring(app, what, name, obj, options, lines)

Process the docstring for a given python object.

Called when autodoc has read and processed a docstring. lines is a list of docstring lines that process_docstring modifies in place to change what Sphinx outputs.

The following settings in conf.py control what styles of docstrings will be parsed:

- napoleon_google_docstring – parse Google style docstrings
- napoleon_numpy_docstring – parse NumPy style docstrings

Parameters

- app (sphinx.application.Sphinx) – Application object representing the Sphinx process.
- what (str) – A string specifying the type of the object to which the docstring belongs. Valid values: “module”, “class”, “exception”, “function”, “method”, “attribute”.
- name (str) – The fully qualified name of the object.
- obj (module, class, exception, function, method, or attribute) – The object to which the docstring belongs.
- options (sphinx.ext.autodoc.Options) – The options given to the directive: an object with attributes inherited_members, undoc_members, show_inheritance and noindex that are True if the flag option of same name was given to the auto directive.
- lines (list of str) – The lines of the docstring, see above.

Note: lines is modified in place

Notes

This function is (to most parts) taken from the sphinx.ext.napoleon module, sphinx version 1.3.1, and adapted to the classes defined here

psyplot.sphinxext.extended_napoleon.setup(app)
Sphinx extension setup function

When the extension is loaded, Sphinx imports this module and executes the setup() function, which in turn notifies Sphinx of everything the extension offers.

Parameters app (sphinx.application.Sphinx) – Application object representing the Sphinx process

Notes

This function uses the setup function of the sphinx.ext.napoleon module

1.12.2 Submodules

psyplot.data module

Classes
AbsoluteTimeDecoder(array)

AbsoluteTimeEncoder(array)

ArrayList(iterable=[], attrs={}, auto_update=None, new_name=True) Base class for creating a list of interactive arrays from a dataset

CFDecoder(ds, x, y, z, t) Class that interpretes the coordinates and attributes according to

DatasetAccessor(ds) A dataset accessor to interface with the psyplot package

InteractiveArray(xarray_obj, *args, **kwargs) Interactive psyplot accessor for the data array

InteractiveBase(plotter, arr_name, auto_update) Class for the communication of a data object with a suitable plotter

InteractiveList(*args, **kwargs) List of InteractiveArray instances that can be plotted itself

Signal(name, cls_signal) Signal to connect functions to a specific event

UGridDecoder(ds, x, y, z, t) Decoder for UGrid data sets

Functions

decode_absolute_time(times)
encode_absolute_time(times)

get_filename_ds(ds[, dump, paths]) Return the filename of the corresponding to a dataset

get_index_from_coord(coord, base_index) Function to return the coordinate as integer, integer array or slice

get_tdata(t_format, files) Get the time information from file names

open_dataset(filename_or_obj[, decode_cf, ...]) Open an instance of xarray.Dataset.

open_mfdataset(paths[, decode_cf, ...]) Open multiple files as a single dataset.

setup_coords(arr_names, sort, dims) Sets up the arr_names dictionary for the plot

to_netcdf(ds, *args, **kwargs) Store the given dataset as a netCDF file

to_slice(arr) Test whether arr is an integer array that can be replaced by a slice

Data

get_fname_funcs functions to use to extract the file name from a data store

t_patterns mapping that translates datetime format strings to regex patterns

class psyplot.data.AbsoluteTimeDecoder(array) Bases: xarray.core.utils.NDArrayMixin Attributes
dtype

dtype
class psyplot.data.AbsoluteTimeEncoder(array) Bases: xarray.core.utils.NDArrayMixin Attributes
dtype

dtype
class psyplot.data.ArrayList(iterable=[], attrs={}, auto_update=None, new_name=True)
Bases: `list`

Base class for creating a list of interactive arrays from a dataset

This list contains and manages `InteractiveArray` instances

**Parameters**

- **iterable** (`iterable`) – The iterable (e.g. another list) defining this list
- **attrs** (`dict-like or iterable, optional`) – Global attributes of this list
- **auto_update** (`bool`) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not. See also the `no_auto_update` attribute. If None, the value from the 'lists.auto_update' key in the `psyplot.rcParams` dictionary is used.
- **new_name** (`bool or str`) – If False, and the `arr_name` attribute of the new array is already in the list, a `ValueError` is raised. If True and the `arr_name` attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, `new_name` is set to ‘arr{0}’'. If not True, this will be used for renaming (if the array name of `arr` is in use or not). '{0}' is replaced by a counter

**Attributes**

- **all_dims**
  - The dimensions for each of the arrays in this list
- **all_names**
  - The variable names for each of the arrays in this list
- **arr_names**
  - Names of the arrays (!not of the variables!) in this list
- **arrays**
  - A list of all the `xarray.DataArray` instances in this list
- **coords**
  - Names of the coordinates of the arrays in this list
- **coords_intersect**
  - Coordinates of the arrays in this list that are used in all arrays
- **dims**
  - Dimensions of the arrays in this list
- **dims_intersect**
  - Dimensions of the arrays in this list that are used in all arrays
- **is_unstructured**
  - A boolean for each array whether it is unstructured or not
- **logger**
  - `logging.Logger` of this instance
- **names**
  - Set of the variable in this list
- **no_auto_update**
  - `bool`.
  - Boolean controlling whether the `start_update()` method.
- **with_plotter**
  - The arrays in this instance that are visualized with a plotter

**Methods**

- **append**, `append(value[, new_name])`
  - Append a new array to the list
- **array_info**, `array_info([dump, paths, attrs, ...])`
  - Get dimension informations on you arrays
- **copy**, `copy([deep])`
  - Returns a copy of the list
- **draw**, `draw()`
  - Draws all the figures in this instance
- **extend**, `extend(iterable[, new_name])`
  - Add further arrays from an iterable to this list
- **from_dataset**, `from_dataset(base[, method, default_slice, ...])`
  - Construct an `ArrayList` instance from an existing base dataset

Continued on next page
Table 29 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>from_dict(d[, alternative_paths, datasets, ...])</code></td>
<td>Create a list from the dictionary returned by <code>array_info()</code></td>
</tr>
<tr>
<td><code>next_available_name([fmt_str, counter])</code></td>
<td>Create a new array out of the given format string</td>
</tr>
<tr>
<td><code>remove(arr)</code></td>
<td>Removes an array from the list</td>
</tr>
<tr>
<td><code>rename(arr[, new_name])</code></td>
<td>Rename an array to find a name that isn’t already in the list</td>
</tr>
<tr>
<td><code>start_update([draw])</code></td>
<td>Conduct the registered plot updates</td>
</tr>
<tr>
<td><code>update([method, dims, fmt, replot, ...])</code></td>
<td>Update the coordinates and the plot</td>
</tr>
</tbody>
</table>

**all_dims**

The dimensions for each of the arrays in this list

**all_names**

The variable names for each of the arrays in this list

**append(value, new_name=False)**

Append a new array to the list

**Parameters**

- `value (InteractiveBase)` – The data object to append to this list
- `new_name (bool or str)` – If False, and the `arr_name` attribute of the new array is already in the list, a `ValueError` is raised. If True and the `arr_name` attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, `new_name` is set to ‘arr[0]’. If not True, this will be used for renaming (if the array name of `arr` is in use or not). ‘{0}’ is replaced by a counter

**Raises**

- `ValueError` – If it was impossible to find a name that isn’t already in the list
- `ValueError` – If `new_name` is False and the array is already in the list

**See also:**

`list.append()`, `extend()`, `rename()`

**arr_names**

Names of the arrays (not of the variables!) in this list

This attribute can be set with an iterable of unique names to change the array names of the data objects in this list.

**array_info**

Get dimension informations on your arrays

This method returns a dictionary containing informations on the array in this instance

**Parameters**

- `dump (bool)` – If True and the dataset has not been dumped so far, it is dumped to a temporary file or the one generated by `paths` is used. If it is False or both, `dump` and `paths` are None, no data will be stored. If it is None and `paths` is not None, `dump` is set to True.
- `paths (iterable or True)` – An iterator over filenames to use if a dataset has no filename. If `paths` is True, an iterator over temporary files will be created without raising a warning
- **attrs** *(bool, optional)* – If True (default), the `ArrayList.attrs` and `xarray.DataArray.attrs` attributes are included in the returning dictionary

- **standardize_dims** *(bool, optional)* – If True (default), the real dimension names in the dataset are replaced by x, y, z and t to be more general.

- **pwd**(str) – Path to the working directory from where the data can be imported. If None, use the current working directory.

- **use_rel_paths**(bool, optional) – If True (default), paths relative to the current working directory are used. Otherwise absolute paths to `pwd` are used

- **ds_description** *(all’ or set of {'fname', 'ds', 'num', 'arr', 'store'})* – Keys to describe the datasets of the arrays. If all, all keys are used. The key descriptions are
  
  **fname** the file name is inserted in the 'fname' key

  **store** the data store class and module is inserted in the 'store' key

  **ds** the dataset is inserted in the 'ds' key

  **num** The unique number assigned to the dataset is inserted in the 'num' key

  **arr** The array itself is inserted in the 'arr' key

- **full_ds**(bool) – If True and 'ds' is in `ds_description`, the entire dataset is included. Otherwise, only the DataArray converted to a dataset is included

- **copy**(bool) – If True, the arrays and datasets are deep copied

Other Parameters

- **‘**kwargs‘** – Any other keyword for the `to_netcdf()` function

- **path**(str, Path or file-like object, optional) – Path to which to save this dataset. File-like objects are only supported by the scipy engine. If no path is provided, this function returns the resulting netCDF file as bytes; in this case, we need to use scipy, which does not support netCDF version 4 (the default format becomes NETCDF3_64BIT).

- **mode** *(‘w’, ‘a’, optional)* – Write (‘w’) or append (‘a’) mode. If mode=’w’, any existing file at this location will be overwritten. If mode=’a’, existing variables will be overwritten.

- **format** *(‘NETCDF4’, ‘NETCDF4_CLASSIC’, ‘NETCDF3_64BIT’, ‘NETCDF3_CLASSIC’)*, optional File format for the resulting netCDF file:
  
  - **NETCDF4**: Data is stored in an HDF5 file, using netCDF4 API features.
  - **NETCDF4_CLASSIC**: Data is stored in an HDF5 file, using only netCDF 3 compatible API features.
  - **NETCDF3_64BIT**: 64-bit offset version of the netCDF 3 file format, which fully supports 2+ GB files, but is only compatible with clients linked against netCDF version 3.6.0 or later.
  - **NETCDF3_CLASSIC**: The classic netCDF 3 file format. It does not handle 2+ GB files very well.

All formats are supported by the netCDF4-python library. scipy.io.netcdf only supports the last two formats.

The default format is NETCDF4 if you are saving a file to disk and have the netCDF4-python library available. Otherwise, xarray falls back to using scipy to write netCDF files and defaults to the NETCDF3_64BIT format (scipy does not support netCDF4).
• **group** (*str, optional*) – Path to the netCDF4 group in the given file to open (only works for format='NETCDF4'). The group(s) will be created if necessary.

• **engine** (*{'netcdf4', 'scipy', 'h5netcdf'}, optional*) – Engine to use when writing netCDF files. If not provided, the default engine is chosen based on available dependencies, with a preference for ‘netcdf4’ if writing to a file on disk.

• **encoding** (*dict, optional*) – Nested dictionary with variable names as keys and dictionaries of variable specific encodings as values, e.g., `{'my_variable': {'dtype': ‘int16’, ‘scale_factor’: 0.1, ‘zlib’: True}, ...}`

The *h5netcdf* engine supports both the NetCDF4-style compression encoding parameters (‘zlib’: True, ‘complevel’: 9) and the h5py ones (‘compression’: 'gzip', 'compression_opts': 9). This allows using any compression plugin installed in the HDF5 library, e.g. LZF.

**Returns** An ordered mapping from array names to dimensions and filename corresponding to the array

**Return type** OrderedDict

**See also:**

`from_dict()`

**arrays**

A list of all the *xarray.DataArray* instances in this list

**coords**

Names of the coordinates of the arrays in this list

**coords_intersect**

Coordinates of the arrays in this list that are used in all arrays

**copy** (*deep=False*)

Returns a copy of the list

**Parameters**

- **deep** (*bool*) – If False (default), only the list is copied and not the contained arrays, otherwise the contained arrays are deep copied

**dims**

Dimensions of the arrays in this list

**dims_intersect**

Dimensions of the arrays in this list that are used in all arrays

**draw()**

Draws all the figures in this instance

**extend**(iterable, **new_name=False**) Add further arrays from an iterable to this list

**Parameters**

- **iterable** – Any iterable that contains *InteractiveBase* instances

- **new_name** (*bool or str*) – If False, and the *arr_name* attribute of the new array is already in the list, a ValueError is raised. If True and the *arr_name* attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, *new_name* is set to ‘arr{0}’. If not True, this will be used for renaming (if the array name of *arr* is in use or not). ' {0} ' is replaced by a counter
Raises

- **ValueError** – If it was impossible to find a name that isn’t already in the list
- **ValueError** – If `new_name` is False and the array is already in the list

See also:

list.extend(), append(), rename()

classmethod from_dataset(base, method='isel', default_slice=None, decoder=None, auto_update=None, prefer_list=False, squeeze=True, attrs=None, load=False, **kwargs)

Construct an ArrayList instance from an existing base dataset

Parameters

- **base (xarray.Dataset)** – Dataset instance that is used as reference
- **method ({'isel', None, 'nearest', ..})** – Selection method of the xarray.Dataset to be used for setting the variables from the informations in `dims`. If `method` is ‘isel’, the xarray.Dataset.isel() method is used. Otherwise it sets the `method` parameter for the xarray.Dataset.sel() method.
- **auto_update (bool)** – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not. See also the `no_auto_update` attribute. If None, the value from the 'lists.auto_update' key in the psyplot.rcParams dictionary is used.
- **prefer_list (bool)** – If True and multiple variable names pher array are found, the InteractiveList class is used. Otherwise the arrays are put together into one InteractiveArray.
- **default_slice (indexer)** – Index (e.g. 0 if `method` is ‘isel’) that shall be used for dimensions not covered by `dims` and `**furtherdims`. If None, the whole slice will be used.
- **decoder (CFDecoder)** – The decoder that shall be used to decoder the `base` dataset
- **squeeze (bool, optional)** – Default True. If True, and the created arrays have an axes with length 1, it is removed from the dimension list (e.g. an array with shape (3, 4, 1, 5) will be squeezed to shape (3, 4, 5))
- **attrs (dict, optional)** – Meta attributes that shall be assigned to the selected data arrays (additional to those stored in the `base` dataset)
- **load (bool or dict)** – If True, load the data from the dataset using the xarray.DataArray.load() method. If `dict`, those will be given to the above mentioned load method

Other Parameters

- **arr_names (string, list of strings or dictionary)** – Set the unique array names of the resulting arrays and (optionally) dimensions.
  
  – if string: same as list of strings (see below). Strings may include {0} which will be replaced by a counter.
  
  – list of strings: those will be used for the array names. The final number of dictionaries in the return depend in this case on the `dims` and `**furtherdims`
  
  – dictionary: Then nothing happens and an OrderedDict version of `arr_names` is returned.
• **sort (list of strings)** – This parameter defines how the dictionaries are ordered. It has no effect if `arr_names` is a dictionary (use a `OrderedDict` for that). It can be a list of dimension strings matching to the dimensions in `dims` for the variable.

• **dims (dict)** – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example `dims = {'name': 't2m', 'time': 0}` will result in one plot for the first time step, whereas `dims = {'name': 't2m', 'time': [0, 1]}` will result in two plots, one for the first (`time == 0`) and one for the second (`time == 1`) time step.

• **"**kwargs"** – The same as `dims` (those will update what is specified in `dims`)

**Returns** The list with the specified `InteractiveArray` instances that hold a reference to the given `base`

**Return type** `ArrayList`

```python
classmethod from_dict(d, alternative_paths={}, datasets=None, pwd=None, ignore_keys=['attrs', 'plotter', 'ds'], only=None, chname={}, **kwargs)```

Create a list from the dictionary returned by `array_info()`

This classmethod creates an `ArrayList` instance from a dictionary containing filename, dimension infos and array names

**Parameters**

• **d (dict)** – The dictionary holding the data

• **alternative_paths (dict or list or str)** – A mapping from original filenames as used in `d` to filenames that shall be used instead. If `alternative_paths` is not None, datasets must be None. Paths must be accessible from the current working directory. If `alternative_paths` is a list (or any other iterable) is provided, the file names will be replaced as they appear in `d` (note that this is very unsafe if `d` is not and OrderedDict)

• **datasets (dict or list or None)** – A mapping from original filenames in `d` to the instances of `xarray.Dataset` to use. If it is an iterable, the same holds as for the `alternative_paths` parameter

• **pwd (str)** – Path to the working directory from where the data can be imported. If None, use the current working directory.

• **ignore_keys (list of str)** – Keys specified in this list are ignored and not seen as array information (note that `attrs` are used anyway)

• **only (string, list or callable)** – Can be one of the following three things:
  – a string that represents a pattern to match the array names that shall be included
  – a list of array names to include
  – a callable with two arguments, a string and a dict such as

```python
def filter_func(arr_name: str, info: dict) -> bool

    'Filter the array names'
    'This function should return True if the array shall be included, else False'

Parameters
----------
```
The function should return True if the array shall be included, else False. This function will also be given to subsequents instances of InteractiveList objects that are contained in the returned value

- **chname** (dict) – A mapping from variable names in the project to variable names that should be used instead

Other Parameters

- **`**kwargs**` – Any other parameter from the psyplot.data.open_dataset function
- **filename_or_obj** (str, Path, file or xarray.backends.*DataStore) – Strings and Path objects are interpreted as a path to a netCDF file or an OpenDAP URL and opened with python-netCDF4, unless the filename ends with .gz, in which case the file is gunzipped and opened with scipy.io.netcdf (only netCDF3 supported). Byte-strings or file-like objects are opened by scipy.io.netcdf (netCDF3) or h5py (netCDF4/HDF).
- **group** (str, optional) – Path to the netCDF4 group in the given file to open (only works for netCDF4 files).
- **decode_cf** (bool, optional) – Whether to decode these variables, assuming they were saved according to CF conventions.
- **mask_and_scale** (bool, optional) – If True, replace array values equal to _FillValue with NA and scale values according to the formula original_values * scale_factor + add_offset, where _FillValue, scale_factor and add_offset are taken from variable attributes (if they exist). If the _FillValue or missing_value attribute contains multiple values a warning will be issued and all array values matching one of the multiple values will be replaced by NA. mask_and_scale defaults to True except for the pseudonetcdf backend.
- **decode_times** (bool, optional) – If True, decode times encoded in the standard NetCDF datetime format into datetime objects. Otherwise, leave them encoded as numbers.
- **autoclose** (bool, optional) – If True, automatically close files to avoid OS Error of too many files being open. However, this option doesn’t work with streams, e.g., BytesIO.
- **concat_characters** (bool, optional) – If True, concatenate along the last dimension of character arrays to form string arrays. Dimensions will only be concatenated over (and removed) if they have no corresponding variable and if they are only used as the last dimension of character arrays.
- **decode_coords** (bool, optional) – If True, decode the ‘coordinates’ attribute to identify coordinates in the resulting dataset.
- **chunks** (int or dict, optional) – If chunks is provided, it used to load the new dataset into dask arrays. chunks={} loads the dataset with dask using a single chunk for all arrays.
- **lock** (False or duck threading.Lock, optional) – Resource lock to use when reading data from disk. Only relevant when using dask or another form of parallelism. By default,
appropriate locks are chosen to safely read and write files with the currently active dask scheduler.

- **cache** *(bool, optional)* — If True, cache data loaded from the underlying datastore in memory as NumPy arrays when accessed to avoid reading from the underlying data-store multiple times. Defaults to True unless you specify the `chunks` argument to use dask, in which case it defaults to False. Does not change the behavior of coordinates corresponding to dimensions, which always load their data from disk into a `pandas.Index`.

- **drop_variables** *(string or iterable, optional)* — A variable or list of variables to exclude from being parsed from the dataset. This may be useful to drop variables with problems or inconsistent values.

- **backend_kwargs** *(dictionary, optional)* — A dictionary of keyword arguments to pass on to the backend. This may be useful when backend options would improve performance or allow user control of dataset processing.

- **use_cftime** *(bool, optional)* — Only relevant if encoded dates come from a standard calendar (e.g., 'gregorian', 'proleptic_gregorian', 'standard', or not specified). If None (default), attempt to decode times to `np.datetime64[ns]` objects; if this is not possible, decode times to `cftime.datetime` objects. If True, always decode times to `cftime.datetime` objects, regardless of whether or not they can be represented using `np.datetime64[ns]` objects. If False, always decode times to `np.datetime64[ns]` objects; if this is not possible raise an error.

- **engine** *({'netcdf4', 'scipy', 'pydap', 'h5netcdf', 'gdal'}, optional)* — Engine to use when reading netCDF files. If not provided, the default engine is chosen based on available dependencies, with a preference for 'netcdf4'.

- **gridfile** *(str)* — The path to a separate grid file or a `xarray.Dataset` instance which may store the coordinates used in `ds`

Returns The list with the interactive objects

Return type `psyplot.data.ArrayList`

See also:

`from_dataset()`, `array_info()`

**is_unstructured**

A boolean for each array whether it is unstructured or not

**logger**

`logging.Logger` of this instance

**names**

Set of the variable in this list

**next_available_name** *(fmt_str='arr{0}', counter=None)*

Create a new array out of the given format string

Parameters

- **format_str** *(str)* — The base string to use. '{0}' will be replaced by a counter

- **counter** *(iterable)* — An iterable where the numbers should be drawn from. If None, `range(100)` is used

Returns A possible name that is not in the current project

Return type `str`
**no_auto_update**

`bool`. Boolean controlling whether the `start_update()` method is automatically called by the `update()` method.

### Examples

You can disable the automatic update via

```python
>>> with data.no_auto_update:
...    data.update(time=1)
...    data.start_update()
```

To permanently disable the automatic update, simply set

```python
>>> data.no_auto_update = True
>>> data.update(time=1)
>>> data.no_auto_update = False  # reenable automatical update
```

**remove** *(arr)*

Removes an array from the list.

**Parameters**

- `arr` *(str or InteractiveBase)* – The array name or the data object in this list to remove

**Raises**

- `ValueError` – If no array with the specified array name is in the list

**rename** *(arr, new_name=True)*

Rename an array to find a name that isn’t already in the list.

**Parameters**

- `arr` *(InteractiveBase)* – A `InteractiveArray` or `InteractiveList` instance whose name shall be checked
- `new_name` *(bool or str)* – If False, and the `arr_name` attribute of the new array is already in the list, a `ValueError` is raised. If True and the `arr_name` attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, `new_name` is set to ‘arr{0}’. If not True, this will be used for renaming (if the array name of `arr` is in use or not). ’{0}’ is replaced by a counter

**Returns**

- `InteractiveBase` – `arr` with changed `arr_name` attribute
- `bool or None` – True, if the array has been renamed, False if not and None if the array is already in the list

**Raises**

- `ValueError` – If it was impossible to find a name that isn’t already in the list
- `ValueError` – If `new_name` is False and the array is already in the list

**start_update** *(draw=None)*

Conduct the registered plot updates.

This method starts the updates from what has been registered by the `update()` method. You can call this method if you did not set the `auto_update` parameter when calling the `update()` method to True and when the `no_auto_update` attribute is True.
Parameters `draw` (`bool` or `None`) – If True, all the figures of the arrays contained in this list will be drawn at the end. If None, it defaults to the `auto_draw` parameter in the `psyplot`.

See also:

`no_auto_update, update()`

`update` (`method='isel', dims={}, fmt={}, replot=False, auto_update=False, draw=None, force=False, todefault=False, enable_post=None, **kwargs`)

Update the coordinates and the plot

This method updates all arrays in this list with the given coordinate values and format options.

Parameters

- `method` (`{'isel', None, 'nearest', ..}`) – Selection method of the xarray.Dataset to be used for setting the variables from the informations in `dims`. If `method` is 'isel', the `xarray.Dataset.isel()` method is used. Otherwise it sets the `method` parameter for the `xarray.Dataset.sel()` method.
- `dims` (`dict`) – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example `dims = {'name': 't2m', 'time': 0}` will result in one plot for the first time step, whereas `dims = {'name': 't2m', 'time': [0, 1]}` will result in two plots, one for the first (time == 0) and one for the second (time == 1) time step.
- `replot` (`bool`) – Boolean that determines whether the data specific format options shall be updated in any case or not. Note, if `dims` is not empty or any coordinate keyword is in `**kwargs`, this will be set to True automatically
- `fmt` (`dict`) – Keys may be any valid format option of the format options in the `plotter`
- `force` (`str, list of str or bool`) – If format option key (i.e. string) or list of format option keys, they are definitely updated whether they changed or not. If True, all the given format options in this call of the are updated.
- `todefault` (`bool`) – If True, all changed format options (except the registered ones) are updated to their default value as stored in the `rc` attribute
- `auto_update` (`bool`) – Boolean determining whether or not the `start_update()` method is called after the end.
- `draw` (`bool` or `None`) – If True, all the figures of the arrays contained in this list will be drawn at the end. If None, it defaults to the `auto_draw` parameter in the `psyplot`.
- `enable_post` (`bool`) – If not None, enable (True) or disable (False) the post format option in the plotters
- `**kwargs` – Any other format option or dimension that shall be updated (additionally to those in `fmt` and `dims`)

Notes

If the `no_auto_update` attribute is True and the given `auto_update` parameter are is False, the update of the plots are registered and conducted at the next call of the `start_update()` method or the next call of this method (if the `auto_update` parameter is then True).

See also:
no_auto_update(), start_update()

with_plotter

The arrays in this instance that are visualized with a plotter

class psyplot.data.CFDecoder (ds=None, x=None, y=None, z=None, t=None)

Bases: object

Class that interpretes the coordinates and attributes accordings to cf-conventions

Methods

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>can_decode (ds, var)</td>
<td>Class method to determine whether the object can be decoded by this decoder class.</td>
</tr>
<tr>
<td>correct_dims (var[, dims, remove])</td>
<td>Expands the dimensions to match the dims in the variable</td>
</tr>
<tr>
<td>decode_coords (ds[, gridfile])</td>
<td>Sets the coordinates and bounds in a dataset</td>
</tr>
<tr>
<td>decode_ds (ds, *args, **kwargs)</td>
<td>Static method to decode coordinates and time informations</td>
</tr>
<tr>
<td>get_cell_node_coord (var[, coords, axis, nans])</td>
<td>Checks whether the bounds in the variable attribute are triangular</td>
</tr>
<tr>
<td>get_coord_idims (coords)</td>
<td>Get the slicers for the given coordinates from the base dataset</td>
</tr>
<tr>
<td>get_decoder (ds, var)</td>
<td>Class method to get the right decoder class that can decode the</td>
</tr>
<tr>
<td>get_idims (arr[, coords])</td>
<td>Get the coordinates in the ds dataset as int or slice</td>
</tr>
<tr>
<td>get_plotbounds (coord[, kind, ignore_shape])</td>
<td>Get the bounds of a coordinate</td>
</tr>
<tr>
<td>get_t (var[, coords])</td>
<td>Get the time coordinate of a variable</td>
</tr>
<tr>
<td>get_tname (var[, coords])</td>
<td>Get the name of the t-dimension</td>
</tr>
<tr>
<td>get_triangles (var[, coords, convert_radian, ...])</td>
<td>Get the triangles for the variable</td>
</tr>
<tr>
<td>get_variable_by_axis (var, axis[, coords])</td>
<td>Return the coordinate matching the specified axis</td>
</tr>
<tr>
<td>get_x (var[, coords])</td>
<td>Get the x-coordinate of a variable</td>
</tr>
<tr>
<td>get_xname (var[, coords])</td>
<td>Get the name of the x-dimension</td>
</tr>
<tr>
<td>get_y (var[, coords])</td>
<td>Get the y-coordinate of a variable</td>
</tr>
<tr>
<td>get_yname (var[, coords])</td>
<td>Get the name of the y-dimension</td>
</tr>
<tr>
<td>get_z (var[, coords])</td>
<td>Get the vertical (z-) coordinate of a variable</td>
</tr>
<tr>
<td>get_zname (var[, coords])</td>
<td>Get the name of the z-dimension</td>
</tr>
<tr>
<td>is_circumpolar (var)</td>
<td>Test if a variable is on a circumpolar grid</td>
</tr>
<tr>
<td>is_triangular (var)</td>
<td>Test if a variable is on a triangular grid</td>
</tr>
<tr>
<td>is_unstructured (var)</td>
<td>Test if a variable is on an unstructured grid</td>
</tr>
<tr>
<td>register_decoder (decoder_class[, pos])</td>
<td>Register a new decoder</td>
</tr>
<tr>
<td>standardize_dims (var[, dims])</td>
<td>Replace the coordinate names through x, y, z and t</td>
</tr>
</tbody>
</table>

Attributes

logger

logging.Logger of this instance

classmethod can_decode (ds, var)

Class method to determine whether the object can be decoded by this decoder class.

Parameters

- **ds** (xarray.Dataset) – The dataset that contains the given var
- **var** (xarray.Variable or xarray.DataArray) – The array to decode
Returns True if the decoder can decode the given array `var`. Otherwise False

Return type `bool`

Notes

The default implementation returns True for any argument. Subclass this method to be specific on what type of data your decoder can decode

`correct_dims(var, dims={}, remove=True)`

Expands the dimensions to match the dims in the variable

Parameters

• `var (xarray.Variable)` – The variable to get the data for
• `dims (dict)` – a mapping from dimension to the slices
• `remove (bool)` – If True, dimensions in `dims` that are not in the dimensions of `var` are removed

`static decode_coords(ds, gridfile=None)`

Sets the coordinates and bounds in a dataset

This static method sets those coordinates and bounds that are marked marked in the netCDF attributes as coordinates in `ds` (without deleting them from the variable attributes because this information is necessary for visualizing the data correctly)

Parameters

• `ds (xarray.Dataset)` – The dataset to decode
• `gridfile (str)` – The path to a separate grid file or a xarray.Dataset instance which may store the coordinates used in `ds`

Returns `ds` with additional coordinates

Return type `xarray.Dataset`

`classmethod decode_ds(ds, *args, **kwargs)`

Static method to decode coordinates and time informations

This method interpretes absolute time informations (stored with units `'day as %Y%m%d.%f'`) and coordinates

Parameters

• `ds (xarray.Dataset)` – The dataset to decode
• `gridfile (str)` – The path to a separate grid file or a xarray.Dataset instance which may store the coordinates used in `ds`
• `decode_times (bool, optional)` – If True, decode times encoded in the standard NetCDF datetime format into datetime objects. Otherwise, leave them encoded as numbers.
• `decode_coords (bool, optional)` – If True, decode the ‘coordinates’ attribute to identify coordinates in the resulting dataset.

Returns The decoded dataset

Return type `xarray.Dataset`

`get_cell_node_coord(var, coords=None, axis='x', nans=None)`

Checks whether the bounds in the variable attribute are triangular
Parameters

- **var** (*xarray.Variable or xarray.DataArray*) – The variable to check
- **coords** (*dict*) – Coordinates to use. If None, the coordinates of the dataset in the *ds* attribute are used.
- **axis** (*{'x', 'y'}*) – The spatial axis to check
- **nans** (*{None, 'skip', 'only'}*) – Determines whether values with nan shall be left (None), skipped ('skip') or shall be the only one returned ('only')

Returns the bounds coordinate (if existent)

**Return type** `xarray.DataArray or None`

**get_coord_idims**(coords)
Get the slicers for the given coordinates from the base dataset

This method converts *coords* to slicers (list of integers or *slice* objects)

Parameters

- **coords** (*dict*) – A subset of the *ds.coords* attribute of the base dataset *ds*

Returns Mapping from coordinate name to integer, list of integer or slice

**Return type** `dict`

**classmethod get_decoder**(ds, var)
Class method to get the right decoder class that can decode the given dataset and variable

Parameters

- **ds** (*xarray.Dataset*) – The dataset that contains the given var
- **var** (*xarray.Variable or xarray.DataArray*) – The array to decode

Returns The decoder for the given dataset that can decode the variable *var*

**Return type** `CFDecoder`

**get_idims**(arr, coords=None)
Get the coordinates in the *ds* dataset as int or slice

This method returns a mapping from the coordinate names of the given *arr* to an integer, slice or an array of integer that represent the coordinates in the *ds* dataset and can be used to extract the given *arr* via the *xarray.Dataset.isel()* method.

Parameters

- **arr** (*xarray.DataArray*) – The data array for which to get the dimensions as integers, slices or list of integers from the dataset in the *base* attribute
- **coords** (*iterable*) – The coordinates to use. If not given all coordinates in the *arr.coords* attribute are used

Returns Mapping from coordinate name to integer, list of integer or slice

**Return type** `dict`

See also: `xarray.Dataset.isel()`, `InteractiveArray.idims()`

**get_plotbounds**(coord, kind=None, ignore_shape=False)
Get the bounds of a coordinate

This method first checks the 'bounds' attribute of the given *coord* and if it fails, it calculates them.
Parameters

- **coord** (*xarray.Coordinate*) – The coordinate to get the bounds for
- **kind** (*str*) – The interpolation method (see `scipy.interpolate.interp1d()`) that is used in case of a 2-dimensional coordinate
- **ignore_shape** (*bool*) – If True and the `coord` has a 'bounds' attribute, this attribute is returned without further check. Otherwise it is tried to bring the 'bounds' into a format suitable for (e.g.) the `matplotlib.pyplot.pcolormesh()` function.

Returns bounds – The bounds with the same number of dimensions as `coord` but one additional array (i.e. if `coord` has shape (4, ), `bounds` will have shape (5, ) and if `coord` has shape (4, 5), `bounds` will have shape (5, 6)

Return type `np.ndarray`

get_t (var, coords=None)

Get the time coordinate of a variable

This method searches for the time coordinate in the `ds`. It first checks whether there is one dimension that holds an 'axis' attribute with 'T', otherwise it looks whether there is an intersection between the t attribute and the variables dimensions, otherwise it returns the coordinate corresponding to the first dimension of `var`

Possible types

- **var** (*xarray.Variable*) – The variable to get the time coordinate for
- **coords** (*dict*) – Coordinates to use. If None, the coordinates of the dataset in the `ds` attribute are used.

Returns The time coordinate or None if no time coordinate could be found

Return type `xarray.Coordinate or None`

get_tname (var, coords=None)

Get the name of the t-dimension

This method gives the name of the time dimension

Parameters

- **var** (*xarray.Variables*) – The variable to get the dimension for
- **coords** (*dict*) – The coordinates to use for checking the axis attribute. If None, they are not used

Returns The coordinate name or None if no time coordinate could be found

Return type `str or None`

See also:

get_t()

get_triangles (var, coords=None, convert_radian=True, copy=False, src_crs=None, target_crs=None, nans=None, stacklevel=1)

Get the triangles for the variable

Parameters

- **var** (*xarray.Variable or xarray.DataArray*) – The variable to use
• `coords (dict)` – Alternative coordinates to use. If None, the coordinates of the `ds` dataset are used

• `convert_radian (bool)` – If True and the coordinate has units in ‘radian’, those are converted to degrees

• `copy (bool)` – If True, vertice arrays are copied

• `src_crs (cartopy.crs.Crs)` – The source projection of the data. If not None, a transformation to the given `target_crs` will be done

• `target_crs (cartopy.crs.Crs)` – The target projection for which the triangles shall be transformed. Must only be provided if the `src_crs` is not None.

• `nans (None, 'skip', 'only')` – Determines whether values with nan shall be left (None), skipped (‘skip’) or shall be the only one returned (‘only’)

Returns The spatial triangles of the variable

Return type `matplotlib.tri.Triangulation`

Raises `ValueError` – If `src_crs` is not None and `target_crs` is None

---

### get_variable_by_axis

`get_variable_by_axis (var, axis, coords=None)`

Return the coordinate matching the specified axis

This method uses to 'axis' attribute in coordinates to return the corresponding coordinate of the given variable

#### Possible types

• `var (xarray.Variable)` – The variable to get the dimension for

• `axis ({'x', 'y', 'z', 't'})` – The axis string that identifies the dimension

• `coords (dict)` – Coordinates to use. If None, the coordinates of the dataset in the `ds` attribute are used.

Returns The coordinate for `var` that matches the given `axis` or None if no coordinate with the right `axis` could be found.

Return type `xarray.Coordinate or None`

#### Notes

This is a rather low-level function that only interpretes the CFConvention. It is used by the `get_x()`, `get_y()`, `get_z()` and `get_t()` methods

---

**Warning:** If None of the coordinates have an 'axis' attribute, we use the 'coordinate' attribute of `var` (if existent). Since however the CF Conventions do not determine the order on how the coordinates shall be saved, we try to use a pattern matching for latitude ('lat') and longitude ('lon'). If this patterns do not match, we interpret the coordinates such that x: -1, y: -2, z: -3. This is all not very safe for awkward dimension names, but works for most cases. If you want to be a hundred percent sure, use the x, y, z and t attribute.

See also:

`get_x(), get_y(), get_z(), get_t()`
get_x (var, coords=None)
Get the x-coordinate of a variable

This method searches for the x-coordinate in the ds. It first checks whether there is one dimension that holds an 'axis' attribute with 'X', otherwise it looks whether there is an intersection between the x attribute and the variables dimensions, otherwise it returns the coordinate corresponding to the last dimension of var

Possible types

- var (xarray.Variable) – The variable to get the x-coordinate for
- coords (dict) – Coordinates to use. If None, the coordinates of the dataset in the ds attribute are used.

Returns The y-coordinate or None if it could be found
Return type xarray.Coordinate or None

get_xname (var, coords=None)
Get the name of the x-dimension

This method gives the name of the x-dimension (which is not necessarily the name of the coordinate if the variable has a coordinate attribute)

Parameters

- var (xarray VARIABLES) – The variable to get the dimension for
- coords (dict) – The coordinates to use for checking the axis attribute. If None, they are not used

Returns The coordinate name
Return type str

See also:
get_x()

get_y (var, coords=None)
Get the y-coordinate of a variable

This method searches for the y-coordinate in the ds. It first checks whether there is one dimension that holds an 'axis' attribute with 'Y', otherwise it looks whether there is an intersection between the y attribute and the variables dimensions, otherwise it returns the coordinate corresponding to the second last dimension of var (or the last if the dimension of var is one-dimensional)

Possible types

- var (xarray.Variable) – The variable to get the y-coordinate for
- coords (dict) – Coordinates to use. If None, the coordinates of the dataset in the ds attribute are used.

Returns The y-coordinate or None if it could be found
Return type xarray.Coordinate or None
**get_yname** *(var, coords=None)*

Get the name of the y-dimension

This method gives the name of the y-dimension (which is not necessarily the name of the coordinate if the variable has a coordinate attribute)

**Parameters**

- **var** *(xarray.Variable)* – The variable to get the dimension for
- **coords** *(dict)* – The coordinates to use for checking the axis attribute. If None, they are not used

**Returns** The coordinate name

**Return type** str

**See also:**

*get_y()*

**get_z** *(var, coords=None)*

Get the vertical (z-) coordinate of a variable

This method searches for the z-coordinate in the `ds`. It first checks whether there is one dimension that holds an 'axis' attribute with 'Z', otherwise it looks whether there is an intersection between the z attribute and the variables dimensions, otherwise it returns the coordinate corresponding to the third last dimension of `var` (or the second last or last if var is two or one-dimensional)

**Possible types**

- **var** *(xarray.Variable)* – The variable to get the z-coordinate for
- **coords** *(dict)* – Coordinates to use. If None, the coordinates of the dataset in the `ds` attribute are used.

**Returns** The z-coordinate or None if no z coordinate could be found

**Return type** xarray.Coordinate or None

**get_zname** *(var, coords=None)*

Get the name of the z-dimension

This method gives the name of the z-dimension (which is not necessarily the name of the coordinate if the variable has a coordinate attribute)

**Parameters**

- **var** *(xarray.Variable)* – The variable to get the dimension for
- **coords** *(dict)* – The coordinates to use for checking the axis attribute. If None, they are not used

**Returns** The coordinate name or None if no vertical coordinate could be found

**Return type** str or None

**See also:**

*get_z()*

**is_circumpolar** *(var)*

Test if a variable is on a circumpolar grid

**Parameters** **var** *(xarray.Variable or xarray.DataArray)* – The variable to check
Returns True, if the grid is triangular, else False
Return type bool

is_triangular (var)
Test if a variable is on a triangular grid
This method first checks the grid_type attribute of the variable (if existent) whether it is equal to "unstructured", then it checks whether the bounds are not two-dimensional.

Parameters var (xarray.Variable or xarray.DataArray) – The variable to check
Returns True, if the grid is triangular, else False
Return type bool

is_unstructured (var)
Test if a variable is on an unstructured grid

Parameters var (xarray.Variable or xarray.DataArray) – The variable to check
Returns True, if the grid is triangular, else False
Return type bool

Notes
Currently this is the same as is_triangular() method, but may change in the future to support hexagonal grids

logger
logging.Logger of this instance

static register_decoder (decoder_class, pos=0)
Register a new decoder
This function registeres a decoder class to use

Parameters
• decoder_class (type) – The class inherited from the CFDecoder
• pos (int) – The position where to register the decoder (by default: the first position

standardize_dims (var, dims={})
Replace the coordinate names through x, y, z and t

Parameters
• var (xarray.Variable) – The variable to use the dimensions of
• dims (dict) – The dictionary to use for replacing the original dimensions

Returns The dictionary with replaced dimensions
Return type dict

class psyplot.data.DatasetAccessor (ds)
Bases: object

A dataset accessor to interface with the psyplot package
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy ([deep])</td>
<td>Copy the array</td>
</tr>
</tbody>
</table>

Continued on next page
Table 32 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create_list(*args, **kwargs)</code></td>
<td>Create a <code>psyplot.data.ArrayList</code> with arrays from this dataset</td>
</tr>
<tr>
<td><code>to_array(*args, **kwargs)</code></td>
<td>Convert this dataset into an xarray.DataArray</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data_store</code></td>
<td>The <code>xarray.backends.common.AbstractStore</code> used to save the</td>
</tr>
<tr>
<td><code>filename</code></td>
<td>The name of the file that stores this dataset</td>
</tr>
<tr>
<td><code>num</code></td>
<td>A unique number for the dataset</td>
</tr>
<tr>
<td><code>plot</code></td>
<td>An object to generate new plots from this dataset</td>
</tr>
</tbody>
</table>

**copy** *(deep=False)*

Copy the array

This method returns a copy of the underlying array in the `arr` attribute. It is more stable because it creates a new `psy` accessor

**create_list(*args, **kwargs)**

Create a `psyplot.data.ArrayList` with arrays from this dataset

**Parameters**

- `base (xarray.Dataset)` – Dataset instance that is used as reference
- `method (['isel', None, 'nearest', ...])` – Selection method of the `xarray.Dataset` to be used for setting the variables from the informations in `dims`. If `method` is `isel`, the `xarray.Dataset.isel()` method is used. Otherwise it sets the `method` parameter for the `xarray.Dataset.sel()` method.
- `auto_update (bool)` – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not. See also the `no_auto_update` attribute. If None, the value from the `lists.auto_update` key in the `psyplot.rcParams` dictionary is used.
- `prefer_list (bool)` – If True and multiple variable names per array are found, the `InteractiveList` class is used. Otherwise the arrays are put together into one `InteractiveArray`.
- `default_slice (indexer)` – Index (e.g. 0 if `method` is `isel`) that shall be used for dimensions not covered by `dims` and `furtherdims`. If None, the whole slice will be used.
- `decoder (CFDecoder)` – The decoder that shall be used to decoder the `base` dataset
- `squeeze (bool, optional)` – Default True. If True, and the created arrays have an axes with length 1, it is removed from the dimension list (e.g. an array with shape (3, 4, 1, 5) will be squeezed to shape (3, 4, 5))
- `attrs (dict, optional)` – Meta attributes that shall be assigned to the selected data arrays (additional to those stored in the `base` dataset)
- `load (bool or dict)` – If True, load the data from the dataset using the `xarray.DataArray.load()` method. If `dict`, those will be given to the above mentioned `load` method

**Other Parameters**

- `arr_names (string, list of strings or dictionary)` – Set the unique array names of the resulting arrays and (optionally) dimensions.
– if string: same as list of strings (see below). Strings may include {0} which will be replaced by a counter.
– list of strings: those will be used for the array names. The final number of dictionaries in the return depend in this case on the dims and **furtherdims
– dictionary: Then nothing happens and an OrderedDict version of arr_names is returned.

• sort (list of strings) – This parameter defines how the dictionaries are ordered. It has no effect if arr_names is a dictionary (use a OrderedDict for that). It can be a list of dimension strings matching to the dimensions in dims for the variable.
• dims (dict) – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example dims = {'name': 't2m', ‘time’: 0} will result in one plot for the first time step, whereas dims = {'name': 't2m', ‘time’: [0, 1]} will result in two plots, one for the first (time == 0) and one for the second (time == 1) time step.
  • ‘**kwargs’ – The same as dims (those will update what is specified in dims)

Returns The list with the specified InteractiveArray instances that hold a reference to the given base

Return type ArrayList

See also:

psyplot.data.ArrayList.from_dataset()

data_store
The xarray.backends.common.AbstractStore used to save the dataset

filename
The name of the file that stores this dataset

num
A unique number for the dataset

plot
An object to generate new plots from this dataset

To make a 2D-plot with the psy-simple plugin, you can just type

```python
project = ds.psy.plot.plot2d(name='variable-name')
```

It will create a new subproject with the extracted and visualized data.

See also:

psyplot.project.DatasetPlotter for the different plot methods

**to_array** (*args, **kwargs)
Convert this dataset into an xarray.DataArray

The data variables of this dataset will be broadcast against each other and stacked along the first axis of the new array. All coordinates of this dataset will remain coordinates.

Parameters

• dim (str, optional) – Name of the new dimension.
• name (str, optional) – Name of the new data array.
Returns array
Return type: xarray.DataArray

class psyplot.data.InteractiveArray(xarray_obj, *args, **kwargs)
Bases: psyplot.data.InteractiveBase

Interactive psyplot accessor for the data array

This class keeps reference to the base xarray.Dataset where the array.DataArray originates from and enables to switch between the coordinates in the array. Furthermore it has a plotter attribute to enable interactive plotting via an psyplot.plotter.Plotter instance.

The *args and **kwargs are essentially the same as for the xarray.DataArray method, additional **kwargs are described below.

Other Parameters

- **base** (xarray.Dataset) – Default: None. Dataset that serves as the origin of the data contained in this DataArray instance. This will be used if you want to update the coordinates via the update() method. If None, this instance will serve as a base as soon as it is needed.
- **decoder** (psyplot.CFDecoder) – The decoder that decodes the base dataset and is used to get bounds. If not given, a new CFDecoder is created
- **idims** (dict) – Default: None. dictionary with integer values and/or slices in the base dictionary. If not given, they are determined automatically
- **plotter** (Plotter) – Default: None. Interactive plotter that makes the plot via formatoption keywords.
- **arr_name** (str) – Default: 'data'. unique string of the array
- **auto_update** (bool) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the update() method or not. See also the no_auto_update attribute. If None, the value from the 'lists.auto_update' key in the psyplot.rcParams dictionary is used.

Attributes

<table>
<thead>
<tr>
<th>base</th>
<th>Base dataset this instance gets its data from</th>
</tr>
</thead>
<tbody>
<tr>
<td>base_variables</td>
<td>A mapping from the variable name to the variable in the base dataset.</td>
</tr>
<tr>
<td>decoder</td>
<td>The decoder of this array</td>
</tr>
<tr>
<td>idims</td>
<td>Coordinates in the base dataset as int or slice</td>
</tr>
<tr>
<td>iter_base_variables</td>
<td>An iterator over the base variables in the base dataset</td>
</tr>
<tr>
<td>logger</td>
<td>logging.Logger of this instance</td>
</tr>
<tr>
<td>onbasechange</td>
<td>Signal to be emitted when the base of the object changes</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>copy([deep])</th>
<th>Copy the array</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldmean([keepdims])</td>
<td>Calculate the weighted mean over the x- and y-dimension</td>
</tr>
<tr>
<td>fldpctl(q[, keepdims])</td>
<td>Calculate the percentiles along the x- and y-dimensions</td>
</tr>
</tbody>
</table>

Continued on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fldstd([keepdims])</code></td>
<td>Calculate the weighted standard deviation over x- and y-dimension</td>
</tr>
<tr>
<td><code>get_coord(what[, base])</code></td>
<td>The x-coordinate of this data array</td>
</tr>
<tr>
<td><code>get_dim(what[, base])</code></td>
<td>The name of the x-dimension of this data array</td>
</tr>
<tr>
<td><code>gridweights([keepdims, keepshape, use_cdo])</code></td>
<td>Calculate the cell weights for each grid cell</td>
</tr>
<tr>
<td><code>init_accessor([base, idims, decoder])</code></td>
<td>Initialize the accessor instance</td>
</tr>
<tr>
<td><code>isel(*args, **kwargs)</code></td>
<td>Return a new DataArray whose dataset is given by integer indexing along the specified dimension(s).</td>
</tr>
<tr>
<td><code>sel(*args, **kwargs)</code></td>
<td>Return a new DataArray whose dataset is given by selecting index labels along the specified dimension(s).</td>
</tr>
<tr>
<td><code>start_update([draw, queues])</code></td>
<td>Conduct the formerly registered updates</td>
</tr>
<tr>
<td><code>to_interactive_list()</code></td>
<td>Return a <code>InteractiveList</code> that contains this object</td>
</tr>
<tr>
<td><code>update([method, dims, fmt, replot, ...])</code></td>
<td>Update the coordinates and the plot</td>
</tr>
</tbody>
</table>

**base**
Base dataset this instance gets its data from

**base_variables**
A mapping from the variable name to the variable in the `base` dataset.

**copy (deep=False)**
Copy the array

This method returns a copy of the underlying array in the `arr` attribute. It is more stable because it creates a new `psy` accessor

**decoder**
The decoder of this array

** fldmean (keepdims=False)**
Calculate the weighted mean over the x- and y-dimension

This method calculates the weighted mean of the spatial dimensions. Weights are calculated using the `gridweights()` method, missing values are ignored. x- and y-dimensions are identified using the decoder’s `CFDecoder.get_xname()` and `get_yname()` methods.

**Parameters**

**keepdims (bool)** – If True, the dimensionality of this array is maintained

**Returns**
The computed fldmeans. The dimensions are the same as in this array, only the spatial dimensions are omitted if `keepdims` is False.

**Return type** `xr.DataArray`

See also:

`fldstd()` For calculating the weighted standard deviation

`fldpctl1()` For calculating weighted percentiles

**fldpctl1 (q, keepdims=False)**
Calculate the percentiles along the x- and y-dimensions

This method calculates the specified percentiles along the given dimension. Percentiles are weighted by the `gridweights()` method and missing values are ignored. x- and y-dimensions are estimated through the decoder’s `CFDecoder.get_xname()` and `get_yname()` methods

**Parameters**
psyplot Documentation, Release 1.2.1

- `q(float or list of floats between 0 and 100)` – The quantiles to estimate
- `keepdims (bool)` – If True, the number of dimensions of the array are maintained

**Returns** The data array with the dimensions. If q is a list or keepdims is True, the first dimension will be the percentile ‘pctl’. The other dimensions are the same as in this array, only the spatial dimensions are omitted if keepdims is False.

**Return type** `xr.DataArray`

See also:

- `fldstd()` For calculating the weighted standard deviation
- `fldmean()` For calculating the weighted mean

**Warning:** This method does load the entire array into memory! So take care if you handle big data.

`fldstd(keepdims=False)`
Calculate the weighted standard deviation over x- and y-dimension

This method calculates the weighted standard deviation of the spatial dimensions. Weights are calculated using the `gridweights()` method, missing values are ignored. x- and y-dimensions are identified using the decoder’s :meth:`~CFDecoder.get_xname and get_yname()` methods.

**Parameters** `keepdims (bool)` – If True, the dimensionality of this array is maintained

**Returns** The computed standard deviations. The dimensions are the same as in this array, only the spatial dimensions are omitted if keepdims is False.

**Return type** `xr.DataArray`

See also:

- `fldmean()` For calculating the weighted mean
- `fldpctl()` For calculating weighted percentiles

`get_coord (what, base=False)`
The x-coordinate of this data array

**Parameters**

- `what ('t', 'x', 'y', 'z')` – The letter of the axis
- `base (bool)` – If True, use the base variable in the base dataset.

`get_dim (what, base=False)`
The name of the x-dimension of this data array

**Parameters**

- `what ('t', 'x', 'y', 'z')` – The letter of the axis
- `base (bool)` – If True, use the base variable in the base dataset.

`gridweights (keepdims=False, keepshape=False, use_cdo=None)`
Calculate the cell weights for each grid cell

**Parameters**

- `keepdims (bool)` – If True, keep the number of dimensions
• **keepshape** (*bool*) – If True, keep the exact shape as the source array and the missing values in the array are masked

• **use_cdo** (*bool** or **None*) – If True, use Climate Data Operators (CDOs) to calculate the weights. Note that this is used automatically for unstructured grids. If None, it depends on the 'gridweights.use_cdo' item in the psyplot.rcParams.

**Returns** The 2D-DataArray with the grid weights

**Return type** xarray.DataArray

### idims

Coordinates in the *base* dataset as int or slice

This attribute holds a mapping from the coordinate names of this array to an integer, slice or an array of integer that represent the coordinates in the *base* dataset

### init_accessor (*base=None, idims=None, decoder=None, *args, **kwargs*)

Initialize the accessor instance

This method initializes the accessor

**Parameters**

• **base** (*xr.Dataset*) – The base dataset for the data

• **idims** (*dict*) – A mapping from dimension name to indices. If not provided, it is calculated when the *idims* attribute is accessed

• **decoder** (*CFDecoder*) – The decoder of this object

• **%(InteractiveBase.parameters)s** –

### isel (*args, **kwargs*)

Return a new DataArray whose dataset is given by integer indexing along the specified dimension(s).

**See also:**

Dataset.isel(), DataArray.sel()

### iter_base_variables

An iterator over the base variables in the *base* dataset

### logger

`logging.Logger` of this instance

### onbasechange

`Signal` to be emitted when the base of the object changes

### sel (*args, **kwargs*)

Return a new DataArray whose dataset is given by selecting index labels along the specified dimension(s).

**Warning:** Do not try to assign values when using any of the indexing methods *isel* or *sel*:

```python
da = xr.DataArray([0, 1, 2, 3], dims=['x'])
# DO NOT do this
da.isel(x=[0, 1, 2])[1] = -1
```

Assigning values with the chained indexing using .sel or .isel fails silently.

**See also:**

Dataset.sel(), DataArray.isel()
**start_update** *(draw=\texttt{None}, queues=\texttt{None})*

Conduct the formerly registered updates

This method conducts the updates that have been registered via the **update**() method. You can call this method if the **no_auto_update** attribute of this instance is True and the **auto_update** parameter in the **update**() method has been set to False.

**Parameters**

- **draw** *(bool or \texttt{None})* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘\texttt{auto_draw}’ parameter in the psyplot. **rcParams** dictionary.

- **queues** *(list of Queue.Queue instances)* – The queues that are passed to the psyplot.plotter.Plotter.start_update() method to ensure a thread-safe update. It can be None if only one single plotter is updated at the same time. The number of jobs that are taken from the queue is determined by the **_njobs**() method. Note that there this parameter is automatically configured when updating from a Project.

**Returns** A boolean indicating whether a redrawing is necessary or not

**Return type** bool

**See also:**

no_auto_update, update()

to_interactive_list()

Return a InteractiveList that contains this object

**update** *(method='isel', dims={}, fmt={}, replot=False, auto_update=False, draw=\texttt{None}, force=False, todefault=False, **kwargs)*

Update the coordinates and the plot

This method updates all arrays in this list with the given coordinate values and formatoptions.

**Parameters**

- **method** *(\texttt{\{'isel\', None, 'nearest', \ldots\}})* – Selection method of the xarray.Dataset to be used for setting the variables from the informations in **dims**. If **method** is ‘isel’, the xarray.Dataset.isel() method is used. Otherwise it sets the **method** parameter for the xarray.Dataset.sel() method.

- **dims** *(dict)* – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example ** dims = \{'name\': ‘t2m’, ‘time\': 0\}** will result in one plot for the first time step, whereas ** dims = \{'name\': ‘t2m’, ‘time\': [0, 1]\}** will result in two plots, one for the first (time == 0) and one for the second (time == 1) time step.

- **replot** *(bool)* – Boolean that determines whether the data specific formatoptions shall be updated in any case or not. Note, if **dims** is not empty or any coordinate keyword is in ****kwargs, this will be set to True automatically.

- **fmt** *(dict)* – Keys may be any valid formatoption of the formatoptions in the plotter

- **force** *(str, list of str or bool)* – If formatoption key (i.e. string) or list of formatoption keys, they are definitely updated whether they changed or not. If True, all the given formatoptions in this call of the are update() method are updated.

- **todefault** *(bool)* – If True, all changed formatoptions (except the registered ones) are updated to their default value as stored in the **rc** attribute.
• **auto_update** *(bool)* – Boolean determining whether or not the *start_update()* method is called after the end.

• **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the *psyplot.rcParams* dictionary.

• **queues** *(list of Queue.Queue instances)* – The queues that are passed to the *psyplot.plotter.Plotter.start_update()* method to ensure a thread-safe update. It can be None if only one single plotter is updated at the same time. The number of jobs that are taken from the queue is determined by the *_njobs()* attribute. Note that there this parameter is automatically configured when updating from a *Project*.

• ****kwargs** – Any other formatoption or dimension that shall be updated (additionally to those in *fmt* and *dims*)

**Notes**

If the *no_auto_update* attribute is True and the given *auto_update* parameter are is False, the update of the plots are registered and conducted at the next call of the *start_update()* method or the next call of this method (if the *auto_update* parameter is then True).

class psyplot.data.InteractiveBase *(plotter=None, arr_name='arr0', auto_update=None)*

Bases: object

Class for the communication of a data object with a suitable plotter

This class serves as an interface for data objects (in particular as a base for *InteractiveArray* and *InteractiveList*) to communicate with the corresponding *Plotter* in the *plotter* attribute.

**Parameters**

• **plotter** *(Plotter)* – Default: None. Interactive plotter that makes the plot via formatoption keywords.

• **arr_name** *(str)* – Default: 'data'. unique string of the array

• **auto_update** *(bool)* – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the *update()* method or not. See also the *no_auto_update* attribute. If None, the value from the 'lists.auto_update' key in the *psyplot.rcParams* dictionary is used.

**Attributes**

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr_name</td>
<td>str</td>
<td>The internal name of the <em>InteractiveBase</em></td>
</tr>
<tr>
<td>ax</td>
<td></td>
<td>The matplotlib axes the plotter of this data object plots on</td>
</tr>
<tr>
<td>block_signals</td>
<td></td>
<td>Block the emitting of signals of this instance</td>
</tr>
<tr>
<td>logger</td>
<td></td>
<td><em>logging.Logger</em> of this instance</td>
</tr>
<tr>
<td>no_auto_update</td>
<td>bool</td>
<td>Boolean controlling whether the <em>start_update()</em> method is called</td>
</tr>
<tr>
<td>onupdate</td>
<td><em>Signal</em></td>
<td>to be emitted when the object has been updated</td>
</tr>
<tr>
<td>plot</td>
<td></td>
<td>An object to visualize this data object</td>
</tr>
<tr>
<td>plotter</td>
<td></td>
<td><em>psyplot.plotter.Plotter</em> instance that makes the interactive</td>
</tr>
</tbody>
</table>
Methods

**start_update**([draw, queues])

Conduct the formerly registered updates

**to_interactive_list()**

Return a `InteractiveList` that contains this object

**update**([fmt, replot, draw, auto_update, ...])

Update the coordinates and the plot

---

**arr_name**

str. The internal name of the `InteractiveBase`

**ax**

The matplotlib axes the plotter of this data object plots on

**block_signals**

Block the emitting of signals of this instance

**logger**

`logging.Logger` of this instance

**no_auto_update**

bool. Boolean controlling whether the `start_update()` method is automatically called by the `update()` method

---

Examples

You can disable the automatic update via

```python
>>> with data.no_auto_update:
...     data.update(time=1)
...     data.start_update()
```

To permanently disable the automatic update, simply set

```python
>>> data.no_auto_update = True
>>> data.update(time=1)
>>> data.no_auto_update = False  # reenable automatical update
```

**onupdate**

`Signal` to be emitted when the object has been updated

**plot**

An object to visualize this data object

To make a 2D-plot with the `psy-simple` plugin, you can just type

```python
plotter = da.psy.plot.plot2d()
```

It will create a new `psyplot.plotter.Plotter` instance with the extracted and visualized data.

See also:

`psyplot.project.DataArrayPlotter` for the different plot methods

**plotter**

`psyplot.plotter.Plotter` instance that makes the interactive plotting of the data
**start_update** *(draw=None, queues=None)*

Conduct the formerly registered updates

This method conducts the updates that have been registered via the `update()` method. You can call this method if the `no_auto_update` attribute of this instance and the `auto_update` parameter in the `update()` method has been set to False.

**Parameters**

- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the `psyplot.rcParams` dictionary.

- **queues** *(list of Queue.Queue instances)* – The queues that are passed to the `psyplot.plotter.Plotter.start_update()` method to ensure a thread-safe update. It can be None if only one single plotter is updated at the same time. The number of jobs that are taken from the queue is determined by the `_njobs()` attribute. Note that there this parameter is automatically configured when updating from a `Project`.

**Returns** A boolean indicating whether a redrawing is necessary or not

**Return type** bool

**See also:** `no_auto_update`, `update()`

**to_interactive_list()**

Return a `InteractiveList` that contains this object

**update** *(fmt={}, replot=False, draw=None, auto_update=False, force=False, todefault=False, **kwargs)*

Update the coordinates and the plot

This method updates all arrays in this list with the given coordinate values and format options.

**Parameters**

- **replot** *(bool)* – Boolean that determines whether the data specific format options shall be updated in any case or not. Note, if `dims` is not empty or any coordinate keyword is in `**kwargs`, this will be set to True automatically.

- **fmt** *(dict)* – Keys may be any valid format option of the format options in the `plotter`.

- **force** *(str, list of str or bool)* – If format option key (i.e. string) or list of format option keys, they are definitely updated whether they changed or not. If True, all the given format options in this call of the `update()` method are updated.

- **todefault** *(bool)* – If True, all changed format options (except the registered ones) are updated to their default value as stored in the `rc` attribute.

- **auto_update** *(bool)* – Boolean determining whether or not the `start_update()` method is called at the end. This parameter has no effect if the `no_auto_update` attribute is set to True.

- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the `psyplot.rcParams` dictionary.

- ****kwargs** – Any other format option that shall be updated (additionally to those in `fmt`)
Notes

If the `no_auto_update` attribute is True and the given `auto_update` parameter are is False, the update of the plots are registered and conducted at the next call of the `start_update()` method or the next call of this method (if the `auto_update` parameter is then True).

```python
class psyplot.data.InteractiveList(*args, **kwargs)
Bases: psyplot.data.ArrayList, psyplot.data.InteractiveBase

List of InteractiveArray instances that can be plotted itself

This class combines the ArrayList and the interactive plotting through psyplot.plotter.Plotter classes. It is mainly used by the psyplot.plotter.simple module

Parameters

• **iterable** (iterable) – The iterable (e.g. another list) defining this list
• **attrs** (dict-like or iterable, optional) – Global attributes of this list
• **auto_update** (bool) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not. See also the `no_auto_update` attribute. If None, the value from the 'lists.auto_update' key in the psyplot.rcParams dictionary is used.
• **new_name** (bool or str) – If False, and the `arr_name` attribute of the new array is already in the list, a ValueError is raised. If True and the `arr_name` attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, `new_name` is set to 'arr{0}'. If not True, this will be used for renaming (if the array name of `arr` is in use or not). '{0}' is replaced by a counter
• **plotter** (Plotter) – Default: None. Interactive plotter that makes the plot via formatoption keywords.
• **arr_name** (str) – Default: 'data'. unique string of the array

Methods

```append(*args, **kwargs)``` Append a new array to the list
```extend(*args, **kwargs)``` Add further arrays from an iterable to this list
```from_dataset(*args, **kwargs)``` Create an InteractiveList instance from the given base dataset
```start_update([draw, queues])``` Conduct the formerly registered updates
```to_dataframe()``` Return a dataframe
```to_interactive_list()``` Return a InteractiveList that contains this object

Attributes

```logger``` logging.Logger of this instance
```no_auto_update``` bool. Boolean controlling whether the `start_update()`
```psy``` Return the list itself

append(*args, **kwargs)

Append a new array to the list

Parameters

```
• **value** (*InteractiveBase*) – The data object to append to this list

• **new_name** (*bool or str*) – If False, and the *arr_name* attribute of the new array is already in the list, a *ValueError* is raised. If True and the *arr_name* attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, *new_name* is set to ‘*arr{0}’*. If not True, this will be used for renaming (if the array name of *arr* is in use or not). '{0}' is replaced by a counter

**Raises**

• *ValueError* – If it was impossible to find a name that isn’t already in the list

• *ValueError* – If *new_name* is False and the array is already in the list

**See also:**

`list.append()`, `extend()`, `rename()`

**extend** (*args, **kwargs*)

Add further arrays from an iterable to this list

**Parameters**

• **iterable** – Any iterable that contains *InteractiveBase* instances

• **new_name** (*bool or str*) – If False, and the *arr_name* attribute of the new array is already in the list, a *ValueError* is raised. If True and the *arr_name* attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, *new_name* is set to ‘*arr{0}’*. If not True, this will be used for renaming (if the array name of *arr* is in use or not). '{0}' is replaced by a counter

**Raises**

• *ValueError* – If it was impossible to find a name that isn’t already in the list

• *ValueError* – If *new_name* is False and the array is already in the list

**See also:**

`list.extend()`, `append()`, `rename()`

**classmethod from_dataset** (*args, **kwargs*)

Create an *InteractiveList* instance from the given base dataset

**Parameters**

• **base** (*xarray.Dataset*) – Dataset instance that is used as reference

• **method** ([*isel*, *None*, *'nearest', ..]) – Selection method of the xarray.Dataset to be used for setting the variables from the informations in *dims*. If *method* is *'isel'*, the *xarray.Dataset.isel()* method is used. Otherwise it sets the *method* parameter for the *xarray.Dataset.sel()* method.

• **auto_update** (*bool*) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the *update()* method or not. See also the *no_auto_update* attribute. If None, the value from the 'lists. auto_update' key in the *psyplot.rcParams* dictionary is used.

• **prefer_list** (*bool*) – If True and multiple variable names per array are found, the *InteractiveList* class is used. Otherwise the arrays are put together into one *InteractiveArray*.

• **default_slice** (*indexer*) – Index (e.g. 0 if *method* is *'isel'*) that shall be used for dimensions not covered by *dims* and *furtherdims*. If None, the whole slice will be used.
• **decoder** ([CFDecoder](#)) – The decoder that shall be used to decoder the *base* dataset

• **squeeze** *(bool, optional)* – Default True. If True, and the created arrays have an axes with length 1, it is removed from the dimension list (e.g. an array with shape (3, 4, 1, 5) will be squeezed to shape (3, 4, 5))

• **attrs** *(dict, optional)* – Meta attributes that shall be assigned to the selected data arrays (additional to those stored in the *base* dataset)

• **load** *(bool or dict)* – If True, load the data from the dataset using the `xarray.Dataset.load()` method. If *dict*, those will be given to the above mentioned load method

• **plotter** ([psyplot.plotter.Plotter](#)) – The plotter instance that is used to visualize the data in this list

• **make_plot** *(bool)* – If True, the plot is made

### Other Parameters

• **arr_names** *(string, list of strings or dictionary)* – Set the unique array names of the resulting arrays and (optionally) dimensions.

  – if string: same as list of strings (see below). Strings may include `{0}` which will be replaced by a counter.

  – list of strings: those will be used for the array names. The final number of dictionaries in the return depend in this case on the `*dims*` and `**furtherdims`

  – dictionary: Then nothing happens and an `OrderedDict` version of `arr_names` is returned.

• **sort** *(list of strings)* – This parameter defines how the dictionaries are ordered. It has no effect if `arr_names` is a dictionary (use an `OrderedDict` for that). It can be a list of dimension strings matching to the dimensions in `dims` for the variable.

• **dims** *(dict)* – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example `dims = {'name': 't2m', 'time': 0}` will result in one plot for the first time step, whereas `dims = {'name': 't2m', 'time': [0, 1]}` will result in two plots, one for the first (time == 0) and one for the second (time == 1) time step.

• `'**kwargs'` – Further keyword arguments may point to any of the dimensions of the data (see `dims`)

### Returns

The list with the specified `InteractiveArray` instances that hold a reference to the given `base`

**Return type** `ArrayList`

---

**Examples**

You can disable the automatic update via

```python
1.12. API Reference
```
>>> with data.no_auto_update:
...    data.update(time=1)
...    data.start_update()

To permanently disable the automatic update, simply set

```python
>>> data.no_auto_update = True
>>> data.update(time=1)
>>> data.no_auto_update = False  # reenable automatical update
```

**psy**

Return the list itself

**start_update** *(draw=None, queues=None)*

Conduct the formerly registered updates

This method conducts the updates that have been registered via the `update()` method. You can call this method if the `auto_update` attribute of this instance is True and the `auto_update` parameter in the `update()` method has been set to False

**Parameters**

- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the `psyplot.rcParams` dictionary
- **queues** *(list of Queue.Queue instances)* – The queues that are passed to the `psyplot.plotter.Plotter.start_update()` method to ensure a thread-safe update. It can be None if only one single plotter is updated at the same time. The number of jobs that are taken from the queue is determined by the `_njobs()` attribute. Note that there this parameter is automatically configured when updating from a `Project`.

**Returns** A boolean indicating whether a redrawing is necessary or not

**Return type** bool

**See also:**

`no_auto_update`, `update()`

to_dataframe()

to_interactive_list() Return a `InteractiveList` that contains this object

class **psyplot.data.Signal** *(name=None, cls_signal=False)*

Bases: object

Signal to connect functions to a specific event

This class behaves almost similar to PyQt’s `QtCore.pyqtBoundSignal`

**Methods**

```python
connect(func)
disconnect([func]) Disconnect a function call to the signal.
emit(*args, **kwargs)
```
emit(*args, **kwargs)

instance = None
owner = None
class psyplot.data.UGridDecoder (ds=None, x=None, y=None, z=None, t=None)
Bases: psyplot.data.CFDecoder

Decoder for UGrid data sets

Warning: Currently only triangles are supported.

Methods

can_decode(ds, var) Check whether the given variable can be decoded.
decode_coords(ds[, gridfile]) Reimplemented to set the mesh variables as coordinates
get_cell_node_coord(var[, coords, axis, nans]) Checks whether the bounds in the variable attribute are triangular
get_mesh(var[, coords]) Get the mesh variable for the given var
get_nodes(coord, coords) Get the variables containing the definition of the nodes
get_triangles(var[, coords, convert_radian, ...]) Get the of the given coordinate.
get_x(var[, coords]) Get the centers of the triangles in the x-dimension
get_y(var[, coords]) Get the centers of the triangles in the y-dimension
is_triangular(*args, **kwargs) Reimplemented to return always True.
is_unstructured(*args, **kwargs) Reimplemented to return always True.

classmethod can_decode (ds, var)
Check whether the given variable can be decoded.

Returns True if a mesh coordinate could be found via the get_mesh() method

Parameters

• ds (xarray.Dataset) – The dataset that contains the given var

• var (xarray.Variable or xarray.DataArray) – The array to decode

Returns True if the decoder can decode the given array var. Otherwise False

Return type bool

static decode_coords (ds, gridfile=None)
Reimplemented to set the mesh variables as coordinates

Parameters

• ds (xarray.Dataset) – The dataset to decode

• gridfile (str) – The path to a separate grid file or a xarray.Dataset instance which may store the coordinates used in ds

Returns ds with additional coordinates

Return type xarray.Dataset
get_cell_node_coord (var, coords=None, axis='x', nans=None)
Checks whether the bounds in the variable attribute are triangular

Parameters
- **var** (xarray.Variable or xarray.DataArray) – The variable to check
- **coords** (dict) – Coordinates to use. If None, the coordinates of the dataset in the ds attribute are used.
- **axis** ("x", "y") – The spatial axis to check
- **nans** (None, 'skip', 'only') – Determines whether values with nan shall be left (None), skipped ('skip') or shall be the only one returned ('only')

Returns the bounds coordinate (if existent)
Return type xarray.DataArray or None

get_mesh (var, coords=None)
Get the mesh variable for the given var

Parameters
- **var** (xarray.Variable) – The data source with the 'mesh' attribute
- **coords** (dict) – The coordinates to use. If None, the coordinates of the dataset of this decoder is used

Returns The mesh coordinate
Return type xarray.Coordinate

get_nodes (coord, coords)
Get the variables containing the definition of the nodes

Parameters
- **coord** (xarray.Coordinate) – The mesh variable
- **coords** (dict) – The coordinates to use to get node coordinates

get_triangles (var, coords=None, convert_radian=True, copy=False, src_crs=None, target_crs=None, nans=None, stacklevel=1)
Get the of the given coordinate.

Parameters
- **var** (xarray.Variable or xarray.DataArray) – The variable to use
- **coords** (dict) – Alternative coordinates to use. If None, the coordinates of the ds dataset are used
- **convert_radian** (bool) – If True and the coordinate has units in ‘radian’, those are converted to degrees
- **copy** (bool) – If True, vertice arrays are copied
- **src_crs** (cartopy.crs.Crs) – The source projection of the data. If not None, a transformation to the given target_crs will be done
- **target_crs** (cartopy.crs.Crs) – The target projection for which the triangles shall be transformed. Must only be provided if the src_crs is not None.
- **nans** (None, 'skip', 'only') – Determines whether values with nan shall be left (None), skipped ('skip') or shall be the only one returned ('only')

Returns The spatial triangles of the variable
Return type matplotlib.tri.Triangulation

Notes
If the 'location' attribute is set to 'node', a delaunay triangulation is performed using the matplotlib.tri.Triangulation class.

Todo: Implement the visualization for UGrid data shown on the edge of the triangles

get_x (var, coords=None)
Get the centers of the triangles in the x-dimension

Returns The y-coordinate or None if it could be found

Return type xarray.Coordinate or None

get_y (var, coords=None)
Get the centers of the triangles in the y-dimension

Returns The y-coordinate or None if it could be found

Return type xarray.Coordinate or None

is_triangular (*args, **kwargs)
Reimplemented to return always True. Any *args and **kwargs are ignored

is_unstructured (*args, **kwargs)
Reimplemented to return always True. Any *args and **kwargs are ignored

psyplot.data.decode_absolute_time (times)

psyplot.data.encode_absolute_time (times)

psyplot.data.get_filename_ds (ds, dump=True, paths=None, **kwargs)
Return the filename of the corresponding to a dataset

This method returns the path to the ds or saves the dataset if there exists no filename

Parameters

• ds (xarray.Dataset) – The dataset you want the path information for

• dump (bool) – If True and the dataset has not been dumped so far, it is dumped to a temporary file or the one generated by paths is used

• paths (iterable or True) – An iterator over filenames to use if a dataset has no filename. If paths is True, an iterator over temporary files will be created without raising a warning

Other Parameters

• "**kwargs" – Any other keyword for the to_netcdf () function

• path (str, Path or file-like object, optional) – Path to which to save this dataset. File-like objects are only supported by the scipy engine. If no path is provided, this function returns the resulting netCDF file as bytes; in this case, we need to use scipy, which does not support netCDF version 4 (the default format becomes NETCDF3_64BIT).

• mode (‘w’, ‘a’), optional) – Write (‘w’) or append (‘a’) mode. If mode=’w’, any existing file at this location will be overwritten. If mode=’a’, existing variables will be overwritten.
• **format** ([‘NETCDF4’, ‘NETCDF4_CLASSIC’, ‘NETCDF3_64BIT’,] –
  ‘NETCDF3_CLASSIC’), optional File format for the resulting netCDF file:
  
  – NETCDF4: Data is stored in an HDF5 file, using netCDF4 API features.
  
  – NETCDF4_CLASSIC: Data is stored in an HDF5 file, using only netCDF 3 compatible
  API features.
  
  – NETCDF3_64BIT: 64-bit offset version of the netCDF 3 file format, which fully supports
  2+ GB files, but is only compatible with clients linked against netCDF version 3.6.0 or
  later.
  
  – NETCDF3_CLASSIC: The classic netCDF 3 file format. It does not handle 2+ GB files
  very well.

  All formats are supported by the netCDF4-python library. scipy.io.netcdf only supports
  the last two formats.

  The default format is NETCDF4 if you are saving a file to disk and have the netCDF4-
  python library available. Otherwise, xarray falls back to using scipy to write netCDF files
  and defaults to the NETCDF3_64BIT format (scipy does not support netCDF4).

• **group** (**str**, optional) – Path to the netCDF4 group in the given file to open (only works for
  format=’NETCDF4’). The group(s) will be created if necessary.

• **engine** ([`‘netcdf4’`, ‘scipy’, ‘h5netcdf’], optional) – Engine to use when writing netCDF
  files. If not provided, the default engine is chosen based on available dependencies, with a
  preference for ‘netcdf4’ if writing to a file on disk.

• **encoding** (**dict**, optional) – Nested dictionary with variable names as keys and diction-
  naries of variable specific encodings as values, e.g., ```{'my_variable': {'dtype': 'int16',
  'scale_factor': 0.1,
  'zlib': True}, ...}```

  The h5netcdf engine supports both the NetCDF4-style compression encoding parameters
  ```{'zlib': True, 'complevel': 9}``` and the h5py ones ```{'compression': 'gzip', 'compression_opts': 9}```. This allows using any compression plugin
  installed in the HDF5 library, e.g. LZF.

**Returns**

• **str or None** – None, if the dataset has not yet been dumped to the harddisk and *dump* is
  False, otherwise the complete the path to the input file

• **str** – The module of the *xarray.backends.common.AbstractDataStore*
  instance that is used to hold the data

• **str** – The class name of the *xarray.backends.common.AbstractDataStore*
  instance that is used to open the data

```python
psyplot.data.get_fname_funcs = [<function _get_fname_netCDF4>], <function _get_fname_scipy>], functions to use to extract the file name from a data store

psyplot.data.get_index_from_coord(coord, base_index)
Function to return the coordinate as integer, integer array or slice

If *coord* is zero-dimensional, the corresponding integer in *base_index* will be supplied. Otherwise it is first tried
  to return a slice, if that does not work an integer array with the corresponding indices is returned.

**Parameters**

• **coord** (*xarray.Coordinate* or *xarray.Variable*) – Coordinate to convert
• base_index (pandas.Index) – The base index from which the coord was extracted

Returns The indexer that can be used to access the coord in the base_index

Return type int, array of ints or slice

psyplot.data.get_tdata(t_format, files)
Get the time information from file names

Parameters

• t_format (str) – The string that can be used to get the time information in the files. Any numeric datetime format string (e.g. %Y, %m, %H) can be used, but not non-numeric strings like %b, etc. See 1 for the datetime format strings

• files (list of str) – The that contain the time informations

Returns

• pandas.Index – The time coordinate

• list of str – The file names as they are sorten in the returned index

References

psyplot.data.open_dataset(filename_or_obj, decode_cf=True, decode_times=True, decode_coords=True, engine=None, gridfile=None, **kwargs)
Open an instance of xarray.Dataset.

This method has the same functionality as the xarray.open_dataset() method except that is supports an additional 'gdal' engine to open gdal Rasters (e.g. GeoTiffs) and that is supports absolute time units like 'day as %Y%m%d.%f' (if decode_cf and decode_times are True).

Parameters

• filename_or_obj (str, Path, file or xarray.backends.*DataStore) – Strings and Path objects are interpreted as a path to a netCDF file or an OpenDAP URL and opened with python-netCDF4, unless the filename ends with .gz, in which case the file is gunzipped and opened with scipy.io.netcdf (only netCDF3 supported). Byte-strings or file-like objects are opened by scipy.io.netcdf (netCDF3) or h5py (netCDF4/HDF).

• group (str, optional) – Path to the netCDF4 group in the given file to open (only works for netCDF4 files).

• decode_cf (bool, optional) – Whether to decode these variables, assuming they were saved according to CF conventions.

• mask_and_scale (bool, optional) – If True, replace array values equal to _FillValue with NA and scale values according to the formula original_values * scale_factor + add_offset, where _FillValue, scale_factor and add_offset are taken from variable attributes (if they exist). If the _FillValue or missing_value attribute contains multiple values a warning will be issued and all array values matching one of the multiple values will be replaced by NA. mask_and_scale defaults to True except for the pseudonetcdf backend.

• decode_times (bool, optional) – If True, decode times encoded in the standard NetCDF datatype format into datetime objects. Otherwise, leave them encoded as numbers.

• autoclose (bool, optional) – If True, automatically close files to avoid OS Error of too many files being open. However, this option doesn’t work with streams, e.g., BytesIO.

1 https://docs.python.org/2/library/datetime.html
• **concat_characters** *(bool, optional)* – If True, concatenate along the last dimension of character arrays to form string arrays. Dimensions will only be concatenated over (and removed) if they have no corresponding variable and if they are only used as the last dimension of character arrays.

• **decode_coords** *(bool, optional)* – If True, decode the ‘coordinates’ attribute to identify coordinates in the resulting dataset.

• **chunks** *(int or dict, optional)* – If chunks is provided, it is used to load the new dataset into dask arrays. chunks={} loads the dataset with dask using a single chunk for all arrays.

• **lock** *(False or duck threading.Lock, optional)* – Resource lock to use when reading data from disk. Only relevant when using dask or another form of parallelism. By default, appropriate locks are chosen to safely read and write files with the currently active dask scheduler.

• **cache** *(bool, optional)* – If True, cache data loaded from the underlying datastore in memory as NumPy arrays when accessed to avoid reading from the underlying data-store multiple times. Defaults to True unless you specify the chunks argument to use dask, in which case it defaults to False. Does not change the behavior of coordinates corresponding to dimensions, which always load their data from disk into a pandas.Index.

• **drop_variables** *(string or iterable, optional)* – A variable or list of variables to exclude from being parsed from the dataset. This may be useful to drop variables with problems or inconsistent values.

• **backend_kwargs** *(dictionary, optional)* – A dictionary of keyword arguments to pass on to the backend. This may be useful when backend options would improve performance or allow user control of dataset processing.

• **use_cftime** *(bool, optional)* – Only relevant if encoded dates come from a standard calendar (e.g. ‘gregorian’, ‘proleptic_gregorian’, ‘standard’, or not specified). If None (default), attempt to decode times to np.datetime64[ns] objects; if this is not possible, decode times to cftime.datetime objects. If True, always decode times to cftime. datetime objects, regardless of whether or not they can be represented using np.datetime64[ns] objects. If False, always decode times to np.datetime64[ns] objects; if this is not possible raise an error.

• **engine** *(‘netcdf4’, ’scipy’, ’pydap’, ’h5netcdf’, ’gdal’), optional)* – Engine to use when reading netCDF files. If not provided, the default engine is chosen based on available dependencies, with a preference for ‘netcdf4’.

• **gridfile** *(str)* – The path to a separate grid file or a xarray.Dataset instance which may store the coordinates used in ds

Returns The dataset that contains the variables from filename_or_obj

Return type xarray.Dataset

**psyplot.data.open_mfdataset** *(paths, decode_cf=True, decode_times=True, decode_coords=True, engine=None, gridfile=None, t_format=None, **kwargs)*

Open multiple files as a single dataset.

This function is essentially the same as the xarray.open_mfdataset() function but (as the open_dataset()) supports additional decoding and the 'gdal' engine. You can further specify the t_format parameter to get the time information from the files and use the results to concatenate the files

Parameters
• **paths** *(str or sequence)* – Either a string glob in the form "path/to/my/files/*.nc" or an explicit list of files to open. Paths can be given as strings or as pathlib Paths.

• **chunks** *(int or dict, optional)* – Dictionary with keys given by dimension names and values given by chunk sizes. In general, these should divide the dimensions of each dataset. If int, chunk each dimension by `chunks`. By default, chunks will be chosen to load entire input files into memory at once. This has a major impact on performance: please see the full documentation for more details [2].

• **concat_dim** *(None, str, DataArray or Index, optional)* – Dimension to concatenate files along. This argument is passed on to `xarray.auto_combine()` along with the dataset objects. You only need to provide this argument if the dimension along which you want to concatenate is not a dimension in the original datasets, e.g., if you want to stack a collection of 2D arrays along a third dimension. By default, xarray attempts to infer this argument by examining component files. Set `concat_dim=None` explicitly to disable concatenation.

• **compat** *({'identical', 'equals', 'broadcast_equals', 'no_conflicts'}, optional)* – String indicating how to compare variables of the same name for potential conflicts when merging:
  
  - `'broadcast_equals'`: all values must be equal when variables are broadcast against each other to ensure common dimensions.
  - `'equals'`: all values and dimensions must be the same.
  - `'identical'`: all values, dimensions and attributes must be the same.
  - `'no_conflicts'`: only values which are not null in both datasets must be equal. The returned dataset then contains the combination of all non-null values.

• **preprocess** *(callable, optional)* – If provided, call this function on each dataset prior to concatenation. You can find the file-name from which each dataset was loaded in `ds.encoding['source']`.

• **lock** *(False or duck threading.Lock, optional)* – Resource lock to use when reading data from disk. Only relevant when using dask or another form of parallelism. By default, appropriate locks are chosen to safely read and write files with the currently active dask scheduler.

• **data_vars** *({'minimal', 'different', 'all' or list of str}, optional)* – These data variables will be concatenated together:
  
  - `'minimal'`: Only data variables in which the dimension already appears are included.
  - `'different'`: Data variables which are not equal (ignoring attributes) across all datasets are also concatenated (as well as all for which dimension already appears). Beware: this option may load the data payload of data variables into memory if they are not already loaded.
  - `'all'`: All data variables will be concatenated.
  - list of str: The listed data variables will be concatenated, in addition to the ‘minimal’ data variables.

• **coords** *({'minimal', 'different', 'all' or list of str}, optional)* – These coordinate variables will be concatenated together:
  
  - `'minimal'`: Only coordinates in which the dimension already appears are included.
– ‘different’: Coordinates which are not equal (ignoring attributes) across all datasets are also concatenated (as well as all for which dimension already appears). Beware: this option may load the data payload of coordinate variables into memory if they are not already loaded.

– ‘all’: All coordinate variables will be concatenated, except those corresponding to other dimensions.

– list of str: The listed coordinate variables will be concatenated, in addition the ‘minimal’ coordinates.

• parallel (bool, optional) – If True, the open and preprocess steps of this function will be performed in parallel using dask.delayed. Default is False.

• **kwargs (optional) – Additional arguments passed on to xarray.open_dataset().

• engine (str) – Engine to use when reading netCDF files. If not provided, the default engine is chosen based on available dependencies, with a preference for ’netcdf4’.

• t_format (str) – The string that can be used to get the time information in the files. Any numeric datetime format string (e.g. %Y, %m, %H) can be used, but not non-numeric strings like %b, etc. See 1 for the datetime format strings

• gridfile (str) – The path to a separate grid file or a xarray.Dataset instance which may store the coordinates used in ds

Returns The dataset that contains the variables from filename_or_obj

Return type xarray.Dataset

psyplot.data.setup_coords(arr_names=None, sort=[], dims={}, **kwargs)
Sets up the arr_names dictionary for the plot

Parameters

• arr_names (string, list of strings or dictionary) – Set the unique array names of the resulting arrays and (optionally) dimensions.

  – if string: same as list of strings (see below). Strings may include {0} which will be replaced by a counter.

  – list of strings: those will be used for the array names. The final number of dictionaries in the return depend in this case on the dims and **furtherdims

  – dictionary: Then nothing happens and an OrderedDict version of arr_names is returned.

• sort (list of strings) – This parameter defines how the dictionaries are ordered. It has no effect if arr_names is a dictionary (use an OrderedDict for that). It can be a list of dimension strings matching to the dimensions in dims for the variable.

• dims (dict) – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example dims = {‘name’: ‘t2m’, ‘time’: 0} will result in one plot for the first time step, whereas dims = {‘name’: ‘t2m’, ‘time’: [0, 1]} will result in two plots, one for the first (time == 0) and one for the second (time == 1) time step.

• **kwargs – The same as dims (those will update what is specified in dims)
**Returns** A mapping from the keys in `arr_names` and to dictionaries. Each dictionary corresponds defines the coordinates of one data array to load.

**Return type** OrderedDict

```python
 psyplot.data.t_patterns = {'%H': '[0-9]{1,2}', '%M': '[0-9]{1,2}', '%S': '[0-9]{1,2}', '%Y': '[0-9]{4}', '%d': '[0-9]{1,2}', '%m': '[0-9]{1,2}'}
```

A mapping that translates datetime format strings to regex patterns.

```python
 psyplot.data.to_netcdf(ds, *args, **kwargs)
```

Store the given dataset as a netCDF file.

This function works essentially the same as the usual `xarray.Dataset.to_netcdf()` method but can also encode absolute time units.

**Parameters**

- `ds` *(xarray.Dataset)* – The dataset to store.
- `path` *(str, Path or file-like object, optional)* – Path to which to save this dataset. File-like objects are only supported by the scipy engine. If no path is provided, this function returns the resulting netCDF file as bytes; in this case, we need to use scipy, which does not support netCDF version 4 (the default format becomes NETCDF3_64BIT).
- `mode` *(str, optional)* – Write (`'w'`) or append (`'a'`) mode. If mode=`'w'`, any existing file at this location will be overwritten. If mode=`'a'`, existing variables will be overwritten.
- `format` *(str, optional)* – File format for the resulting netCDF file:
  - `NETCDF4`: Data is stored in an HDF5 file, using netCDF4 API features.
  - `NETCDF4_CLASSIC`: Data is stored in an HDF5 file, using only netCDF 3 compatible API features.
  - `NETCDF3_64BIT`: 64-bit offset version of the netCDF 3 file format, which fully supports 2+ GB files, but is only compatible with clients linked against netCDF version 3.6.0 or later.
  - `NETCDF3_CLASSIC`: The classic netCDF 3 file format. It does not handle 2+ GB files very well.

All formats are supported by the netCDF4-python library. scipy.io.netcdf only supports the last two formats. The default format is NETCDF4 if you are saving a file to disk and have the netCDF4-python library available. Otherwise, xarray falls back to using scipy to write netCDF files and defaults to the NETCDF3_64BIT format (scipy does not support netCDF4).

- `group` *(str, optional)* – Path to the netCDF4 group in the given file to open (only works for format='NETCDF4'). The group(s) will be created if necessary.
- `engine` *(str, optional)* – Engine to use when writing netCDF files. If not provided, the default engine is chosen based on available dependencies, with a preference for `netcdf4` if writing to a file on disk.
- `encoding` *(dict, optional)* – Nested dictionary with variable names as keys and dictionaries of variable specific encodings as values, e.g., `{'my_variable': {'dtype': 'int16', 'scale_factor': 0.1, 'zlib': True}, ...}`

The `h5netcdf` engine supports both the NetCDF4-style compression encoding parameters `{'zlib': True, 'complevel': 9}` and the h5py ones `{'compression':...`
'gzip', 'compression_opts': 9). This allows using any compression plugin installed in the HDF5 library, e.g. LZF.

```python
psyplot.data.to_slice(arr)
```

Test whether `arr` is an integer array that can be replaced by a slice

**Parameters**

- `arr` *(numpy.array)* – Numpy integer array

**Returns**

- If `arr` could be converted to an array, this is returned, otherwise `None` is returned

**Return type**

- slice or None

See also:

- `get_index_from_coord()`

**psyplot.docstring module**

Docstring module of the psyplot package

We use the docrep package for managing our docstrings

**Classes**

- `PsyplotDocstringProcessor(*args, **kwargs)`
  A `docrep.DocstringProcessor` subclass with possible types section

**Functions**

- `append_original_doc(parent[, num])`
  Return an iterator that append the docstring of the given parent

- `dedent(func)`
  Dedent the docstring of a function and substitute with params

- `indent(text[, num])`
  Indent the given string

**Data**

- `docstrings(func)`
  `docrep.PsyplotDocstringProcessor` instance that simplifies the reuse

**class**

- `psyplot.docstring.PsyplotDocstringProcessor(*args, **kwargs)`
  A `docrep.DocstringProcessor` subclass with possible types section

  **Parameters**

  - `and **kwargs` *(+args)* – Parameters that shall be used for the substitution. Note that you can only provide either `*args` or `**kwargs`, furthermore most of the methods like `get_sections` require `**kwargs` to be provided.

  **Methods**

  - `get_sections(s, base[, sections])`  
    Extract the specified sections out of the given string

**Attributes**
### psyplot Documentation, Release 1.2.1

#### param_like_sections

Built-in mutable sequence.

**get_sections(s, base, sections=[ 'Parameters', 'Other Parameters', 'Possible types'])**

Extract the specified sections out of the given string

The same as the docrep.DocstringProcessor.get_sections() method but uses the 'Possible types' section by default, too

**Parameters**

- `s (str)` – Docstring to split
- `base (str)` – base to use in the sections attribute
- `sections (list of str)` – sections to look for. Each section must be followed by a newline character (‘n’) and a bar of ‘-‘ (following the numpy (napoleon) docstring conventions).

**Returns** The replaced string

**Return type** `str`

#### psyplot.docstring.append_original_doc(parent, num=0)

Return an iterator that append the docstring of the given `parent` function to the applied function

#### psyplot.docstring.dedent(func)

Dedent the docstring of a function and substitute with params

**Parameters** `func (function)` – function with the documentation to dedent

#### psyplot.docstring.docstrings(func) = <psyplot.docstring.PsyplotDocstringProcessor object>

docrep.PsyplotDocstringProcessor instance that simplifies the reuse of docstrings from between different python objects.

#### psyplot.docstring.indent(text, num=4)

Indent the given string

### psyplot.gdal_store module

Gdal Store for reading GeoTIFF files into an `xarray.Dataset`

This module contains the definition of the `GdalStore` class that can be used to read in a GeoTIFF file into an `xarray.Dataset`. It requires that you have the python gdal module installed.

#### Examples

to open a GeoTIFF file named 'my_tiff.tiff' you can do:

```python
>>> from psyplot.gdal_store import GdalStore
>>> from xarray import open_dataset
>>> ds = open_dataset(GdalStore('my_tiff'))
```

Or you use the `engine` of the `psyplot.open_dataset()` function:

```python
>>> ds = open_dataset('my_tiff.tiff', engine='gdal')
```
Classes

GdalStore(filename_or_obj) Datastore to read raster files suitable for the gdal package

class psyplot.gdal_store.GdalStore(filename_or_obj)
    Bases: xarray.backends.common.AbstractDataStore
    Datastore to read raster files suitable for the gdal package
    
    We recommend to use the psyplot.open_dataset() function to open a geotiff file:

    ```python
    >>> ds = psyplot.open_dataset('my_geotiff.tiff', engine='gdal')
    ```

    Methods

    get_attrs()
    get_variables()

    Notes

    The GdalStore object is not as elaborate as, for example, the gdal_translate command. Many attributes, e.g. variable names or netCDF dimensions will not be interpreted. We only support two dimensional arrays and each band is saved into one variable named like 'Band1', 'Band2', .... If you want a more elaborate translation of your GDAL Raster, convert the file to a netCDF file using gdal_translate or the gdal.GetDriverByName('netCDF').CreateCopy method. However this class does not create an extra file on your hard disk as it is done by GDAL.

    Parameters filename_or_obj(str) – The path to the GeoTIFF file or a gdal dataset

    get_attrs()
    get_variables()

psyplot.plotter module

Core package for interactive visualization in the psyplot package

This package defines the Plotter and Formatoption classes, the core of the visualization in the psyplot package. Each Plotter combines a set of formatoption keys where each formatoption key is represented by a Formatoption subclass.

Classes

DictFormatoption(key[, plotter, ...]) Base formatoption class defining an alternative set_value that works for dictionaries.
Formatoption(key[, plotter, index_in_list, ...]) Abstract formatoption
FormatoptionMeta Meta class for formatoptions
Plotter([data, ax, auto_update, project, ...]) Interactive plotting object for one or more data arrays
PostProcDependencies The dependencies of this formatoption
PostProcessing(key[, plotter, ...]) Apply your own postprocessing script
PostTiming(key[, plotter, index_in_list, ...]) Determine when to run the post formatoption
Data

<table>
<thead>
<tr>
<th>BEFOREPLOTTING</th>
<th>Priority value of formatoptions that are updated before the plot it made.</th>
</tr>
</thead>
<tbody>
<tr>
<td>END</td>
<td>Priority value of formatoptions that are updated at the end.</td>
</tr>
<tr>
<td>START</td>
<td>Priority value of formatoptions that are updated before the data is loaded.</td>
</tr>
<tr>
<td>groups</td>
<td>dict. Mapping from group to group names</td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th><code>default_print_func</code></th>
<th>the default function to use when printing formatoption infos (the default is)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>format_time(x)</code></td>
<td>Formats date values</td>
</tr>
<tr>
<td><code>is_data_dependent(fmto, data)</code></td>
<td>Check whether a formatoption is data dependent</td>
</tr>
</tbody>
</table>

```python
psyplot.plotter.BEFOREPLOTTING = 20
```

Priority value of formatoptions that are updated before the plot it made.

```python
class psyplot.plotter.DictFormatoption(key, plotter=None, index_in_list=None, additional_children=[], additional_dependencies=[], **kwargs)
```

Base formatoption class defining an alternative set_value that works for dictionaries.

Parameters

- **key** (*str*) – formatoption key in the plotter
- **plotter** (*psyplot.plotter.Plotter*) – Plotter instance that holds this formatoption. If None, it is assumed that this instance serves as a descriptor.
- **index_in_list** (*int or None*) – The index that shall be used if the data is a psyplot.InteractiveList
- **additional_children** (*list or str*) – Additional children to use (see the children attribute)
- **additional_dependencies** (*list or str*) – Additional dependencies to use (see the dependencies attribute)
- ****kwargs – Further keywords may be used to specify different names for children, dependencies and connection formatoptions that match the setup of the plotter. Hence, keywords may be anything of the children, dependencies and connections attributes, with values being the name of the new formatoption in this plotter.

Methods

<table>
<thead>
<tr>
<th><code>set_value(value[, validate, todefault])</code></th>
<th>Set (and validate) the value in the plotter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_value(value, validate=True, todefault=False)</code></td>
<td>Set (and validate) the value in the plotter</td>
</tr>
</tbody>
</table>

Parameters

- **value** – Value to set
• **validate** *(bool)* – if True, validate the value before it is set

• **todefault** *(bool)* – True if the value is updated to the default value

**Notes**

• If the current value in the plotter is None, then it will be set with the given value, otherwise the current value in the plotter is updated

• If the value is an empty dictionary, the value in the plotter is cleared

```
 psyplot.plotter.END = 10
 Priority value of formatoptions that are updated at the end.
```

```
class psyplot.plotter.Formatoption(key, plotter=None, index_in_list=None, additional_children=[], additional_dependencies=[], **kwargs)
Bases: object
Abstract formatoption
This class serves as an abstract version of a formatoption descriptor that can be used by Plotter instances.

Parameters

• **key** *(str)* – formatoption key in the plotter

• **plotter** *(psyplot.plotter.Plotter)* – Plotter instance that holds this formatoption. If None, it is assumed that this instance serves as a descriptor.

• **index_in_list** *(int or None)* – The index that shall be used if the data is a psyplot.InteractiveList

• **additional_children** *(list or str)* – Additional children to use (see the children attribute)

• **additional_dependencies** *(list or str)* – Additional dependencies to use (see the dependencies attribute)

• ****kwargs** – Further keywords may be used to specify different names for children, dependencies and connection formatoptions that match the setup of the plotter. Hence, keywords may be anything of the children, dependencies and connections attributes, with values being the name of the new formatoption in this plotter.

**Interface to the data**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any_decoder</td>
<td>Return the first possible decoder</td>
</tr>
<tr>
<td>ax</td>
<td>The axes this Formatoption plots on</td>
</tr>
<tr>
<td>data</td>
<td>The data that is plotted</td>
</tr>
<tr>
<td>data_dependent</td>
<td>bool or a callable. This attribute indicates whether this</td>
</tr>
<tr>
<td>decoder</td>
<td>The CFDecoder instance that decodes the</td>
</tr>
<tr>
<td>index_in_list</td>
<td>int or None. Index that is used in case the plotting data is a</td>
</tr>
<tr>
<td>iter_data</td>
<td>Returns an iterator over the plot data arrays</td>
</tr>
<tr>
<td>iter_raw_data</td>
<td>Returns an iterator over the original data arrays</td>
</tr>
<tr>
<td>project</td>
<td>Project of the plotter of this instance</td>
</tr>
<tr>
<td>raw_data</td>
<td>The original data of the plotter of this formatoption</td>
</tr>
</tbody>
</table>

Continued on next page
Table 53 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_data</code> (data[, i])</td>
<td>Replace the data to plot</td>
</tr>
<tr>
<td><code>set_decoder</code> (decoder[, i])</td>
<td>Replace the data to plot</td>
</tr>
</tbody>
</table>

**Interface for the plotter**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>changed</code></td>
<td>bool indicating whether the value changed compared to the</td>
</tr>
<tr>
<td><code>check_and_set</code> (value[, todefault, validate])</td>
<td>Checks the value and sets the value if it changed</td>
</tr>
<tr>
<td><code>diff</code> (value)</td>
<td>Checks whether the given value differs from what is currently set</td>
</tr>
<tr>
<td><code>finish_update</code>()</td>
<td>Finish the update, initialization and sharing process</td>
</tr>
<tr>
<td><code>initialize_plot</code> (value, *args, **kwargs)</td>
<td>Method that is called when the plot is made the first time</td>
</tr>
<tr>
<td><code>key</code></td>
<td>str. Formatoption key of this class in the</td>
</tr>
<tr>
<td><code>lock</code></td>
<td>A threading.Rlock instance to lock while updating</td>
</tr>
<tr>
<td><code>plot_fmt</code></td>
<td>bool. Has to be True if the formatoption has a make_plot</td>
</tr>
<tr>
<td><code>plotter</code></td>
<td>Plotter. Plotter instance this</td>
</tr>
<tr>
<td><code>priority</code></td>
<td>int. Priority value of the formatoption determining when</td>
</tr>
<tr>
<td><code>remove</code>()</td>
<td>Method to remove the effects of this formatoption</td>
</tr>
<tr>
<td><code>requires_clearing</code></td>
<td>bool. True if an update of this formatoption requires a</td>
</tr>
<tr>
<td><code>requires_replot</code></td>
<td>Boolean that is True if an update of the formatoption requires a replot</td>
</tr>
<tr>
<td><code>set_value</code> (value[, validate, todefault])</td>
<td>Set (and validate) the value in the plotter.</td>
</tr>
<tr>
<td><code>share</code> (fmto[, initializing])</td>
<td>Share the settings of this formatoption with other data objects</td>
</tr>
<tr>
<td><code>update</code> (value)</td>
<td>Method that is call to update the formatoption on the axes</td>
</tr>
<tr>
<td><code>update_after_plot</code></td>
<td>bool. True if this formatoption needs an update after the plot</td>
</tr>
</tbody>
</table>

**Interface to other formatoptions**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>children</code></td>
<td>list of str. List of formatoptions that have to be updated before this</td>
</tr>
<tr>
<td><code>connections</code></td>
<td>list of str. Connections to other formatoptions that are (different</td>
</tr>
<tr>
<td><code>dependencies</code></td>
<td>list of str. List of formatoptions that force an update of this</td>
</tr>
<tr>
<td><code>parents</code></td>
<td>list of str. List of formatoptions that, if included in the update,</td>
</tr>
<tr>
<td><code>shared</code></td>
<td>set of the Formatoption instance that are shared</td>
</tr>
<tr>
<td><code>shared_by</code></td>
<td>None if the formatoption is not controlled by another formatoption</td>
</tr>
</tbody>
</table>

**Formatoption intrinsic**

---

**1.12. API Reference 131**
<table>
<thead>
<tr>
<th>Formatoption</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Default value of this formatoption</td>
</tr>
<tr>
<td>validate</td>
<td>Validation method of the formatoption</td>
</tr>
<tr>
<td>value</td>
<td>Value of the formatoption in the corresponding plotter or value2pickle</td>
</tr>
<tr>
<td>value2pickle</td>
<td>The value that can be used when pickling the information of the project</td>
</tr>
<tr>
<td>value2share</td>
<td>The value that is passed to shared formatoptions (by default, the</td>
</tr>
</tbody>
</table>

**Information attributes**

<table>
<thead>
<tr>
<th>Formatoption</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_key</td>
<td>The key of this formatoption in the psyplot. groupparams</td>
</tr>
<tr>
<td>group</td>
<td>str. Key of the group name in groups of this</td>
</tr>
<tr>
<td>groupname</td>
<td>Long name of the group this formatoption belongs too.</td>
</tr>
<tr>
<td>name</td>
<td>str. A bit more verbose name than the formatoption key to be</td>
</tr>
</tbody>
</table>

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_fmt_widget</td>
<td>Get a widget to update the formatoption in the GUI</td>
</tr>
</tbody>
</table>

**Miscellaneous**

<table>
<thead>
<tr>
<th>Formatoption</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_kwargs</td>
<td>dict key word arguments that are passed to the logger</td>
</tr>
<tr>
<td>logger</td>
<td>Logger of the plotter</td>
</tr>
</tbody>
</table>

**any_decoder**

Return the first possible decoder

**ax**

The axes this Formatoption plots on

**changed**

bool indicating whether the value changed compared to the default or not.

**check_and_set** (value, todefault=False, validate=True)

Checks the value and sets the value if it changed

This method checks the value and sets it only if the `diff()` method result of the given `value` is True

**Parameters**

- **value** – A possible value to set
- **todefault** (bool) – True if the value is updated to the default value

**Returns** A boolean to indicate whether it has been set or not

**Return type** bool

**children** = []

list of str. List of formatoptions that have to be updated before this one is updated. Those formatoptions are only updated if they exist in the update parameters.
connections = []
    list of str. Connections to other formatoptions that are (different from dependencies and children) not important for the update process

data
    The data that is plotted

data_dependent = False
    bool or a callable. This attribute indicates whether this Formatoption depends on the data and should be updated if the data changes. If it is a callable, it must accept one argument: the new data. (Note: This is automatically set to True for plot formatoptions)

decoder
    The CFDecoder instance that decodes the raw_data

default
    Default value of this formatoption

default_key
    The key of this formatoption in the psyplot.rcParams

dependencies = []
    list of str. List of formatoptions that force an update of this formatoption if they are updated.

diff(value)
    Checks whether the given value differs from what is currently set

    Parameters
    value -- A possible value to set (make sure that it has been validate via the validate attribute before)

    Returns
    True if the value differs from what is currently set

    Return type
    bool

finish_update()
    Finish the update, initialization and sharing process

    This function is called at the end of the Plotter.start_update(), Plotter.initialize_plot() or the Plotter.share() methods.

get_fmt_widget(parent, project)
    Get a widget to update the formatoption in the GUI

    This method should return a QWidget that is loaded by the psyplot-gui when the formatoption is selected in the psyplot_gui.main.Mainwindow.fmt_widget. It should call the insert_text() method when the update text for the formatoption should be changed.

    Parameters

    • parent (psyplot_gui.fmt_widget.FormatoptionWidget) -- The parent widget that contains the returned QWidge

    • project (psyplot.project.Project) -- The current subproject (see psyplot.project.gcp())

    Returns
    The widget to control the formatoption

    Return type
    PyQt5.QtWidgets.QWidget

group = 'misc'
    str. Key of the group name in groups of this formatoption keyword

groupname
    Long name of the group this formatoption belongs too.
index_in_list = 0
int or None. Index that is used in case the plotting data is a psyplot.InteractiveList

init_kwargs
dict key word arguments that are passed to the initialization of a new instance when accessed from the
descriptor

initialize_plot(value, *args, **kwargs)
Method that is called when the plot is made the first time

Parameters value – The value to use for the initialization

iter_data
Returns an iterator over the plot data arrays

iter_raw_data
Returns an iterator over the original data arrays

key = None
str. Formatoption key of this class in the Plotter class

lock
A threading.RLock instance to lock while updating

This lock is used when multiple plotter instances are updated at the same time while sharing formatoptions.

logger
Logger of the plotter

name = None
str. A bit more verbose name than the formatoption key to be included in the gui. If None, the key is
used in the gui

parents = []
list of str. List of formatoptions that, if included in the update, prevent the update of this formatoption.

plot_fmt = False
bool. Has to be True if the formatoption has a make_plot method to make the plot.

plotter
Plotter. Plotter instance this formatoption belongs to

priority = 10
int. Priority value of the the formatoption determining when the formatoption is updated.

• 10: at the end (for labels, etc.)
• 20: before the plotting (e.g. for colormaps, etc.)
• 30: before loading the data (e.g. for lonlatbox)

project
Project of the plotter of this instance

raw_data
The original data of the plotter of this formatoption

remove()
Method to remove the effects of this formatoption

This method is called when the axes is cleared due to a formatoption with requires_clearing set to
True. You don’t necessarily have to implement this formatoption if your plot results are removed by the
usual matplotlib.axes.Axes.clear() method.
requires_clearing = False

   bool. True if an update of this formatoption requires a clearing of the axes and reinitialization of the plot

requires_replot = False

   Boolean that is True if an update of the formatoption requires a replot

set_data (data, i=None)

   Replace the data to plot

   This method may be used to replace the data that is visualized by the plotter. It changes its behaviour depending on whether an psyplot.data.InteractiveList is visualized or a single pysplot.data.InteractiveArray

   Parameters

   • data (psyplot.data.InteractiveBase) – The data to insert
     • i (int) – The position in the InteractiveList where to insert the data (if the plotter visualizes a list anyway)

Notes

   This method uses the Formatoption.data attribute

set_decoder (decoder, i=None)

   Replace the data to plot

   This method may be used to replace the data that is visualized by the plotter. It changes its behaviour depending on whether an psyplot.data.InteractiveList is visualized or a single pysplot.data.InteractiveArray

   Parameters

   • decoder (psyplot.data.CFDecoder) – The decoder to insert
     • i (int) – The position in the InteractiveList where to insert the data (if the plotter visualizes a list anyway)

set_value (value, validate=True, todefault=False)

   Set (and validate) the value in the plotter. This method is called by the plotter when it attempts to change the value of the formatoption.

   Parameters

   • value – Value to set
     • validate (bool) – if True, validate the value before it is set
     • todefault (bool) – True if the value is updated to the default value

share (fmto, initializing=False, **kwargs)

   Share the settings of this formatoption with other data objects

   Parameters

   • fmto (Formatoption) – The Formatoption instance to share the attributes with
     • **kwargs – Any other keyword argument that shall be passed to the update method of fmto

shared = {}

   set of the Formatoption instance that are shared with this instance.
shared_by
None if the formatoption is not controlled by another formatoption of another plotter, otherwise the corresponding Formatoption instance

update (value)
Method that is call to update the formatoption on the axes

Parameters
value – Value to update

update_after_plot = False
bool. True if this formatoption needs an update after the plot has changed

validate
Validation method of the formatoption

value
Value of the formatoption in the corresponding plotter or the shared value

value2pickle
The value that can be used when pickling the information of the project

value2share
The value that is passed to shared formatoptions (by default, the value attribute)

class psyplot.plotter.FormatoptionMeta
Bases: abc.ABCMeta
Meta class for formatoptions

This class serves as a meta class for formatoptions and allows a more efficient docstring generation by using the psyplot.docstring.docstrings when creating a new formatoption class

Assign an automatic documentation to the formatoption

class psyplot.plotter.Plotter (data=None, ax=None, auto_update=None, project=None, draw=False, make_plot=True, clear=False, enable_post=False, **kwargs)
Bases: dict
Interactive plotting object for one or more data arrays

This class is the base for the interactive plotting with the psyplot module. It capabilities are determined by it’s descriptor classes that are derived from the Formatoption class

Parameters

• data (InteractiveArray or ArrayList, optional) – Data object that shall be visualized. If given and plot is True, the initialize_plot() method is called at the end. Otherwise you can call this method later by yourself

• ax (matplotlib.axes.Axes) – Matplotlib Axes to plot on. If None, a new one will be created as soon as the initialize_plot() method is called

• auto_update (bool) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the update() method or not. See also the no_auto_update attribute. If None, the value from the 'lists.auto_update' key in the psyplot.rcParams dictionary is used.

• draw (bool or None) – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the 'auto_draw' parameter in the psyplot.rcParams dictionary

• make_plot (bool) – If True, and data is not None, the plot is initialized. Otherwise only the framework between plotter and data is set up
• **clear** *(bool)* – If True, the axes is cleared first

• **enable_post** *(bool)* – If True, the post formatoption is enabled and post processing scripts are allowed

• **kwargs** – Any formatoption key from the formatoptions attribute that shall be used

Attributes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ax</td>
<td>Axes instance of the plot</td>
</tr>
<tr>
<td>base_variables</td>
<td>A mapping from the base_variable names to the variables</td>
</tr>
<tr>
<td>changed</td>
<td>dict containing the key value pairs that are not the data</td>
</tr>
<tr>
<td>data</td>
<td>The psyplot.InteractiveBase instance of this plot</td>
</tr>
<tr>
<td>enable_post</td>
<td>bool that has to be True if the post processing script in figs2draw</td>
</tr>
<tr>
<td>figs2draw</td>
<td>All figures that have been manipulated through sharing and the own figure.</td>
</tr>
<tr>
<td>fmt_groups</td>
<td>A mapping from the formatoption group to the formatoptions</td>
</tr>
<tr>
<td>groups</td>
<td>A mapping from the group short name to the group description</td>
</tr>
<tr>
<td>include_links</td>
<td>Temporarily include links in the key descriptions from show_keys(), show_docs() and show_summaries().</td>
</tr>
<tr>
<td>iter_base_variables</td>
<td>A mapping from the base_variable names to the variables</td>
</tr>
<tr>
<td>logger</td>
<td>logging.Logger of this plot</td>
</tr>
<tr>
<td>no_auto_update</td>
<td>bool. Boolean controlling whether the start_update()</td>
</tr>
<tr>
<td>no_validation</td>
<td>Temporarily disable the validation</td>
</tr>
<tr>
<td>plot_data</td>
<td>The data that is used for plotting</td>
</tr>
<tr>
<td>plot_data_decoder</td>
<td>The decoder to use for the formatoptions.</td>
</tr>
<tr>
<td>post</td>
<td>Apply your own postprocessing script</td>
</tr>
<tr>
<td>post_timing</td>
<td>Determine when to run the post formatoption</td>
</tr>
<tr>
<td>project</td>
<td>psyplot.project.Project instance this plot belongs to</td>
</tr>
<tr>
<td>rc</td>
<td>Default values for this plot</td>
</tr>
</tbody>
</table>

Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check_data(name, dims, is_unstructured)</td>
<td>A validation method for the data shape</td>
</tr>
<tr>
<td>check_key(key[, raise_error])</td>
<td>Checks whether the key is a valid formatoption</td>
</tr>
<tr>
<td>draw()</td>
<td>Draw the figures and those that are shared and have been changed</td>
</tr>
<tr>
<td>get_enhanced_attrs(arr[, axes])</td>
<td>Return the validation function for a specified formatoption</td>
</tr>
<tr>
<td>get_vfunc(key)</td>
<td></td>
</tr>
<tr>
<td>has_changed(key[, include_last])</td>
<td>Determine whether a formatoption changed in the last update</td>
</tr>
<tr>
<td>initialize_plot([data, ax, make_plot, ...])</td>
<td>Initialize the plot for a data array</td>
</tr>
<tr>
<td>make_plot()</td>
<td>Method for making the plot</td>
</tr>
</tbody>
</table>
Table 61 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reinit([draw, clear])</code></td>
<td>Reinitializes the plot with the same data and on the same axes.</td>
</tr>
<tr>
<td><code>share(plotters[, keys, draw, auto_update])</code></td>
<td>Share the formatoptions of this plotter with others</td>
</tr>
<tr>
<td><code>show()</code></td>
<td>Shows all open figures</td>
</tr>
<tr>
<td><code>show_docs([keys, indent])</code></td>
<td>Classmethod to print the full documentations of the formatoptions</td>
</tr>
<tr>
<td><code>show_keys([keys, indent, grouped, func, ...])</code></td>
<td>Classmethod to return a nice looking table with the given formatoptions</td>
</tr>
<tr>
<td><code>show_summaries([keys, indent])</code></td>
<td>Classmethod to print the summaries of the formatoptions</td>
</tr>
<tr>
<td><code>start_update([draw, indent, queues, update_shared])</code></td>
<td>Conduct the registered plot updates</td>
</tr>
<tr>
<td><code>unshare(plotters[, keys, auto_update, draw])</code></td>
<td>Close the sharing connection of this plotter with others</td>
</tr>
<tr>
<td><code>unshare_me([keys, auto_update, draw, ...])</code></td>
<td>Close the sharing connection of this plotter with others</td>
</tr>
<tr>
<td><code>update([fmt, replot, auto_update, draw, ...])</code></td>
<td>Update the formatoptions and the plot</td>
</tr>
</tbody>
</table>

`ax`  
Axes instance of the plot

`base_variables`  
A mapping from the base_variable names to the variables

`changed`

`dict` containing the key value pairs that are not the default

`classmethod check_data(name, dims, is_unstructured)`  
A validation method for the data shape

The default method does nothing and should be subclassed to validate the results. If the plotter accepts an `InteractiveList`, it should accept a list for name and dims

**Parameters**

- `name (str or list of str)` – The variable name(s) of the data
- `dims (list of str or list of lists of str)` – The dimension name(s) of the data
- `is_unstructured (bool or list of bool)` – True if the corresponding array is unstructured

**Returns**

- `list of bool or None` – True, if everything is okay, False in case of a serious error, None if it is intermediate. Each object in this list corresponds to one in the given `name`
- `list of str` – The message giving more information on the reason. Each object in this list corresponds to one in the given `name`

`check_key(key, raise_error=True, *args, **kwargs)`  
Checks whether the key is a valid formatoption

**Parameters**

- `key (str)` – Key to check
- `raise_error (bool)` – If not True, a list of similar keys is returned
- `msg (str)` – The additional message that shall be used if no close match to key is found
• \*args, \*\*kwargs – They are passed to the `difflib.get_close_matches()` function (i.e. \( n \) to increase the number of returned similar keys and \( \text{cutoff} \) to change the sensibility)

Returns

• \textit{str} – The key if it is a valid string, else an empty string
• \textit{list} – A list of similar formatoption strings (if found)
• \textit{str} – An error message which includes

Raises \textbf{KeyError} – If the key is not a valid formatoption and \textit{raise\_error} is True

data

The \texttt{psyplot.InteractiveBase} instance of this plotter

draw()

Draw the figures and those that are shared and have been changed

\texttt{enable\_post = False}

\texttt{bool} that has to be \texttt{True} if the post processing script in the \texttt{post} formatoption should be enabled

\texttt{figs2draw}

All figures that have been manipulated through sharing and the own figure.

**Notes**

Using this property set will reset the figures too draw

\texttt{fmt\_groups}

A mapping from the formatoption group to the formatoptions

\texttt{get\_enhanced\_attrs}(arr, axes=['x', 'y', 't', 'z'])

\texttt{get\_vfnc}(key)

Return the validation function for a specified formatoption

**Parameters**

\texttt{key (str)} – Formatoption key in the \texttt{rc} dictionary

**Returns**

Validation function for this formatoption

**Return type**

\texttt{function}

\texttt{groups}

A mapping from the group short name to the group description

\texttt{has\_changed}(key, \texttt{include\_last=True})

Determine whether a formatoption changed in the last update

**Parameters**

• \texttt{key (str)} – A formatoption key contained in this plotter

• \texttt{include\_last (bool)} – if \texttt{True} and the formatoption has been included in the last update, the return value will not be None. Otherwise the return value will only be not None if it changed during the last update

**Returns**

• None, if the value has not been changed during the last update or \textit{key} is not a valid formatoption key

• a list of length two with the old value in the first place and the given \textit{value} at the second
Return type  None or list

include_links (value=None)
Temporarily include links in the key descriptions from show_keys(), show_docs() and show_summaries(). Note that this is a class attribute, so each change to the value of this attribute will affect all instances and subclasses

initialize_plot (data=None, ax=None, make_plot=True, clear=False, draw=False, remove=False, priority=None)
Initialize the plot for a data array

Parameters

• data (InteractiveArray or ArrayList, optional) – Data object that shall be visualized.
  – If not None and plot is True, the given data is visualized.
  – If None and the data attribute is not None, the data in the data attribute is visualized
  – If both are None, nothing is done.
• ax (matplotlib.axes.Axes) – Matplotlib Axes to plot on. If None, a new one will be created as soon as the initialize_plot() method is called
• make_plot (bool) – If True, and data is not None, the plot is initialized. Otherwise only the framework between plotter and data is set up
• clear (bool) – If True, the axes is cleared first
• draw (bool or None) – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the psyplot.rcParams dictionary
• remove (bool) – If True, old effects by the formatoptions in this plotter are undone first
• priority (int) – If given, initialize only the formatoption with the given priority. This value must be out of START, BEFOREPLOTTING or END

iter_base_variables
A mapping from the base_variable names to the variables

logger
logging.Logger of this plotter

make_plot ()
Method for making the plot

This method is called at the end of the BEFOREPLOTTING stage if and only if the plot_fmt attribute is set to True

no_auto_update
bool. Boolean controlling whether the start_update() method is automatically called by the update() method

Examples
You can disable the automatic update via

```
>>> with data.no_auto_update:
...    data.update(time=1)
...    data.start_update()
```

To permanently disable the automatic update, simply set
no_validation
Temporarily disable the validation

Examples
Although it is not recommended to set a value with disabled validation, you can disable it via:

```python
>>> with plotter.no_validation:
...     plotter['ticksize'] = 'x'
```

To permanently disable the validation, simply set

```python
>>> plotter.no_validation = True
>>> plotter['ticksize'] = 'x'
>>> plotter.no_validation = False  # reenable validation
```

plot_data
The data that is used for plotting

plot_data_decoder = None
The decoder to use for the formatoptions. If None, the decoder of the raw data is used

post
Apply your own postprocessing script
This formatoption let’s you apply your own post processing script. Just enter the script as a string and it will be executed. The formatoption will be made available via the self variable

Possible types

• None – Don’t do anything
• str – The post processing script as string

Note: This formatoption uses the built-in exec() function to compile the script. Since this poses a security risk when loading psyplot projects, it is by default disabled through the Plotter.enable_post attribute. If you are sure that you can trust the script in this formatoption, set this attribute of the corresponding Plotter to True

Examples
Assume, you want to manually add the mean of the data to the title of the matplotlib axes. You can simply do this via

```python
from psyplot.plotter import Plotter
from xarray import DataArray
plotter = Plotter(DataArray([1, 2, 3]))
```
# enable the post format option
plotter.enable_post = True
plotter.update(post="self.ax.set_title(str(self.data.mean()))")
plotter.ax.get_title()
'2.0'

By default, the post format option is only ran, when it is explicitly updated. However, you can use the post_timing format option, to run it automatically. E.g. for running it after every update of the plotter, you can set

```python
plotter.update(post_timing='always')
```

See also:

**post_timing** Determine the timing of this format option

**post_timing**
Determine when to run the post format option

This format option determines, whether the post format option should be run never, after replot or after every update.

**Possible types**

- ‘never’ – Never run post processing scripts
- ‘always’ – Always run post processing scripts
- ‘replot’ – Only run post processing scripts when the data changes or a replot is necessary

See also:

**post** The post processing format option

**project**
`psyplot.project.Project` instance this plotter belongs to

**rc**
Default values for this plotter

This SubDict stores the default values for this plotter. A modification of the dictionary does not affect other plotter instances unless you set the trace attribute to True

**reinit** *(draw=None, clear=False)*
Reinitializes the plot with the same data and on the same axes.

**Parameters**

- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the psyplot.rcParams dictionary
- **clear** *(bool)* – Whether to clear the axes or not
**Warning:** The axes may be cleared when calling this method (even if `clear` is set to False)!

**share** *(plotters, keys=None, draw=None, auto_update=False)*

Share the formatoptions of this plotter with others

This method shares the formatoptions of this `Plotter` instance with others to make sure that, if the formatoption of this changes, those of the others change as well

**Parameters**

- **plotters** *(list of `Plotter` instances or a `Plotter`)* – The plotters to share the formatoptions with
- **keys** *(string or iterable of strings)* – The formatoptions to share, or group names of formatoptions to share all formatoptions of that group (see the `fmt_groups` property). If None, all formatoptions of this plotter are unshared.
- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the `‘auto_draw’` parameter in the `psyplot.rcParams` dictionary
- **auto_update** *(bool)* – Boolean determining whether or not the `start_update()` method is called at the end. This parameter has no effect if the `no_auto_update` attribute is set to True.

**See also:**

`unshare()`, `unshare_me()`

**show()**

Shows all open figures

**classmethod show_docs** *(keys=None, indent=0, *args, **kwargs)*

Classmethod to print the full documentations of the formatoptions

**Parameters**

- **keys** *(list of str or None)* – If None, the all formatoptions of the given class are used. Group names from the `psyplot.plotter.groups` mapping are replaced by the formatoptions
- **indent** *(int)* – The indentation of the table
- **grouped** *(bool, optional)* – If True, the formatoptions are grouped corresponding to the `Formatoption.groupname` attribute

**Other Parameters**

- **func** *(function or None)* – The function the is used for returning (by default it is printed via the `print()` function or (when using the gui) in the help explorer). The given function must take a string as argument
- **include_links** *(bool or None, optional)* – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the `psyplot.plotter.Plotter.include_links` attribute.
- **‘*args,**kwargs’** – They are passed to the `difflib.get_close_matches()` function (i.e. `n` to increase the number of returned similar keys and `cutoff` to change the sensibility)

**Returns** None if `func` is the print function, otherwise anything else

**Return type** results of `func`
See also:

```
show_keys(), show_docs()
```

**classmethod show_keys**

```python
def show_keys(keys=None, indent=0, grouped=False, func=None, include_links=False, *args, **kwargs)
```

Classmethod to return a nice looking table with the given formatoptions

**Parameters**

- **keys** *(list of str or None)* – If None, the all formatoptions of the given class are used. Group names from the `psyplot.plotter.groups` mapping are replaced by the formatoptions
- **indent** *(int)* – The indentation of the table
- **grouped** *(bool, optional)* – If True, the formatoptions are grouped corresponding to the `Formatoption.groupname` attribute

**Other Parameters**

- **func** *(function or None)* – The function the is used for returning (by default it is printed via the `print()` function or (when using the gui) in the help explorer). The given function must take a string as argument
- **include_links** *(bool or None, optional)* – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the `psyplot.plotter.Plotter.include_links` attribute.
- **‘*args,**kwargs‘* – They are passed to the `difflib.get_close_matches()` function (i.e. `n` to increase the number of returned similar keys and `cutoff` to change the sensibility)

**Returns** None if `func` is the print function, otherwise anything else

**Return type** results of `func`

See also:

```
show_summaries(), show_docs()
```

**classmethod show_summaries**

```python
def show_summaries(keys=None, indent=0, *args, **kwargs)
```

Classmethod to print the summaries of the formatoptions

**Parameters**

- **keys** *(list of str or None)* – If None, the all formatoptions of the given class are used. Group names from the `psyplot.plotter.groups` mapping are replaced by the formatoptions
- **indent** *(int)* – The indentation of the table
- **grouped** *(bool, optional)* – If True, the formatoptions are grouped corresponding to the `Formatoption.groupname` attribute

**Other Parameters**

- **func** *(function or None)* – The function the is used for returning (by default it is printed via the `print()` function or (when using the gui) in the help explorer). The given function must take a string as argument
- **include_links** *(bool or None, optional)* – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the `psyplot.plotter.Plotter.include_links` attribute.
• ‘*args,**kwargs’ – They are passed to the `difflib.get_close_matches()` function (i.e. `n` to increase the number of returned similar keys and `cutoff` to change the sensibility)

  **Returns** None if `func` is the print function, otherwise anything else

  **Return type** results of `func`

  **See also:**
  - `show_keys()`, `show_docs()`

  **start_update** `(draw=None, queues=None, update_shared=True)`

  Conduct the registered plot updates

  This method starts the updates from what has been registered by the `update()` method. You can call this method if you did not set the `auto_update` parameter to True when calling the `update()` method and when the `no_auto_update` attribute is True.

  **Parameters**

  - **draw** (`bool or None`) – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the `psyplot.rcParams` dictionary

  - **queues** (list of `Queue.Queue` instances) – The queues that are passed to the `psyplot.plotter.Plotter.start_update()` method to ensure a thread-safe update. It can be None if only one single plotter is updated at the same time. The number of jobs that are taken from the queue is determined by the `_njobs()` attribute. Note that there this parameter is automatically configured when updating from a `Project`.

  **Returns** A boolean indicating whether a redrawing is necessary or not

  **Return type** bool

  **See also:**
  - `no_auto_update`, `update()`

  **unshare** `(plotters, keys=None, auto_update=False, draw=None)`

  Close the sharing connection of this plotter with others

  This method undoes the sharing connections made by the `share()` method and releases the given `plotters` again, such that the formatoptions in this plotter may be updated again to values different from this one.

  **Parameters**

  - **plotters** (list of `Plotter` instances or a `Plotter`) – The plotters to release

  - **keys** (`string or iterable of strings`) – The formatoptions to unshare, or group names of formatoptions to unshare all formatoptions of that group (see the `fmt_groups` property). If None, all formatoptions of this plotter are unshared.

  - **draw** (`bool or None`) – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the `psyplot.rcParams` dictionary

  - **auto_update** (`bool`) – Boolean determining whether or not the `start_update()` method is called at the end. This parameter has no effect if the `no_auto_update` attribute is set to True.

  **See also:**
  - `share()`, `unshare_me()`
unshare_me (keys=None, auto_update=False, draw=None, update_other=True)

Close the sharing connection of this plotter with others.

This method undoes the sharing connections made by the share() method and release this plotter again.

Parameters

- **keys** *(string or iterable of strings)* – The formatoptions to unshare, or group names of formatoptions to unshare all formatoptions of that group (see the fmt_groups property). If None, all formatoptions of this plotter are unshared.
- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the psyplot.rcParams dictionary
- **auto_update** *(bool)* – Boolean determining whether or not the start_update() method is called at the end. This parameter has no effect if the no_auto_update attribute is set to True.

See also:

share(), unshare()

update (fmt={}, replot=False, auto_update=False, draw=None, force=False, todefault=False, **kwargs)

Update the formatoptions and the plot

If the data attribute of this plotter is None, the plotter is updated like a usual dictionary (see dict.update()). Otherwise the update is registered and the plot is updated if auto_update is True or if the start_update() method is called (see below).

Parameters

- **fmt** *(dict)* – Keys can be any valid formatoptions with the corresponding values (see the formatoptions attribute)
- **replot** *(bool)* – Boolean that determines whether the data specific formatoptions shall be updated in any case or not.
- **force** *(str, list of str or bool)* – If formatoption key (i.e. string) or list of formatoption keys, they are definitely updated whether they changed or not. If True, all the given formatoptions in this call of the are update() method are updated
- **todefault** *(bool)* – If True, all changed formatoptions (except the registered ones) are updated to their default value as stored in the rc attribute
- **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the psyplot.rcParams dictionary
- **queues** *(list of Queue.Queue instances)* – The queues that are passed to the psyplot.plotter.Plotter.start_update() method to ensure a thread-safe update. It can be None if only one single plotter is updated at the same time. The number of jobs that are taken from the queue is determined by the _njobs() attribute. Note that there this parameter is automatically configured when updating from a Project.
- **auto_update** *(bool)* – Boolean determining whether or not the start_update() method is called at the end. This parameter has no effect if the no_auto_update attribute is set to True.
- ****kwargs** – Any other formatoption that shall be updated (additionally to those in fmt)
Notes

If the `no_auto_update` attribute is True and the given `auto_update` parameter are is False, the update of the plots are registered and conducted at the next call of the `start_update()` method or the next call of this method (if the `auto_update` parameter is then True).

class psyplot.plotter.PostProcDependencies

Bases: object

The dependencies of this formatoption

class psyplot.plotter.PostProcessing(key, plotter=None, index_in_list=None, additional_children=[], additional_dependencies=[], **kwargs)

Bases: psyplot.plotter.Formatoption

Apply your own postprocessing script

This formatoption let’s you apply your own post processing script. Just enter the script as a string and it will be executed. The formatoption will be made available via the `self` variable

Possible types

Attributes

| children  | Built-in mutable sequence. |
| data_dependent | True if the corresponding `post_timing` |
| dependencies | Built-in mutable sequence. |
| group | str(object='') -> str |
| name | str(object='') -> str |
| post_timing | post_timing Formatoption instance in the plotter |
| priority | Convert a string or number to a floating point number, if possible |

Methods

| update(value) | Method that is call to update the formatoption on the axes |
| validate(value) | Validation method of the formatoption |

• `None` – Don’t do anything
• `str` – The post processing script as string

Note: This formatoption uses the built-in `exec()` function to compile the script. Since this poses a security risk when loading psyplot projects, it is by default disabled through the `Plotter.enable_post` attribute. If you are sure that you can trust the script in this formatoption, set this attribute of the corresponding `Plotter` to `True`.

Examples

Assume, you want to manually add the mean of the data to the title of the matplotlib axes. You can simply do this via
from psyplot.plotter import Plotter
from xarray import DataArray
plotter = Plotter(DataArray([1, 2, 3]))
# enable the post formatoption
plotter.enable_post = True
plotter.update(post="self.ax.set_title(str(self.data.mean()))")
plotter.ax.get_title()
'2.0'

By default, the post formatoption is only ran, when it is explicitly updated. However, you can use the post_timing formatoption, to run it automatically. E.g. for running it after every update of the plotter, you can set

plotter.update(post_timing='always')

See also:

post_timing  Determine the timing of this formatoption

Parameters

- **key**(str) – formatoption key in the plotter
- **plotter**(psyplot.plotter.Plotter) – Plotter instance that holds this formatoption. If None, it is assumed that this instance serves as a descriptor.
- **index_in_list**(int or None) – The index that shall be used if the data is a psyplot.InteractiveList
- **additional_children**(list or str) – Additional children to use (see the children attribute)
- **additional_dependencies**(list or str) – Additional dependencies to use (see the dependencies attribute)
- ****kwargs – Further keywords may be used to specify different names for children, dependencies and connection formatoptions that match the setup of the plotter. Hence, keywords may be anything of the children, dependencies and connections attributes, with values being the name of the new formatoption in this plotter.

children = ['post_timing']
data_dependent
True if the corresponding post_timing formatoption is set to 'replot' to run the post processing script after every change of the data
default = None
dependencies = []
group = 'post_processing'
name = 'Custom post processing script'
post_timing
  post_timing Formatoption instance in the plotter
priority = -inf
update \((value)\)
Method that is call to update the formatoption on the axes

Parameters

- **value** – Value to update

static validate \((value)\)
Validation method of the formatoption

---

**class psyplot.plotter.PostTiming** (key, plotter=None, index_in_list=None, additional_children=[], additional_dependencies=[], **kwargs)

**Bases:** psyplot.plotter.Formatoption

Determine when to run the post formatoption

This formatoption determines, whether the post formatoption should be run never, after replot or after every update.

**Possible types**

**Attributes**

- **default**
- **group**
- **name**
- **priority**

- Convert a string or number to a floating point number, if possible.

**Methods**

- **get_fmt_widget** (parent, project)
- **update** (value)
- **validate** (value)

- ‘never’ – Never run post processing scripts
- ‘always’ – Always run post processing scripts
- ‘replot’ – Only run post processing scripts when the data changes or a replot is necessary

See also:

**post** The post processing formatoption

Parameters

- **key** *(str)* – formatoption key in the plotter
- **plotter** *(psyplot.plotter.Plotter)* – Plotter instance that holds this formatoption. If None, it is assumed that this instance serves as a descriptor.
- **index_in_list** *(int or None)* – The index that shall be used if the data is a psyplot.InteractiveList
- **additional_children** *(list or str)* – Additional children to use (see the children attribute)
• `additional_dependencies(list or str)` – Additional dependencies to use (see the dependencies attribute)

• `**kwargs` – Further keywords may be used to specify different names for children, dependencies and connection formatoptions that match the setup of the plotter. Hence, keywords may be anything of the `children`, `dependencies` and `connections` attributes, with values being the name of the new formatoption in this plotter.

```
default = 'never'
```

`get_fmt_widget(parent, project)`

Get a widget to update the formatoption in the GUI

This method should return a QWidget that is loaded by the psyplot-gui when the formatoption is selected in the `psyplot_gui.main.Mainwindow.fmt_widget`. It should call the `insert_text()` method when the update text for the formatoption should be changed.

**Parameters**

- `parent` (*psyplot_gui.fmt_widget.FormatoptionWidget*) – The parent widget that contains the returned QWidget

- `project` (*psyplot.project.Project*) – The current subproject (see `psyplot.project.gcp()`)

**Returns** The widget to control the formatoption

**Return type** PyQt5.QtWidgets.QWidget

```
group = 'post_processing'
name = 'Timing of the post processing'
priority = -inf
```

`update(value)`

Method that is call to update the formatoption on the axes

**Parameters**

- `value` – Value to update

```
static validate(value)
```

Validation method of the formatoption

```
psyplot.plotter.START = 30
```

Priority value of formatoptions that are updated before the data is loaded.

```
psyplot.plotter.default_print_func()
```

the default function to use when printing formatoption infos (the default is use print or in the gui, use the help explorer)

```
psyplot.plotter.format_time(x)
```

Formats date values

This function formats `datetime.datetime` and `datetime.timedelta` objects (and the corresponding numpy objects) using the `xarray.core.formatting.format_timestamp()` and the `xarray.core.formatting.format_timedelta()` functions.

**Parameters**

- `x (object)` – The value to format. If not a time object, the value is returned

**Returns** Either the formatted time object or the initial `x`

**Return type** str or `x`

```
psyplot.plotter.groups = {'axes': 'Axes formatoptions', 'colors': 'Color coding formatoptions', 'data': 'Data manipulation formatoptions', 'grid': 'Grid formatoptions', 'labels': 'Labeling of axes formatoptions', 'map': 'Map formatoptions', 'regression': 'Fitting formatoptions', 'ticks': 'Axis tick formatoptions', 'vector': 'Vector plot formatoptions'}
```

`dict`. Mapping from group to group names
psyplot.plotter.is_data_dependent(fmto, data)

Check whether a formatoption is data dependent

Parameters

- **fmto** (*Formatoption*) – The *Formatoption* instance to check
- **data** (*xarray.DataArray*) – The data array to use if the *data_dependent* attribute is a callable

Returns

True, if the formatoption depends on the data

Return type **bool**

**psyplot.plugin_template module**

Module for creating a new template for a psyplot plugin

**Functions**

```python
main([args])
new_plugin(odir[, py_name, version, description])
```

Create a new plugin for the psyplot package

**psyplot.plugin_template.main** *(args=None)*

**psyplot.plugin_template.new_plugin** *(odir, py_name=None, version='0.0.1.dev0', description='New plugin')*

Create a new plugin for the psyplot package

Parameters

- **odir** (*str*) – The name of the directory where the data will be copied to. The directory must not exist! The name of the directory also defines the name of the package.
- **py_name** (*str*) – The name of the python package. If None, the basename of *odir* is used (and ‘-’ is replaced by ‘_’)
- **version** (*str*) – The version of the package
- **description** (*str*) – The description of the plugin

**psyplot.project module**

Project module of the psyplot Package

This module contains the *Project* class that serves as the main part of the psyplot API. One instance of the *Project* class serves as coordinator of multiple plots and can be distributed into subprojects that keep reference to the main project without holding all array instances

Furthermore this module contains an easy pyplot-like API to the current subproject.

**Classes**

```python
Cdo(*args, **kwargs)
```

Subclass of the original cdo.Cdo class in the cdo.py module

```python
DataArrayPlotter(da, *args, **kwargs)
```

Interface between the *xarray.Dataset* and the psyplot project

Continued on next page
### Table 67 – continued from previous page

<table>
<thead>
<tr>
<th>Class/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataArrayPlotter</strong></td>
<td>Interface for the <code>DataArrayPlotter</code> to a plotter</td>
</tr>
<tr>
<td><strong>DatasetPlotter</strong></td>
<td>Interface between the <code>xarray.Dataset</code> and the psyplot project</td>
</tr>
<tr>
<td><strong>DatasetPlotterInterface</strong></td>
<td>Interface for the <code>DatasetPlotter</code> to a plotter</td>
</tr>
<tr>
<td><strong>PROJECT_CLS</strong></td>
<td>The project class that is used for creating new projects</td>
</tr>
<tr>
<td><strong>PlotterInterface</strong></td>
<td>Base class for visualizing a data array from an predefined plotter</td>
</tr>
<tr>
<td><strong>Project</strong></td>
<td>A manager of multiple interactive data projects</td>
</tr>
<tr>
<td><strong>ProjectPlotter</strong></td>
<td>Plotting methods of the <code>psyplot.project.Project</code></td>
</tr>
</tbody>
</table>

### Functions

- `close([num, figs, data, ds, remove_only])`: Close the project
- `gcp([main])`: Get the current project
- `get_project_nums()`: Returns the project numbers of the open projects
- `multiple_subplots([rows, cols, maxplots, n, ...])`: Function to create subplots.
- `project([num])`: Create a new main project
- `register_plotter(identifier, module, ...[, ...])`: Register a `psyplot.plotter.Plotter` for the projects
- `scp(project)`: Set the current project
- `unregister_plotter(identifier[, sorter, ...])`: Unregister a `psyplot.plotter.Plotter` for the projects

### Data

- `plot`: `ProjectPlotter` of the current project. See the class documentation

```python
class psyplot.project.Cdo(*args, **kwargs)
    Bases: cdo.Cdo

    Subclass of the original cdo.Cdo class in the cdo.py module

    Requirements are a working cdo binary and the installed cdo.py python module.

    For a documentation of an operator, use the python help function, for a list of operators, use the builtin dir function. Further documentation on the operators can be found here: https://code.zmaw.de/projects/cdo/wiki/Cdo%7Brbpy%7D and on the usage of the cdo.py module here: https://code.zmaw.de/projects/cdo

    For a demonstration script on how cdos are implemented, see the examples of the psyplot package

    Compared to the original cdo.Cdo class, the following things changed, the default cdf handler is the `psyplot.data.open_dataset()` function and the following keywords are implemented for each cdo operator. If any of those is specified, the return will be a subproject (i.e. an instance of `psyplot.project.Project`)

    Other Parameters

    - `plot_method (str or psyplot.project.PlotterInterface)`: An registered plotting function to plot the data (e.g. `psyplot.project.plot_mapplot` to plot on a map). If `None`, no plot will be created. In any case, the returned value is a subproject. If string, it must correspond to the attribute of the `psyplot.project.ProjectPlotter` class
```

152 Chapter 1. Documentation
• **name** *(str or list of str)* – The variable names to plot/extract

• **fmt** *(dict)* – Format options that shall be when initializing the plot (you can however also specify them as extra keyword arguments)

• **make_plot** *(bool)* – If True, the data is plotted at the end. Otherwise you have to call the `psyplot.plotter.Plotter.initialize_plot()` method or the `psyplot.plotter.Plotter.reinit()` method by yourself

• **ax** *(None, tuple (x, y[, z]) or (list of) matplotlib.axes.Axes)* – Specifies the subplots on which to plot the new data objects.
  – If None, a new figure will be created for each created plotter
  – If tuple (x, y[, z]), x specifies the number of rows, y the number of columns and the optional third parameter z the maximal number of subplots per figure.
  – If `matplotlib.axes.Axes` (or list of those, e.g. created by the `matplotlib.pyplot.subplots()` function), the data will be plotted on these subplots

• **method** *(‘isel’, None, ‘nearest’, ...)* – Selection method of the xarray.Dataset to be used for setting the variables from the informations in `dims`. If `method` is ‘isel’, the `xarray.Dataset.isel()` method is used. Otherwise it sets the `method` parameter for the `xarray.Dataset.sel()` method.

**Methods**

```python
loadCdf(*args, **kwargs)
```
Load data handler as specified by `self.cdfMod`

**Examples**

Calculate the timmean of a 3-dimensional array and plot it on a map using the psy-maps package

```python
cdo = psy.Cdo()
sp = cdo.timmean(input='ifile.nc', name='temperature',
                 plot_method='mapplot')
```

which is essentially the same as

```python
sp = cdo.timmean(input='ifile.nc', name='temperature',
                 plot_method=psy.plot.mapplot)
# and
sp = psy.plot.mapplot(
    cdo.timmean(input='ifile.nc', returnCdf=True),
    name='temperature', plot_method=psy.plot.mapplot)
```

```python
loadCdf(*args, **kwargs)
```
Load data handler as specified by `self.cdfMod`

**class** `psyplot.project.DataArrayPlotter` *(da, *args, **kwargs)*

**Bases:** `psyplot.project.ProjectPlotter`

Interface between the `xarray.Dataset` and the psyplot project

This class can be used to make new plots from a given dataset and add them to the current `psyplot.project()` Attributes
**barplot**(*args, **kwargs)*

Make a bar plot of one-dimensional data

This plotting method visualizes the data via a psy_simple.plotters.BarPlotter plotters.

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.barplot()
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>alpha</th>
<th>axiscolor</th>
<th>categorical</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
</tr>
<tr>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
<td>sym_lims</td>
</tr>
<tr>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
<td>tight</td>
</tr>
<tr>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
</tr>
<tr>
<td>transpose</td>
<td>widths</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
</tr>
<tr>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
<td>yticklabels</td>
</tr>
<tr>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

To explore the formatoptions and their documentations, use the **keys**, **summaries** and **docs** methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> da.psy.plot.barplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> da.psy.plot.barplot.summaries('title')

# show the full documentation
```
combined(*args, **kwargs)
Plot a 2D scalar field with an overlying vector field

This plotting method visualizes the data via a psy_simple.plotters.
CombinedSimplePlotter plotters

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.combined()
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>arrosize</th>
<th>arrowstyle</th>
<th>axiscolor</th>
<th>bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbar</td>
<td>cbar_spacing</td>
<td>clabel</td>
<td>clabelprops</td>
</tr>
<tr>
<td>clabelsize</td>
<td>clabelweight</td>
<td>cmap</td>
<td>color</td>
</tr>
<tr>
<td>cticklabels</td>
<td>ctickprops</td>
<td>cticks</td>
<td>cticksize</td>
</tr>
<tr>
<td>ctickweight</td>
<td>datagrid</td>
<td>density</td>
<td>extend</td>
</tr>
<tr>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
</tr>
<tr>
<td>grid</td>
<td>interp_bounds</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>levels</td>
<td>linewidth</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
</tr>
<tr>
<td>miss_color</td>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>vbounds</td>
<td>vbar</td>
</tr>
<tr>
<td>vbarspacing</td>
<td>vclabel</td>
<td>vclabelprops</td>
<td>vclabelsize</td>
</tr>
<tr>
<td>vclabelweight</td>
<td>vcmap</td>
<td>vcticklabels</td>
<td>vctickprops</td>
</tr>
<tr>
<td>vcticks</td>
<td>vcticksize</td>
<td>vctickweight</td>
<td>vplot</td>
</tr>
<tr>
<td>xlabel</td>
<td>xlim</td>
<td>xrotation</td>
<td>xticklabels</td>
</tr>
<tr>
<td>xtickprops</td>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>

Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple
# formatoptions
>>> da.psy.plot.combined.keys('labels')

# show the summaries of a group of formatoptions or of a
# formatoption
>>> da.psy.plot.combined.summaries('title')

# show the full documentation
>>> da.psy.plot.combined.docs('plot')
```

(continues on next page)
density(*args, **kwargs)

Make a density plot of point data

This plotting method visualizes the data via a psy_simple.plotters.DensityPlotter plotters

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.density()
```

Possible format options are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>bins</th>
<th>bounds</th>
<th>cbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>clabelsize</td>
</tr>
<tr>
<td>clabelweight</td>
<td>cmap</td>
<td>coord</td>
<td>cticklabels</td>
</tr>
<tr>
<td>ctickprops</td>
<td>cticks</td>
<td>cticksize</td>
<td>ctickweight</td>
</tr>
<tr>
<td>datagrid</td>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
</tr>
<tr>
<td>ftitleprops</td>
<td>fgititlesize</td>
<td>fgititleweight</td>
<td>grid</td>
</tr>
<tr>
<td>interp_bounds</td>
<td>labelprops</td>
<td>labelsize</td>
<td>labelweight</td>
</tr>
<tr>
<td>levels</td>
<td>maskbetween</td>
<td>maskgeq</td>
<td>maskgreater</td>
</tr>
<tr>
<td>maskleq</td>
<td>maskless</td>
<td>miss_color</td>
<td>normed</td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
<td>precision</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrange</td>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
</tr>
<tr>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
<td>yrange</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>

Examples

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple # format options
>>> da.psy.plot.density.keys('labels')

# show the summaries of a group of format options or of a # format option
>>> da.psy.plot.density.summaries('title')

# show the full documentation
>>> da.psy.plot.density.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.density.plot
```
**fldmean** (*args, **kwargs)

Calculate and plot the mean over x- and y-dimensions

This plotting method visualizes the data via a psy_simple.plotters.FldmeanPlotter plotters.

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.fldmean()
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>color</th>
<th>coord</th>
<th>err_calc</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>errortitle</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
</tr>
<tr>
<td>labelsize</td>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
</tr>
<tr>
<td>linewidth</td>
<td>marker</td>
<td>markersize</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleg</td>
<td>maskless</td>
</tr>
<tr>
<td>mean</td>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
</tr>
<tr>
<td>xlabel</td>
<td>ylabeled</td>
<td>yrotation</td>
<td>yticklabels</td>
</tr>
<tr>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> da.psy.plot.fldmean.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> da.psy.plot.fldmean.summaries('title')

# show the full documentation
>>> da.psy.plot.fldmean.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.fldmean.plot
```

**lineplot** (*args, **kwargs)

Make a line plot of one-dimensional data

This plotting method visualizes the data via a psy_simple.plotters.LinePlotter plotters.

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.lineplot()
```
Possible format options are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>color</th>
<th>coord</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>erroralpha</td>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
</tr>
<tr>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
<td>linewidth</td>
</tr>
<tr>
<td>marker</td>
<td>markersize</td>
<td>maskbetween</td>
<td>maskgeq</td>
</tr>
<tr>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
<td>plot</td>
</tr>
<tr>
<td>post</td>
<td>post_timing</td>
<td>sym_lims</td>
<td>text</td>
</tr>
<tr>
<td>ticksize</td>
<td>tickweight</td>
<td>tight</td>
<td>title</td>
</tr>
<tr>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
<td>transpose</td>
</tr>
<tr>
<td>xlabel</td>
<td>xlim</td>
<td>xrotation</td>
<td>xticklabels</td>
</tr>
<tr>
<td>xtickprops</td>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>

Examples

To explore the format options and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple
# format options
>>> da.psy.plot.lineplot.keys('labels')

# show the summaries of a group of format options or of a
# formatoption
>>> da.psy.plot.lineplot.summaries('title')

# show the full documentation
>>> da.psy.plot.lineplot.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.lineplot.plot
```

`mapcombined(*args, **kwargs)`

Plot a 2D scalar field with an overlying vector field on a map

This plotting method visualizes the data via a `psy_maps.plotters.CombinedPlotter` plotters

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.mapcombined()
```

Possible format options are
Examples

To explore the formatoptions and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple # formatoptions
>>> da.psy.plot.mapcombined.keys('labels')

# show the summaries of a group of formatoptions or of a # formatoption
>>> da.psy.plot.mapcombined.summaries('title')

# show the full documentation
>>> da.psy.plot.mapcombined.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.mapcombined.plot
```

`mapplot(*args, **kwargs)`

Plot a 2D scalar field on a map

This plotting method visualizes the data via a `psy_maps.plotters.FieldPlotter` plotters.

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.mapplot()
```

Possible formatoptions are
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple
# formatoptions
>>> da.psy.plot.mapplot.keys('labels')

# show the summaries of a group of formatoptions or of a
# formatoption
>>> da.psy.plot.mapplot.summaries('title')

# show the full documentation
>>> da.psy.plot.mapplot.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.mapplot.plot
```

**mapvector** (*args, **kwargs)

Plot a 2D vector field on a map

This plotting method visualizes the data via a psy_maps.plotters.VectorPlotter plotters

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.mapvector()
```

Possible formatoptions are

```
<table>
<thead>
<tr>
<th>bounds</th>
<th>cbar</th>
<th>cbarSpacing</th>
<th>clabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>clabelprops</td>
<td>clabelsize</td>
<td>clabelweight</td>
<td>clat</td>
</tr>
<tr>
<td>clip</td>
<td>clon</td>
<td>cmap</td>
<td>cticklabels</td>
</tr>
<tr>
<td>ctickprops</td>
<td>cticks</td>
<td>cticksize</td>
<td>ctickweight</td>
</tr>
<tr>
<td>datagrid</td>
<td>extend</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid_color</td>
<td>grid_labels</td>
</tr>
<tr>
<td>grid_labelsize</td>
<td>grid_settings</td>
<td>interp_bounds</td>
<td>levels</td>
</tr>
<tr>
<td>lonlatbox</td>
<td>lsm</td>
<td>map_extent</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskeq</td>
<td>maskless</td>
</tr>
<tr>
<td>miss_color</td>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
</tr>
<tr>
<td>projection</td>
<td>stock_img</td>
<td>text</td>
<td>tight</td>
</tr>
<tr>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
</tr>
<tr>
<td>transform</td>
<td>xgrid</td>
<td>ygrid</td>
<td></td>
</tr>
</tbody>
</table>
```
Examples

To explore the format options and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple format options
>>> da.psy.plot.mapvector.keys('labels')

# show the summaries of a group of format options or of a format option
>>> da.psy.plot.mapvector.summaries('title')

# show the full documentation
>>> da.psy.plot.mapvector.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.mapvector.plot
```

`plot2d(*args, **kwargs)`

Make a simple plot of a 2D scalar field

This plotting method visualizes the data via a `psy_simple.plotters.Simple2DPlotter` plotters

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.plot2d()
```

Possible format options are
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> da.psy.plot.plot2d.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> da.psy.plot.plot2d.summaries('title')

# show the full documentation
>>> da.psy.plot.plot2d.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.plot2d.plot
```

**vector** (*args, **kwargs)

Make a simple plot of a 2D vector field

This plotting method visualizes the data via a psy_simple.plotters.SimpleVectorPlotter plotters.

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.vector()
```

Possible formatoptions are
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> da.psy.plot.vector.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> da.psy.plot.vector.summaries('title')

# show the full documentation
>>> da.psy.plot.vector.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.vector.plot
```

`violinplot(*args, **kwargs)`

Make a violin plot of your data

This plotting method visualizes the data via a psy_simple.plotters.ViolinPlotter plotters.

To plot a variable in this dataset, type:

```python
>>> da.psy.plot.violinplot()
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>arrowsize</th>
<th>arrowstyle</th>
<th>axiscolor</th>
<th>bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbar</td>
<td>barspacing</td>
<td>clabel</td>
<td>clabelprops</td>
</tr>
<tr>
<td>clabelsize</td>
<td>clabelweight</td>
<td>cmap</td>
<td>color</td>
</tr>
<tr>
<td>cticklabels</td>
<td>cticklabelslength</td>
<td>cticks</td>
<td>cticksize</td>
</tr>
<tr>
<td>ctickweight</td>
<td>datagrid</td>
<td>density</td>
<td>extend</td>
</tr>
<tr>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
</tr>
<tr>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
<td>labelweight</td>
</tr>
<tr>
<td>linewidth</td>
<td>maskbetween</td>
<td>maskgeq</td>
<td>maskgreater</td>
</tr>
<tr>
<td>maskleq</td>
<td>maskless</td>
<td>plot</td>
<td>post</td>
</tr>
<tr>
<td>post_timing</td>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
</tr>
<tr>
<td>tickweight</td>
<td>tight</td>
<td>title</td>
<td>titleweight</td>
</tr>
<tr>
<td>titlesize</td>
<td>titleweight</td>
<td>transpose</td>
<td>xlabel</td>
</tr>
<tr>
<td>xlim</td>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
</tr>
<tr>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
</tr>
<tr>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
</tr>
</tbody>
</table>
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> da.psy.plot.violinplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> da.psy.plot.violinplot.summaries('title')

# show the full documentation
>>> da.psy.plot.violinplot.docs('plot')

# or access the documentation via the attribute
>>> da.psy.plot.violinplot.plot
```

```python
class psyplot.project.DataArrayPlotterInterface (methodname, module, plotter_name, project_plotter=None)
Bases: psyplot.project.PlotterInterface

Interface for the DataArrayPlotter to a plotter

Methods

check_data(*args, **kwargs) Check whether the plotter of this plot method can visualize the data
```

```python
class psyplot.project.DatasetPlotter (ds, *args, **kwargs)
Bases: psyplot.project.ProjectPlotter

Interface between the xarray.Dataset and the psyplot project

This class can be used to make new plots from a given dataset and add them to the current psyplot.

data

Attributes

class psyplot.project.DatasetPlotter (ds, *args, **kwargs)
Bases: psyplot.project.ProjectPlotter

Interface between the xarray.Dataset and the psyplot project

This class can be used to make new plots from a given dataset and add them to the current psyplot.

Attributes

barplot(*args, **kwargs) Make a bar plot of one-dimensional data
combined(*args, **kwargs) Plot a 2D scalar field with an overlying vector field
```

Continued on next page
Table 73 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>density(*args, **kwargs)</code></td>
<td>Make a density plot of point data</td>
</tr>
<tr>
<td><code>fldmean(*args, **kwargs)</code></td>
<td>Calculate and plot the mean over x- and y-dimensions</td>
</tr>
<tr>
<td><code>lineplot(*args, **kwargs)</code></td>
<td>Make a line plot of one-dimensional data</td>
</tr>
<tr>
<td><code>mapcombined(*args, **kwargs)</code></td>
<td>Plot a 2D scalar field with an overlying vector field on a map</td>
</tr>
<tr>
<td><code>mapplot(*args, **kwargs)</code></td>
<td>Plot a 2D scalar field on a map</td>
</tr>
<tr>
<td><code>mapvector(*args, **kwargs)</code></td>
<td>Plot a 2D vector field on a map</td>
</tr>
<tr>
<td><code>plot2d(*args, **kwargs)</code></td>
<td>Make a simple plot of a 2D scalar field</td>
</tr>
<tr>
<td><code>vector(*args, **kwargs)</code></td>
<td>Make a simple plot of a 2D vector field</td>
</tr>
<tr>
<td><code>violinplot(*args, **kwargs)</code></td>
<td>Make a violin plot of your data</td>
</tr>
</tbody>
</table>

**barplot(*args, **kwargs)**

Make a bar plot of one-dimensional data

This plotting method adds data arrays and plots them via `psy_simple.plotters.BarPlotter` plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.barplot(name=['my_variable'], ...)
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>alpha</th>
<th>axiscolor</th>
<th>categorical</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
</tr>
<tr>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
</tr>
<tr>
<td>maskgreater</td>
<td>maskless</td>
<td>symmetry</td>
<td>sym_limits</td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
<td>sym_metrics</td>
</tr>
<tr>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
<td>tight</td>
</tr>
<tr>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
</tr>
<tr>
<td>transpose</td>
<td>widths</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
</tr>
<tr>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
<td>yticklabels</td>
</tr>
<tr>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

To explore the formatoptions and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> ds.psy.plot.barplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> ds.psy.plot.barplot.summaries('title')

# show the full documentation
>>> ds.psy.plot.barplot.docs('plot')
```

(continues on next page)
combined(*args, **kwargs)
Plot a 2D scalar field with an overlying vector field

This plotting method adds data arrays and plots them via psy_simple.plotters.
CombinedSimplePlotter plotters

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.combined(name=[['my_variable'], ['u_var', 'v_var']]), ...)
```

Possible format options are

<table>
<thead>
<tr>
<th>arrowsize</th>
<th>arrowstyle</th>
<th>axiscolor</th>
<th>bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbar</td>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
</tr>
<tr>
<td>clabelsize</td>
<td>clabelweight</td>
<td>cmap</td>
<td>color</td>
</tr>
<tr>
<td>cticklabels</td>
<td>cтикprops</td>
<td>cticks</td>
<td>cticksize</td>
</tr>
<tr>
<td>ctickweight</td>
<td>datagrid</td>
<td>density</td>
<td>extend</td>
</tr>
<tr>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
</tr>
<tr>
<td>grid</td>
<td>interp_bounds</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>levels</td>
<td>linewidth</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
</tr>
<tr>
<td>miss_color</td>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>vbounds</td>
<td>vcbarspacing</td>
</tr>
<tr>
<td>vcbarspacing</td>
<td>vclabel</td>
<td>vclabelprops</td>
<td>vclabelsize</td>
</tr>
<tr>
<td>vclabelweight</td>
<td>vcmap</td>
<td>vcticklabels</td>
<td>vctickprops</td>
</tr>
<tr>
<td>vcticks</td>
<td>vcticksizesize</td>
<td>vctickweight</td>
<td>vplot</td>
</tr>
<tr>
<td>xlabel</td>
<td>xlim</td>
<td>xrotation</td>
<td>xticklabels</td>
</tr>
<tr>
<td>xtickprops</td>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>

**Examples**

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple format options
>>> ds.psy.plot.combined.keys('labels')

# show the summaries of a group of format options or of a format option
>>> ds.psy.plot.combined.summaries('title')

# show the full documentation
>>> ds.psy.plot.combined.docs('plot')
```
density(*args, **kwargs)

Make a density plot of point data

This plotting method adds data arrays and plots them via psy_simple.plotters.

DensityPlotter plotters

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.density(name=['my_variable'], ...)
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>bins</th>
<th>bounds</th>
<th>cbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbar spacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>clabelsize</td>
</tr>
<tr>
<td>cbar weight</td>
<td>cgrid</td>
<td>cgridprops</td>
<td>cgrid size</td>
</tr>
<tr>
<td>cgrid props</td>
<td>cgrid size</td>
<td>cgrid weight</td>
<td></td>
</tr>
<tr>
<td>cgrid</td>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
</tr>
<tr>
<td>figtitle props</td>
<td>figtitle size</td>
<td>figtitle weight</td>
<td>grid</td>
</tr>
<tr>
<td>interp_grid</td>
<td>labelprops</td>
<td>label size</td>
<td>labelweight</td>
</tr>
<tr>
<td>levels</td>
<td>maskbetween</td>
<td>maskgreater</td>
<td>mask less</td>
</tr>
<tr>
<td>mask</td>
<td>maskless</td>
<td>miss color</td>
<td>normed</td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post timing</td>
<td>precision</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tick weight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>title props</td>
<td>title size</td>
</tr>
<tr>
<td>title weight</td>
<td>transpose</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrange</td>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
</tr>
<tr>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
<td>yrange</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>

Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple format options
>>> ds.psy.plot.density.keys('labels')

# show the summaries of a group of format options or of a format option
>>> ds.psy.plot.density.summaries('title')

# show the full documentation
>>> ds.psy.plot.density.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.density.plot
```
**fldmean** (*args, **kwargs)

Calculate and plot the mean over x- and y-dimensions

This plotting method adds data arrays and plots them via `psy_simple.plotters.FldmeanPlotter` plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.fldmean(name=['my_variable'], ...)
```

Possible format options are:

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>color</th>
<th>coord</th>
<th>err_calc</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>erronalpha</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
</tr>
<tr>
<td>labelsize</td>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
</tr>
<tr>
<td>linewidth</td>
<td>marker</td>
<td>markersize</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleg</td>
<td>maskless</td>
</tr>
<tr>
<td>mean</td>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>xlabell</td>
<td>xim</td>
</tr>
<tr>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
</tr>
<tr>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
<td>yticklabels</td>
</tr>
<tr>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

To explore the format options and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple format options
>>> ds.psy.plot.fldmean.keys('labels')

# show the summaries of a group of format options or of a format option
>>> ds.psy.plot.fldmean.summaries('title')

# show the full documentation
>>> ds.psy.plot.fldmean.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.fldmean.plot
```

**lineplot** (*args, **kwargs)

Make a line plot of one-dimensional data

This plotting method adds data arrays and plots them via `psy_simple.plotters.LinePlotter` plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.lineplot(name=['my_variable'], ...)
```
Possible format options are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>color</th>
<th>coord</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>erroralpha</td>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
</tr>
<tr>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
<td>linewidth</td>
</tr>
<tr>
<td>marker</td>
<td>markersize</td>
<td>maskbetween</td>
<td>maskgeq</td>
</tr>
<tr>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
<td>plot</td>
</tr>
<tr>
<td>post</td>
<td>post_timing</td>
<td>sym_lims</td>
<td>text</td>
</tr>
<tr>
<td>ticksize</td>
<td>tickweight</td>
<td>tight</td>
<td>title</td>
</tr>
<tr>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
<td>transpose</td>
</tr>
<tr>
<td>xlabel</td>
<td>xlim</td>
<td>xrotation</td>
<td>xticklabels</td>
</tr>
<tr>
<td>xtickprops</td>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>

Examples

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple # format options
>>> ds.psy.plot.lineplot.keys('labels')

# show the summaries of a group of formatoptions or of a # formatoption
>>> ds.psy.plot.lineplot.summaries('title')

# show the full documentation
>>> ds.psy.plot.lineplot.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.lineplot.plot
```

`mapcombined(*args, **kwargs)`

Plot a 2D scalar field with an overlying vector field on a map

This plotting method adds data arrays and plots them via `psy_maps.plotters`. CombinedPlotter plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.mapcombined(name=[['my_variable', 'u_var', 'v_var']]), ...)
```

Possible format options are
Examples

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple
# format options
>>> ds.psy.plot.mapcombined.keys('labels')

# show the summaries of a group of format options or of a
# format option
>>> ds.psy.plot.mapcombined.summaries('title')

# show the full documentation
>>> ds.psy.plot.mapcombined.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.mapcombined.plot
```

**mapplot (**`*args`, **`**kwargs`**)**

Plot a 2D scalar field on a map

This plotting method adds data arrays and plots them via psy_maps.plotters. FieldPlotter plotters

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.mapplot(name=['my_variable'], ...)
```

Possible format options are

```plaintext
<table>
<thead>
<tr>
<th>arrowsize</th>
<th>arrowstyle</th>
<th>bounds</th>
<th>cbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>clabelsize</td>
</tr>
<tr>
<td>clabelweight</td>
<td>clat</td>
<td>clon</td>
<td>cclon</td>
</tr>
<tr>
<td>cmap</td>
<td>color</td>
<td>cticklabels</td>
<td>ctickprops</td>
</tr>
<tr>
<td>cticks</td>
<td>cticksize</td>
<td>ctickweight</td>
<td>datagrid</td>
</tr>
<tr>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid_color</td>
<td>grid_labels</td>
</tr>
<tr>
<td>grid_labelsize</td>
<td>grid_settings</td>
<td>interp_bounds</td>
<td>levels</td>
</tr>
<tr>
<td>linewidth</td>
<td>lonlatbox</td>
<td>lsm</td>
<td>map_extent</td>
</tr>
<tr>
<td>maskbetween</td>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
</tr>
<tr>
<td>maskless</td>
<td>miss_color</td>
<td>plot</td>
<td>post</td>
</tr>
<tr>
<td>post_timing</td>
<td>projection</td>
<td>stock_img</td>
<td>text</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transform</td>
<td>vbounds</td>
<td>vcbarspacing</td>
</tr>
<tr>
<td>vcbarspacing</td>
<td>vclabel</td>
<td>vclabelprops</td>
<td>vclabelsize</td>
</tr>
<tr>
<td>vclabelweight</td>
<td>vcmapp</td>
<td>vcticklabels</td>
<td>vctickprops</td>
</tr>
<tr>
<td>vcticks</td>
<td>vcticksize</td>
<td>vctickweight</td>
<td>vplot</td>
</tr>
<tr>
<td>xgrid</td>
<td>ygrid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Examples

To explore the formatoptions and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> ds.psy.plot.mapplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> ds.psy.plot.mapplot.summaries('title')

# show the full documentation
>>> ds.psy.plot.mapplot.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.mapplot.plot
```

**mapvector** (*args, **kwargs)

Plot a 2D vector field on a map

This plotting method adds data arrays and plots them via `psy_maps.plotters.VectorPlotter` plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.mapvector(name=[['u_var', 'v_var'], ...])
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>bounds</th>
<th>cbar</th>
<th>cbar spacing</th>
<th>clabel</th>
</tr>
</thead>
<tbody>
<tr>
<td>clabel props</td>
<td>clabel size</td>
<td>clabel weight</td>
<td>clat</td>
</tr>
<tr>
<td>clip</td>
<td>clon</td>
<td>cmap</td>
<td>cltick labels</td>
</tr>
<tr>
<td>ctick props</td>
<td>cticks</td>
<td>ctick size</td>
<td>ctick weight</td>
</tr>
<tr>
<td>datagrid</td>
<td>extend</td>
<td>figtitle</td>
<td>figtitle props</td>
</tr>
<tr>
<td>fig titles size</td>
<td>figtitle weight</td>
<td>grid color</td>
<td>grid labels</td>
</tr>
<tr>
<td>grid label size</td>
<td>grid settings</td>
<td>interp bounds</td>
<td>levels</td>
</tr>
<tr>
<td>lonlat box</td>
<td>lsm</td>
<td>map extent</td>
<td>mask between</td>
</tr>
<tr>
<td>mask geq</td>
<td>mask greater</td>
<td>mask eq</td>
<td>mask less</td>
</tr>
<tr>
<td>miss color</td>
<td>plot</td>
<td>post</td>
<td>post timing</td>
</tr>
<tr>
<td>projection</td>
<td>stock_img</td>
<td>text</td>
<td>tight</td>
</tr>
<tr>
<td>title</td>
<td>title props</td>
<td>title size</td>
<td>title weight</td>
</tr>
<tr>
<td>transform</td>
<td>xgrid</td>
<td>ygrid</td>
<td></td>
</tr>
<tr>
<td>arrowsize</td>
<td>arrowstyle</td>
<td>bounds</td>
<td>cbar</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>clabelweight</td>
<td>clat</td>
<td>clip</td>
<td>clon</td>
</tr>
<tr>
<td>cmap</td>
<td>color</td>
<td>cticklabels</td>
<td>ctickprops</td>
</tr>
<tr>
<td>cticks</td>
<td>cticksize</td>
<td>ctickweight</td>
<td>datagrid</td>
</tr>
<tr>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid_color</td>
<td>grid_labels</td>
</tr>
<tr>
<td>grid_labelsize</td>
<td>grid_settings</td>
<td>linewidth</td>
<td>lonlatbox</td>
</tr>
<tr>
<td>lsm</td>
<td>map_extent</td>
<td>maskbetween</td>
<td>maskleq</td>
</tr>
<tr>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
<td>plot</td>
</tr>
<tr>
<td>post</td>
<td>post_timing</td>
<td>projection</td>
<td>stock_img</td>
</tr>
<tr>
<td>text</td>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
</tr>
<tr>
<td>titlesize</td>
<td>titleweight</td>
<td>transform</td>
<td>xgrid</td>
</tr>
<tr>
<td>ygrid</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Examples

To explore the formatoptions and their documentations, use the `keys`, `summarise` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> ds.psy.plot.mapvector.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> ds.psy.plot.mapvector.summaries('title')

# show the full documentation
>>> ds.psy.plot.mapvector.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.mapvector.plot
```

#### plot2d(*args, **kwargs)

Make a simple plot of a 2D scalar field

This plotting method adds data arrays and plots them via `psy_simple.plotters`. `Simple2DPlotter` plotters

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.plot2d(name=['my_variable'], ...)
```

Possible formatoptions are
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> ds.psy.plot.plot2d.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> ds.psy.plot.plot2d.summaries('title')

# show the full documentation
>>> ds.psy.plot.plot2d.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.plot2d.plot
```

**vector** (*args, **kwargs*)

Make a simple plot of a 2D vector field

This plotting method adds data arrays and plots them via psy_simple.plotters. SimpleVectorPlotter plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.vector(name=[['u_var', 'v_var']], ...)  # Possible formatoptions are...```
Examples

To explore the formatoptions and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
# show the keys corresponding to a group or multiple
# formatoptions
>>> ds.psy.plot.vector.keys('labels')

# show the summaries of a group of formatoptions or of a
# formatoption
>>> ds.psy.plot.vector.summaries('title')

# show the full documentation
>>> ds.psy.plot.vector.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.vector.plot
```

`violinplot(*args, **kwargs)`

Make a violin plot of your data

This plotting method adds data arrays and plots them via `psy_simple.plotTERS.ViolinPlotter` plotters.

To plot a variable in this dataset, type:

```python
>>> ds.psy.plot.violinplot(name=['my_variable'], ...)
```

Possible formatoptions are
### Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
# show the keys corresponding to a group or multiple formatoptions
>>> ds.psy.plot.violinplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> ds.psy.plot.violinplot.summaries('title')

# show the full documentation
>>> ds.psy.plot.violinplot.docs('plot')

# or access the documentation via the attribute
>>> ds.psy.plot.violinplot.plot
```

```python
class psyplot.project.DatasetPlotterInterface (methodname, module, plotter_name, project_plotter=None)

    Bases: psyplot.project.PlotterInterface

    Interface for the DatasetPlotter to a plotter

<table>
<thead>
<tr>
<th>attr_name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr_names</td>
<td>Names of the arrays (!not of the variables!) in this list</td>
</tr>
<tr>
<td>axes</td>
<td>A mapping from axes to data objects with the plotter in this axes</td>
</tr>
<tr>
<td>barplot</td>
<td>List of data arrays that are plotted by psy_simple.plotters.BarPlotter plotters</td>
</tr>
<tr>
<td>block_signals</td>
<td>List of data arrays that are plotted by psy_simple.plotters.BlockSignals Plotter plotters</td>
</tr>
<tr>
<td>combined</td>
<td>List of data arrays that are plotted by psy_simple.plotters.CombinedSimplePlotter plotters</td>
</tr>
<tr>
<td>datasets</td>
<td>A mapping from dataset numbers to datasets in this list</td>
</tr>
<tr>
<td>density</td>
<td>List of data arrays that are plotted by psy_simple.plotters.DensityPlotter plotters</td>
</tr>
<tr>
<td>dsnames</td>
<td>The set of dataset names in this instance</td>
</tr>
<tr>
<td>dsnames_map</td>
<td>A dictionary from the dataset numbers in this list to their</td>
</tr>
<tr>
<td>figs</td>
<td>A mapping from figures to data objects with the plotter in this</td>
</tr>
<tr>
<td>fldmean</td>
<td>List of data arrays that are plotted by psy_simple.plottersFldmeanPlotter plotters</td>
</tr>
<tr>
<td>is_cmp</td>
<td>Boolean that is True if the project is the current main project</td>
</tr>
<tr>
<td>is_csp</td>
<td>Boolean that is True if the project is the current subproject</td>
</tr>
<tr>
<td>is_main</td>
<td>bool. True if this Project is a main project</td>
</tr>
</tbody>
</table>
```

Continued on next page
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(*args, **kwargs)</td>
<td>Append a new array to the list</td>
</tr>
<tr>
<td>close(figs, data, ds, remove_only)</td>
<td>Close this project instance</td>
</tr>
<tr>
<td>disable()</td>
<td>Disables the plotters in this list</td>
</tr>
<tr>
<td>docs(*args, **kwargs)</td>
<td>Show the available format options in this project and their full docu</td>
</tr>
<tr>
<td>enable()</td>
<td></td>
</tr>
<tr>
<td>export(output[, tight, concat, close_pdf, ...])</td>
<td>Exports the figures of the project to one or more image files</td>
</tr>
<tr>
<td>extend(*args, **kwargs)</td>
<td>Add further arrays from an iterable to this list</td>
</tr>
<tr>
<td>from_dataset(*args, **kwargs)</td>
<td>Construct an ArrayList instance from an existing base dataset</td>
</tr>
<tr>
<td>joined_attrs(delimiter, enhanced,...)</td>
<td>Join the attributes of the arrays in this project</td>
</tr>
<tr>
<td>keys(*args, **kwargs)</td>
<td>Show the available format options in this project</td>
</tr>
<tr>
<td>load_project(fname[, auto_update,...])</td>
<td>Load a project from a file or dict</td>
</tr>
<tr>
<td>new(num)</td>
<td>Create a new main project</td>
</tr>
<tr>
<td>save_project(fname, pwd, pack)</td>
<td>Save this project to a file</td>
</tr>
<tr>
<td>scp(project)</td>
<td>Set the current project</td>
</tr>
<tr>
<td>share(base, keys, by)</td>
<td>Share the format options of one plotter with all the others</td>
</tr>
<tr>
<td>show()</td>
<td>Shows all open figures</td>
</tr>
<tr>
<td>summaries(*args, **kwargs)</td>
<td>Show the available format options and their summaries in this project</td>
</tr>
<tr>
<td>unshare(**kwargs)</td>
<td>Unshare the format options of all the plotters in this instance</td>
</tr>
</tbody>
</table>

### Attributes

The project class that is used for creating new projects

**Class** psyplot.project.Project

Bases: object
Base class for visualizing a data array from an predefined plotter

See the `__call__()` method for details on plotting. **Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check_data(ds, name, dims)</code></td>
<td>A validation method for the data shape</td>
</tr>
<tr>
<td><code>docs(*args, **kwargs)</code></td>
<td>Method to print the full documentations of the formatoptions</td>
</tr>
<tr>
<td><code>keys(*args, **kwargs)</code></td>
<td>Classmethod to return a nice looking table with the given formatoptions</td>
</tr>
<tr>
<td><code>summaries(*args, **kwargs)</code></td>
<td>Method to print the summaries of the formatoptions</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plotter_cls</code></td>
<td>The plotter class</td>
</tr>
<tr>
<td><code>print_func</code></td>
<td>The function that is used to return a formatoption</td>
</tr>
</tbody>
</table>

### `check_data(ds, name, dims)`
A validation method for the data shape

**Parameters**

- `name (list of lists of strings)` – The variable names (see the `check_data()` method of the `plotter_cls` attribute for details)
- `dims (list of dictionaries)` – The dimensions of the arrays. It will be enhanced by the default dimensions of this plot method
- `is_unstructured (bool or list of bool)` – True if the corresponding array is unstructured.

**Returns**

- `list of bool or None` – True, if everything is okay, False in case of a serious error, None if it is intermediate. Each object in this list corresponds to one in the given `name`
- `list of str` – The message giving more information on the reason. Each object in this list corresponds to one in the given `name`

### `docs(*args, **kwargs)`
Method to print the full documentations of the formatoptions

**Parameters**

- `keys (list of str or None)` – If None, the all formatoptions of the given class are used. Group names from the `psyplot.plotter.groups` mapping are replaced by the formatoptions
- `indent (int)` – The indentation of the table
- `grouped (bool, optional)` – If True, the formatoptions are grouped corresponding to the Formatoption.groupname attribute

**Other Parameters**

- `func (function or None)` – The function the is used for returning (by default it is printed via the `print()` function or (when using the gui) in the help explorer). The given function must take a string as argument
- `include_links (bool or None, optional)` – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the `psyplot.plotter.Plotter.include_links` attribute.
• ‘‘*args,**kwargs‘‘ – They are passed to the difflib.get_close_matches() function (i.e. \( n \) to increase the number of returned similar keys and \( \text{cutoff} \) to change the sensibility)

**Returns** None if \( \text{func} \) is the print function, otherwise anything else

**Return type** results of \( \text{func} \)

See also:

keys (), docs ()

keys (**args,**kwargs)

Classmethod to return a nice looking table with the given formatoptions

**Parameters**

• **keys** (list of str or None) – If None, the all formatoptions of the given class are used. Group names from the psyplot.plotter.groups mapping are replaced by the formatoptions

• **indent** (int) – The indentation of the table

• **grouped** (bool, optional) – If True, the formatoptions are grouped corresponding to the Formatoption.groupname attribute

**Other Parameters**

• **func** (function or None) – The function the is used for returning (by default it is printed via the print() function or (when using the gui) in the help explorer). The given function must take a string as argument

• **include_links** (bool or None, optional) – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the psyplot.plotter.Plotter.include_links attribute.

• ‘‘*args,**kwargs‘‘ – They are passed to the difflib.get_close_matches() function (i.e. \( n \) to increase the number of returned similar keys and \( \text{cutoff} \) to change the sensibility)

**Returns** None if \( \text{func} \) is the print function, otherwise anything else

**Return type** results of \( \text{func} \)

See also:

summaries (), docs ()

plotter_cls

The plotter class

print_func

The function that is used to return a formatoption

By default the print() function is used (i.e. it is printed to the terminal)

sumaries (**args,**kwargs)

Method to print the summaries of the formatoptions

**Parameters**

• **keys** (list of str or None) – If None, the all formatoptions of the given class are used. Group names from the psyplot.plotter.groups mapping are replaced by the formatoptions

• **indent** (int) – The indentation of the table
• grouped (bool, optional) – If True, the formatoptions are grouped corresponding to the Formatoption.groupname attribute

Other Parameters
• func (function or None) – The function the is used for returning (by default it is printed via the print() function or (when using the gui) in the help explorer). The given function must take a string as argument
• include_links (bool or None, optional) – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the psyplot.plotter.Plotter.include_links attribute.
• **args, **kwargs – They are passed to the difflib.get_close_matches() function (i.e. n to increase the number of returned similar keys and cutoff to change the sensibility)

Returns None if func is the print function, otherwise anything else

Return type results of func

See also:
keys(), docs()

class psyplot.project.Project (*args, **kwargs)
Bases: psyplot.data.ArrayList

A manager of multiple interactive data projects

Parameters
• iterable (iterable) – The iterable (e.g. another list) defining this list
• attrs (dict-like or iterable, optional) – Global attributes of this list
• auto_update (bool) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the update() method or not. See also the no_auto_update attribute. If None, the value from the 'lists.auto_update' key in the psyplot.rcParams dictionary is used.
• new_name (bool or str) – If False, and the arr_name attribute of the new array is already in the list, a ValueError is raised. If True and the arr_name attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, new_name is set to 'arr{0}'. If not True, this will be used for renaming (if the array name of arr is in use or not). ' {0} ' is replaced by a counter
• main (Project) – The main project this subproject belongs to (or None if this project is the main project)
• num (int) – The number of the project

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Append a new array to the list</td>
</tr>
<tr>
<td>close</td>
<td>Close this project instance</td>
</tr>
<tr>
<td>disable</td>
<td>Disables the plotters in this list</td>
</tr>
<tr>
<td>docs</td>
<td>Show the available formatoptions in this project and their full docu</td>
</tr>
<tr>
<td>enable</td>
<td></td>
</tr>
</tbody>
</table>
### Table 78 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>export(output[, tight, concat, close_pdf, ...])</code></td>
<td>Exports the figures of the project to one or more image files</td>
</tr>
<tr>
<td><code>extend(*args, **kwargs)</code></td>
<td>Add further arrays from an iterable to this list</td>
</tr>
<tr>
<td><code>from_dataset(*args, **kwargs)</code></td>
<td>Construct an ArrayList instance from an existing base dataset</td>
</tr>
<tr>
<td><code>joined_attrs([delimiter, enhanced, ...])</code></td>
<td>Join the attributes of the arrays in this project</td>
</tr>
<tr>
<td><code>keys(*args, **kwargs)</code></td>
<td>Show the available formatoptions in this project</td>
</tr>
<tr>
<td><code>load_project(fname[, auto_update, ...])</code></td>
<td>Load a project from a file or dict</td>
</tr>
<tr>
<td><code>new([num])</code></td>
<td>Create a new main project</td>
</tr>
<tr>
<td><code>save_project([fname, pwd, pack])</code></td>
<td>Save this project to a file</td>
</tr>
<tr>
<td><code>scp(project)</code></td>
<td>Set the current project</td>
</tr>
<tr>
<td><code>share([base, keys, by])</code></td>
<td>Share the formatoptions of one plotter with all the others</td>
</tr>
<tr>
<td><code>show()</code></td>
<td>Shows all open figures</td>
</tr>
<tr>
<td><code>summaries(*args, **kwargs)</code></td>
<td>Show the available formatoptions and their summaries in this project</td>
</tr>
<tr>
<td><code>unshare(**kwargs)</code></td>
<td>Unshare the formatoptions of all the plotters in this instance</td>
</tr>
</tbody>
</table>

#### Attributes

- `arr_names` Names of the arrays (!not of the variables!) in this list
- `axes` A mapping from axes to data objects with the plotter in this axes
- `barplot` List of data arrays that are plotted by `psy_simple.plotters.BarPlotter` plotters
- `block_signals([value])` Wrapper around a boolean defining an `__enter__` and `__exit__` method
- `combined` List of data arrays that are plotted by `psy_simple.plotters.CombinedSimplePlotter` plotters
- `datasets` A mapping from dataset numbers to datasets in this list
- `density` List of data arrays that are plotted by `psy_simple.plotters.DensityPlotter` plotters
- `dsnames` The set of dataset names in this instance
- `dsnames_map` A dictionary from the dataset numbers in this list to their
- `figs` A mapping from figures to data objects with the plotter in this
- `fldmean` List of data arrays that are plotted by `psy_simple.plottersFldmeanPlotter` plotters
- `is_cmp` Boolean that is True if the project is the current main project
- `is_csp` Boolean that is True if the project is the current sub-project
- `is_main` `bool`. True if this `Project` is a main project

Continued on next page
Table 79 – continued from previous page

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lineplot</td>
<td>List of data arrays that are plotted by psy_simple.plotters.LinePlotter plotters</td>
</tr>
<tr>
<td>logger</td>
<td>logging.Logger of this instance</td>
</tr>
<tr>
<td>main</td>
<td>Project. The main project of this subproject</td>
</tr>
<tr>
<td>mapcombined</td>
<td>List of data arrays that are plotted by psy_maps.plotters.CombinedPlotter plotters</td>
</tr>
<tr>
<td>mapplot</td>
<td>List of data arrays that are plotted by psy_maps.plotters.FieldPlotter plotters</td>
</tr>
<tr>
<td>maps</td>
<td>List of data arrays that are plotted by psy_maps.plotters.MapPlotter plotters</td>
</tr>
<tr>
<td>mapvector</td>
<td>List of data arrays that are plotted by psy_maps.plotters.VectorPlotter plotters</td>
</tr>
<tr>
<td>oncpchange</td>
<td>Signal to be emitted when the current main and/or subproject changes</td>
</tr>
<tr>
<td>plot</td>
<td>Plotting instance of this Project.</td>
</tr>
<tr>
<td>plot2d</td>
<td>List of data arrays that are plotted by psy_simple.plotters.Simple2DPlotter plotters</td>
</tr>
<tr>
<td>plotters</td>
<td>A list of all the plotters in this instance</td>
</tr>
<tr>
<td>simple</td>
<td>List of data arrays that are plotted by psy_simple.plotters.SimplePlotterBase plotters</td>
</tr>
<tr>
<td>vector</td>
<td>List of data arrays that are plotted by psy_simple.plotters.SimpleVectorPlotter plotters</td>
</tr>
<tr>
<td>violinplot</td>
<td>List of data arrays that are plotted by psy_simple.plotters.ViolinPlotter plotters</td>
</tr>
<tr>
<td>with_plotter</td>
<td>The arrays in this instance that are visualized with a plotter</td>
</tr>
</tbody>
</table>

**append(***args, **kwargs)**

Append a new array to the list

**Parameters**

- **value** (InteractiveBase) – The data object to append to this list
- **new_name** (bool or str) – If False, and the arr_name attribute of the new array is already in the list, a ValueError is raised. If True and the arr_name attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, new_name is set to ‘arr[0]’. If not True, this will be used for renaming (if the array name of arr is in use or not). '{0}' is replaced by a counter

**Raises**

- **ValueError** – If it was impossible to find a name that isn’t already in the list
- **ValueError** – If new_name is False and the array is already in the list

**See also:**

- list.append(), extend(), rename()

**arr_names**

Names of the arrays (!not of the variables!) in this list
This attribute can be set with an iterable of unique names to change the array names of the data objects in this list.

**axes**

A mapping from axes to data objects with the plotter in this axes

**barplot**

List of data arrays that are plotted by `psy_simple.plotters.BarPlotter` plotters

**block_signals** *(value=None)*

Wrapper around a boolean defining an `__enter__` and `__exit__` method

**Notes**

If you want to use this class as an instance property, rather use the `_temp_bool_prop()` because this class as a descriptor is ment to be a class descriptor

**close** *(figs=True, data=False, ds=False, remove_only=False)*

Close this project instance

**Parameters**

- **figs** *(bool)* – Close the figures
- **data** *(bool)* – delete the arrays from the (main) project
- **ds** *(bool)* – If True, close the dataset as well
- **remove_only** *(bool)* – If True and `figs` is True, the figures are not closed but the plotters are removed

**combined**

List of data arrays that are plotted by `psy_simple.plotters.CombinedSimplePlotter` plotters

**datasets**

A mapping from dataset numbers to datasets in this list

**density**

List of data arrays that are plotted by `psy_simple.plotters.DensityPlotter` plotters

**disable**

Disables the plotters in this list

**docs** *(*args, **kwargs)*

Show the available formatoptions in this project and their full docu

**Parameters**

- **keys** *(list of str or None)* – If None, the all formatoptions of the given class are used. Group names from the `psyplot.plotter.groups` mapping are replaced by the formatoptions
- **indent** *(int)* – The indentation of the table
- **grouped** *(bool, optional)* – If True, the formatoptions are grouped corresponding to the Formatoption.groupname attribute

**Other Parameters**

- **func** *(function or None)* – The function the is used for returning (by default it is printed via the `print()` function or (when using the gui) in the help explorer). The given function must take a string as argument
• **include_links** (bool or None, optional) – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the `psyplot.plotter.Plotter.include_links` attribute.

• `**args**, **kwargs** – They are passed to the `difflib.get_close_matches()` function (i.e. `n` to increase the number of returned similar keys and `cutoff` to change the sensibility)

**Returns** None if `func` is the print function, otherwise anything else

**Return type** results of `func`

dsnames
The set of dataset names in this instance
dsnames_map
A dictionary from the dataset numbers in this list to their filenames

enable()

export (output, tight=False, concat=True, close_pdf=None, use_time=False, **kwargs)
Exports the figures of the project to one or more image files

**Parameters**

• **output** (str, iterable or `matplotlib.backends.backend_pdf.PdfPages`) – if string or list of strings, those define the names of the output files. Otherwise you may provide an instance of `matplotlib.backends.backend_pdf.PdfPages` to save the figures in it. If string (or iterable of strings), attribute names in the xarray.DataArray.attrs attribute as well as index dimensions are replaced by the respective value (see examples below). Furthermore a single format string without key (e.g. `%i`, `%s`, `%d`, etc.) is replaced by a counter.

• **tight** (bool) – If True, it is tried to figure out the tight bbox of the figure (same as `bbox_inches='tight'`)

• **concat** (bool) – if True and the output format is `pdf`, all figures are concatenated into one single pdf

• **close_pdf** (bool or None) – If True and the figures are concatenated into one single pdf, the resulting pdf instance is closed. If False it remains open. If None and `output` is a string, it is the same as `close_pdf=True`, if None and `output` is neither a string nor an iterable, it is the same as `close_pdf=False`

• **use_time** (bool) – If True, formatting strings for the `datetime.datetime.strftime()` are expected to be found in `output` (e.g. '%m', '%Y', etc.). If so, other formatting strings must be escaped by double '%' (e.g. '%%i' instead of ('%i'))

• ****kwargs** – Any valid keyword for the `matplotlib.pyplot.savefig()` function

**Returns** a `PdfPages` instance if `output` is a string and `close_pdf` is False, otherwise None

**Return type** `matplotlib.backends.backend_pdf.PdfPages` or `None`

**Examples**

Simply save all figures into one single pdf:

```python
>>> p = psy.gcp()
>>> p.export('my_plots.pdf')
```
Save all figures into separate pngs with increasing numbers (e.g. 'my_plots_1.png'):

```python
>>> p.export('my_plots_%i.png')
```

Save all figures into separate pngs with the name of the variables shown in each figure (e.g. 'my_plots_t2m.png'):

```python
>>> p.export('my_plots_${name}.png')
```

Save all figures into separate pngs with the name of the variables shown in each figure and with increasing numbers (e.g. 'my_plots_1_t2m.png'):

```python
>>> p.export('my_plots_${i}_${name}.png')
```

Specify the names for each figure directly via a list:

```python
>>> p.export(['my_plots1.pdf', 'my_plots2.pdf'])
```

```python
extend(*args, **kwargs)
```

Add further arrays from an iterable to this list.

Parameters

- **iterable** – Any iterable that contains InteractiveBase instances
- **new_name** (**bool** or **str**) – If False, and the `arr_name` attribute of the new array is already in the list, a ValueError is raised. If True and the `arr_name` attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, `new_name` is set to ‘arr[0]’. If not True, this will be used for renaming (if the array name of `arr` is in use or not). ‘{0}’ is replaced by a counter

Raises

- **ValueError** – If it was impossible to find a name that isn’t already in the list
- **ValueError** – If `new_name` is False and the array is already in the list

See also:

- `list.extend()`, `append()`, `rename()`

```python
figs
```

A mapping from figures to data objects with the plotter in this figure

```python
fldmean
```

List of data arrays that are plotted by psy_simple.plotters.FldmeanPlotter plotters

```python
classmethod from_dataset(*args, **kwargs)
```

Construct an ArrayList instance from an existing base dataset

Parameters

- **base** (**xarray.Dataset**) – Dataset instance that is used as reference
- **method** (**{'isel', ‘None’, ‘nearest’, ..}**) – Selection method of the xarray.Dataset to be used for setting the variables from the informations in `dims`. If `method` is ‘isel’, the `xarray.Dataset.isel()` method is used. Otherwise it sets the `method` parameter for the `xarray.Dataset.sel()` method.
- **auto_update** (**bool**) – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not.
See also the \texttt{no\_auto\_update} attribute. If None, the value from the 'lists.\texttt{auto\_update}' key in the \texttt{psyplot.rcParams} dictionary is used.

- \texttt{prefer\_list} (\texttt{bool}) – If True and multiple variable names per array are found, the \texttt{InteractiveList} class is used. Otherwise the arrays are put together into one \texttt{InteractiveArray}.

- \texttt{default\_slice} (\texttt{indexer}) – Index (e.g. 0 if \texttt{method} is ‘isel’) that shall be used for dimensions not covered by \texttt{dims} and \texttt{furtherdims}. If None, the whole slice will be used.

- \texttt{decoder} (\texttt{CFDecoder}) – The decoder that shall be used to decoder the \texttt{base} dataset

- \texttt{squeeze} (\texttt{bool}, optional) – Default True. If True, and the created arrays have a an axes with length 1, it is removed from the dimension list (e.g. an array with shape (3, 4, 1, 5) will be squeezed to shape (3, 4, 5))

- \texttt{attrs} (\texttt{dict}, optional) – Meta attributes that shall be assigned to the selected data arrays (additional to those stored in the \texttt{base} dataset)

- \texttt{load} (\texttt{bool} or \texttt{dict}) – If True, load the data from the dataset using the \texttt{xarray.\texttt{DataArray.load()}} method. If \texttt{dict}, those will be given to the above mentioned load method

- \texttt{main} (\texttt{Project}) – The main project that this project corresponds to

\textbf{Other Parameters}

- \texttt{arr\_names} (\texttt{string, list of strings or dictionary}) – Set the unique array names of the resulting arrays and (optionally) dimensions.
  - if string: same as list of strings (see below). Strings may include \{0\} which will be replaced by a counter.
  - list of strings: those will be used for the array names. The final number of dictionaries in the return depend in this case on the \texttt{dims} and \texttt{**furtherdims}
  - dictionary: Then nothing happens and an \texttt{OrderedDict} version of \texttt{arr\_names} is returned.

- \texttt{sort} (\texttt{list of strings}) – This parameter defines how the dictionaries are ordered. It has no effect if \texttt{arr\_names} is a dictionary (use a \texttt{OrderedDict} for that). It can be a list of dimension strings matching to the dimensions in \texttt{dims} for the variable.

- \texttt{dims} (\texttt{dict}) – Keys must be variable names of dimensions (e.g. time, level, lat or lon) or ‘name’ for the variable name you want to choose. Values must be values of that dimension or iterables of the values (e.g. lists). Note that strings will be put into a list. For example \texttt{dims = \{‘name’: ‘t2m’, ‘time’: 0\}} will result in one plot for the first time step, whereas \texttt{dims = \{‘name’: ‘t2m’, ‘time’: [0, 1]\}} will result in two plots, one for the first (time == 0) and one for the second (time == 1) time step.

- \texttt{**kwargs} – The same as \texttt{dims} (those will update what is specified in \texttt{dims})

\textbf{Returns} The newly created project instance

\textbf{Return type} \texttt{Project}

\texttt{is\_cmp}

- Boolean that is True if the project is the current main project

\texttt{is\_csp}

- Boolean that is True if the project is the current subproject

\texttt{is\_main}

- \texttt{bool}. True if this \texttt{Project} is a main project
joined_attrs (delimiter=',', enhanced=True, plot_data=False, keep_all=True)

Join the attributes of the arrays in this project

Parameters

- delimiter (str) – The string that shall be used as the delimiter in case that there are multiple values for one attribute in the arrays. If None, they will be returned as sets
- enhanced (bool) – If True, the psyplot.plotter.Plotter.get_enhanced_attrs() method is used, otherwise the xarray.DataArray.attrs attribute is used.
- plot_data (bool) – If True, use the psyplot.plotter.Plotter.plot_data attribute of the plotters rather than the raw data in this project
- keep_all (bool) – If True, all formatoptions are kept. Otherwise only the intersection

Returns A mapping from the attribute to the joined attributes which are either strings or (if there is only one attribute value), the data type of the corresponding value

Return type  dict

keys (*args, **kwargs)

Show the available formatoptions in this project

Parameters

- keys (list of str or None) – If None, the all formatoptions of the given class are used. Group names from the psyplot.plotter.groups mapping are replaced by the formatoptions
- indent (int) – The indentation of the table
- grouped (bool, optional) – If True, the formatoptions are grouped corresponding to the Formatoption.groupname attribute

Other Parameters

- func (function or None) – The function the is used for returning (by default it is printed via the print() function or (when using the gui) in the help explorer). The given function must take a string as argument
- include_links (bool or None, optional) – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the psyplot.plotter.Plotter.include_links attribute.
- """*args,**kwargs""" – They are passed to the difflib.get_close_matches() function (i.e. n to increase the number of returned similar keys and cutoff to change the sensibility)

Returns None if func is the print function, otherwise anything else

Return type  results of func

lineplot

List of data arrays that are plotted by psy_simple.plotters.LinePlotter plotters

classmethod load_project (fname, auto_update=None, make_plot=True, draw=False, alternative_axes=None, main=False, encoding=None, enable_post=False, new_fig=True, clear=None, **kwargs)

Load a project from a file or dict

This classmethod allows to load a project that has been stored using the save_project() method and reads all the data and creates the figures.
Since the data is stored in external files when saving a project, make sure that the data is accessible under the relative paths as stored in the file `fname` or from the current working directory if `fname` is a dictionary. Alternatively use the `alternative_paths` parameter or the `pwd` parameter.

Parameters

- `fname` *(str or dict)* – The string might be the path to a file created with the `save_project()` method, or it might be a dictionary from this method.

- `auto_update` *(bool)* – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not. See also the `no_auto_update` attribute. If None, the value from the `'lists.auto_update'` key in the `psyplot.rcParams` dictionary is used.

- `make_plot` *(bool)* – If True, the data is plotted at the end. Otherwise you have to call the `psyplot.plotter.Plotter.initialize_plot()` method or the `psyplot.plotter.Plotter.reinit()` method by yourself.

- `draw` *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the `'auto_draw'` parameter in the `psyplot.rcParams` dictionary.

- `alternative_axes` *(dict, None or list)* – alternative axes instances to use
  - If it is None, the axes and figures from the saving point will be reproduced.
  - a dictionary should map from array names in the created project to matplotlib axes instances
  - a list should contain axes instances that will be used for iteration

- `main` *(bool, optional)* – If True, a new main project is created and returned. Otherwise (by default default) the data is added to the current main project.

- `encoding` *(str)* – The encoding to use for loading the project. If None, it is automatically determined by pickle. Note: Set this to `'latin1'` if using a project created with python2 on python3.

- `enable_post` *(bool)* – If True, the `post` formatoption is enabled and post processing scripts are allowed. Do only set this parameter to True if you know you can trust the information in `fname`.

- `new_fig` *(bool)* – If True (default) and `alternative_axes` is None, new figures are created if the figure already exists.

- `clear` *(bool)* – If True, axes are cleared before making the plot. This is only necessary if the `ax` keyword consists of subplots with projection that differs from the one that is needed.

- `pwd` *(str or None, optional)* – Path to the working directory from where the data can be imported. If None and `fname` is the path to a file, `pwd` is set to the directory of this file. Otherwise the current working directory is used.

- `alternative_paths` *(dict or list or str)* – A mapping from original filenames as used in `d` to filenames that shall be used instead. If `alternative_paths` is not None, datasets must be None. Paths must be accessible from the current working directory. If `alternative_paths` is a list (or any other iterable) is provided, the file names will be replaced as they appear in `d` (note that this is very unsafe if `d` is not and OrderedDict).

- `datasets` *(dict or list or None)* – A mapping from original filenames in `d` to the instances of `xarray.Dataset` to use. If it is an iterable, the same holds as for the `alternative_paths` parameter.
• **ignore_keys** (*list of str*) – Keys specified in this list are ignored and not seen as array information (note that *attrs* are used anyway)

• **only** (*string, list or callable*) – Can be one of the following three things:
  – a string that represents a pattern to match the array names that shall be included
  – a list of array names to include
  – a callable with two arguments, a string and a dict such as

```python
def filter_func(arr_name: str, info: dict): -> bool
    """Filter the array names

    This function should return True if the array shall be included, else False
    Parameters
    ----------
    arr_name: str
        The array name (i.e. the `arr_name` attribute)
    info: dict
        The dictionary with the array informations. Common keys are `name` that points to the variable name
        and `dims` that points to the dimensions and `fname` that points to the file name
    """
    return True or False
```

The function should return True if the array shall be included, else False. This function will also be given to subsequents instances of `InteractiveList` objects that are contained in the returned value

• **chname** (*dict*) – A mapping from variable names in the project to variable names that should be used instead

**Other Parameters**

• **d** (*dict*) – The dictionary holding the data

• **alternative_paths** (*dict or list or str*) – A mapping from original filenames as used in *d* to filenames that shall be used instead. If *alternative_paths* is not None, datasets must be None. Paths must be accessible from the current working directory. If *alternative_paths* is a list (or any other iterable) is provided, the file names will be replaced as they appear in *d* (note that this is very unsafe if *d* is not an OrderedDict)

• **datasets** (*dict or list or None*) – A mapping from original filenames in *d* to the instances of `xarray.Dataset` to use. If it is an iterable, the same holds as for the *alternative_paths* parameter

• **pwd** (*str*) – Path to the working directory from where the data can be imported. If None, use the current working directory.

• **ignore_keys** (*list of str*) – Keys specified in this list are ignored and not seen as array information (note that *attrs* are used anyway)

• **only** (*string, list or callable*) – Can be one of the following three things:
  – a string that represents a pattern to match the array names that shall be included
  – a list of array names to include
– a callable with two arguments, a string and a dict such as

```python
def filter_func(arr_name: str, info: dict): -> bool
    '''
    Filter the array names
    This function should return True if the array shall be included, else False
    Parameters
    ----------
    arr_name: str
        The array name (i.e. the `arr_name` attribute)
    info: dict
        The dictionary with the array informations. Common keys are `name` that points to the variable name and `dims` that points to the dimensions and `fname` that points to the file name
    '''
    return True or False
```

The function should return `True` if the array shall be included, else `False`. This function will also be given to subsequents instances of `InteractiveList` objects that are contained in the returned value

- **chname (dict)** - A mapping from variable names in the project to variable names that should be used instead

**Returns** The project in state of the saving point

**Return type** `Project`

- `logger` - `logging.Logger` of this instance

- `main` - `Project`. The main project of this subproject

- `mapcombined` - List of data arrays that are plotted by `psy_maps.plotters.CombinedPlotter` plotters

- `mapplot` - List of data arrays that are plotted by `psy_maps.plotters.FieldPlotter` plotters

- `maps` - List of data arrays that are plotted by `psy_maps.plotters.MapPlotter` plotters

- `mapvector` - List of data arrays that are plotted by `psy_maps.plotters.VectorPlotter` plotters

**classmethod new** (`num=None, *args, **kwargs`) Create a new main project

**Parameters**

- `num (int)` - The number of the project
- `iterable (iterable)` - The iterable (e.g. another list) defining this list
- `attrs (dict-like or iterable, optional)` - Global attributes of this list
- `auto_update (bool)` - Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not.
See also the no_auto_update attribute. If None, the value from the 'lists. auto_update' key in the psyplot.rcParams dictionary is used.

- **new_name** *(bool or str)* – If False, and the arr_name attribute of the new array is already in the list, a ValueError is raised. If True and the arr_name attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, new_name is set to ‘arr{0}’. If not True, this will be used for renaming (if the array name of arr is in use or not). '{0}' is replaced by a counter

- **main** *(Project)* – The main project this subproject belongs to (or None if this project is the main project)

**Returns** The with the given num (if it does not already exist, it is created)

**Return type** *Project*

See also:

- **scp()** Sets the current project
- **gcp()** Returns the current project

**oncpchange**

signal to be emitted when the current main and/or subproject changes

**plot**

Plotting instance of this *Project*. See the *ProjectPlotter* class for method documentations

**plot2d**

List of data arrays that are plotted by *psy_simple.plotters.Simple2DPlotter* plotter

**plotters**

A list of all the plotters in this instance

**save_project** *(fname=None, pwd=None, pack=False, **kwargs)*

Save this project to a file

**Parameters**

- **fname** *(str or None)* – If None, the dictionary will be returned. Otherwise the necessary information to load this project via the *load()* method is saved to *fname* using the *pickle* module

- **pwd** *(str or None, optional)* – Path to the working directory from where the data can be imported. If None and *fname* is the path to a file, *pwd* is set to the directory of this file. Otherwise the current working directory is used.

- **pack** *(bool)* – If True, all datasets are packed into the folder of *fname* and will be used if the data is loaded

- **dump** *(bool)* – If True and the dataset has not been dumped so far, it is dumped to a temporary file or the one generated by *paths* is used. If it is False or both, *dump* and *paths* are None, no data will be stored. If it is None and *paths* is not None, *dump* is set to True.

- **paths** *(iterable or True)* – An iterator over filenames to use if a dataset has no filename. If paths is True, an iterator over temporary files will be created without raising a warning

- **attrs** *(bool, optional)* – If True (default), the *ArrayList.attrs* and *xarray.DataArray.attrs* attributes are included in the returning dictionary

- **standardize_dims** *(bool, optional)* – If True (default), the real dimension names in the dataset are replaced by x, y, z and t to be more general.
• **use_rel_paths** *(bool, optional)* – If True (default), paths relative to the current working directory are used. Otherwise absolute paths to `pwd` are used.

• **ds_description** *("all" or set of {"fname", "ds", "num", "arr", "store"})* – Keys to describe the datasets of the arrays. If all, all keys are used. The key descriptions are:
  
  - **fname** the file name is inserted in the "fname" key
  - **store** the data store class and module is inserted in the "store" key
  - **ds** the dataset is inserted in the "ds" key
  - **num** The unique number assigned to the dataset is inserted in the "num" key
  - **arr** The array itself is inserted in the "arr" key

• **full_ds** *(bool)* – If True and "ds" is in ds_description, the entire dataset is included. Otherwise, only the DataArray converted to a dataset is included.

**Notes**

You can also store the entire data in the pickled file by setting `ds_description={'ds'}`.

**classmethod scp**(project)

Set the current project.

**Parameters**

- **project** *(Project or None)* – The project to set. If it is None, the current subproject is set to empty. If it is a sub project (see: attr:`Project.is_main`), the current subproject is set to this project. Otherwise it replaces the current main project.

**See also:**

- **scp()** The global version for setting the current project
- **gcp()** Returns the current project
- **project()** Creates a new project

**share**(base=None, keys=None, by=None, **kwargs)

Share the formatoptions of one plotter with all the others.

This method shares specified formatoptions from `base` with all the plotters in this instance.

**Parameters**

- **base** *(None, Plotter, xarray.DataArray, InteractiveList, or list of them)* – The source of the plotter that shares its formatoptions with the others. It can be None (then the first instance in this project is used), a Plotter or any data object with a `psy` attribute. If `by` is not None, then it is expected that `base` is a list of data objects for each figure/axes.

- **keys** *(string or iterable of strings)* – The formatoptions to share, or group names of formatoptions to share all formatoptions of that group (see the `fmt_groups` property). If None, all formatoptions of this plotter are unshared.

- **by** *(\{'fig', 'figure', 'ax', 'axes'\})* – Share the formatoptions only with the others on the same 'figure' or the same 'axes'. In this case, base must either be None or a list of the types specified for `base`
psyplot Documentation, Release 1.2.1

• **draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the `auto_draw` parameter in the psyplot.rcparams dictionary.

• **auto_update** *(bool)* – Boolean determining whether or not the start_update() method is called at the end. This parameter has no effect if the no_auto_update attribute is set to True.

See also:

psyplot.plotter.share()

**static show()**

Shows all open figures

**simple**

List of data arrays that are plotted by psy_simple.plotters.SimplePlotterBase plots

**summaries** *(*args, **kwargs)*

Show the available formatoptions and their summaries in this project

Parameters

• **keys** *(list of str or None)* – If None, the all formatoptions of the given class are used. Group names from the psyplot.plotter.groups mapping are replaced by the formatoptions.

• **indent** *(int)* – The indentation of the table

• **grouped** *(bool, optional)* – If True, the formatoptions are grouped corresponding to the Formatoption.groupname attribute

Other Parameters

• **func** *(function or None)* – The function the is used for returning (by default it is printed via the print() function or (when using the gui) in the help explorer). The given function must take a string as argument

• **include_links** *(bool or None, optional)* – Default False. If True, links (in restructured formats) are included in the description. If None, the behaviour is determined by the psyplot.plotter.Plotter.include_links attribute.

• **"*args,**kwargs"** – They are passed to the difflib.get_close_matches() function (i.e. n to increase the number of returned similar keys and cutoff to change the sensibility)

Returns

None if func is the print function, otherwise anything else

Return type

results of func

**unshare(** **kwargs)**

Unshare the formatoptions of all the plotters in this instance

This method uses the psyplot.plotter.Plotter.unshare_me() method to release the specified formatoptions in keys.

Parameters

• **keys** *(string or iterable of strings)* – The formatoptions to unshare, or group names of formatoptions to unshare all formatoptions of that group (see the fmt_groups property). If None, all formatoptions of this plotter are unshared.
**draw** *(bool or None)* – Boolean to control whether the figure of this array shall be drawn at the end. If None, it defaults to the ‘auto_draw’ parameter in the psyplot. rcParams dictionary

**auto_update** *(bool)* – Boolean determining whether or not the start_update() method is called at the end. This parameter has no effect if the no_auto_update attribute is set to True.

See also:

```
psyplot.plotter.Plotter.unshare(), psyplot.plotter.Plotter.unshare_me()
```

---

vector

List of data arrays that are plotted by `psy_simple.plotters.SimpleVectorPlotter` plotters

violinplot

List of data arrays that are plotted by `psy_simple.plotters.ViolinPlotter` plotters

---

**class** psyplot.project.ProjectPlotter *(project=None)*

Bases: object

Plotting methods of the `psyplot.project.Project` class

**Attributes**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>barplot(*args, *<em>kwargs)</em></td>
<td>Make a bar plot of one-dimensional data</td>
</tr>
<tr>
<td>combined(*args, *<em>kwargs)</em></td>
<td>Plot a 2D scalar field with an overlying vector field</td>
</tr>
<tr>
<td>density(*args, *<em>kwargs)</em></td>
<td>Make a density plot of point data</td>
</tr>
<tr>
<td>fldmean(*args, *<em>kwargs)</em></td>
<td>Calculate and plot the mean over x- and y- dimensions</td>
</tr>
<tr>
<td>lineplot(*args, *<em>kwargs)</em></td>
<td>Make a line plot of one-dimensional data</td>
</tr>
<tr>
<td>mapcombined(*args, *<em>kwargs)</em></td>
<td>Plot a 2D scalar field with an overlying vector field on a map</td>
</tr>
<tr>
<td>mapplot(*args, *<em>kwargs)</em></td>
<td>Plot a 2D scalar field on a map</td>
</tr>
<tr>
<td>mapvector(*args, *<em>kwargs)</em></td>
<td>Plot a 2D vector field on a map</td>
</tr>
<tr>
<td>plot2d(*args, *<em>kwargs)</em></td>
<td>Make a simple plot of a 2D scalar field</td>
</tr>
<tr>
<td>project.vector(*args, *<em>kwargs)</em></td>
<td>Make a simple plot of a 2D vector field</td>
</tr>
<tr>
<td>violinplot(*args, *<em>kwargs)</em></td>
<td>Make a violin plot of your data</td>
</tr>
</tbody>
</table>

**Methods**

**show_plot_methods()**

Print the plotmethods of this instance

**barplot(*args, **kwargs)*

Make a bar plot of one-dimensional data

This plotting method adds data arrays and plots them via `psy_simple.plotters.BarPlotter` plotters

To plot data from a netCDF file type:

```
>>> psy.plot.barplot(filename, name=['my_variable'], ...)
```

Possible format options are

---
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy
# show the keys corresponding to a group or multiple formatoptions
>>> psy.plot.barplot.keys('labels')
# show the summaries of a group of formatoptions or of a formatoption
>>> psy.plot.barplot.summaries('title')
# show the full documentation
>>> psy.plot.barplot.docs('plot')
# or access the documentation via the attribute
>>> psy.plot.barplot.plot
```

**combined**(*args, **kwargs)

Plot a 2D scalar field with an overlying vector field

This plotting method adds data arrays and plots them via psy_simple.plotters. CombinedSimplePlotter plotters

To plot data from a netCDF file type:

```python
>>> psy.plot.combined(filename, name=['my_variable', ['u_var,', 'v_var']], ...)
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>alpha</th>
<th>axiscolor</th>
<th>categorical</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>coord</td>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
</tr>
<tr>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskless</td>
<td></td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
<td>sym_lims</td>
</tr>
<tr>
<td>text</td>
<td>ticks</td>
<td>tickweight</td>
<td>tight</td>
</tr>
<tr>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
</tr>
<tr>
<td>transpose</td>
<td>widths</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
</tr>
<tr>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
<td>yticklabels</td>
</tr>
<tr>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples

To explore the format options and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
>>> import psyplot.project as psy
# show the keys corresponding to a group or multiple
# formatoptions
>>> psy.plot.combined.keys('labels')
# show the summaries of a group of formatoptions or of a
# formatoption
>>> psy.plot.combined.summaries('title')
# show the full documentation
>>> psy.plot.combined.docs('plot')
# or access the documentation via the attribute
>>> psy.plot.combined.plot
```

density(*args, **kwargs)

Make a density plot of point data

This plotting method adds data arrays and plots them via `psy_simple.plotters`. DensityPlotter plotters

To plot data from a netCDF file type:

```python
>>> psy.plot.density(filename, name=['my_variable'], ...)
```

Possible format options are
Examples

To explore the formatoptions and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple formatoptions
>>> psy.plot.density.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> psy.plot.density.summaries('title')

# show the full documentation
>>> psy.plot.density.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.density.plot
```

### fldmean(*args, **kwargs)

Calculate and plot the mean over x- and y-dimensions

This plotting method adds data arrays and plots them via `psy_simple.plotters.FldmeanPlotter` plotters.

To plot data from a netCDF file type:

```python
>>> psy.plot.fldmean(filename, name=['my_variable'], ...)
```

Possible formatoptions are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>bins</th>
<th>bounds</th>
<th>cbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>clabelsize</td>
</tr>
<tr>
<td>clabelweight</td>
<td>cmap</td>
<td>coord</td>
<td>clabelprops</td>
</tr>
<tr>
<td>ctickprops</td>
<td>cticks</td>
<td>cticksize</td>
<td>ctickweight</td>
</tr>
<tr>
<td>datagrid</td>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
</tr>
<tr>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid</td>
</tr>
<tr>
<td>interp_bounds</td>
<td>labelprops</td>
<td>labelsize</td>
<td>labelweight</td>
</tr>
<tr>
<td>levels</td>
<td>maskbetween</td>
<td>maskgeq</td>
<td>maskgreater</td>
</tr>
<tr>
<td>maskleq</td>
<td>maskless</td>
<td>miss_color</td>
<td>normed</td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
<td>precision</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrange</td>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtckprops</td>
</tr>
<tr>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
<td>yrange</td>
</tr>
<tr>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
</tr>
</tbody>
</table>
Examples

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple formatoptions
>>> psy.plot.fldmean.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> psy.plot.fldmean.summaries('title')

# show the full documentation
>>> psy.plot.fldmean.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.fldmean.plot
```

**lineplot** (*args, **kwargs)

Make a line plot of one-dimensional data

This plotting method adds data arrays and plots them via psy_simple.plotters. LinePlotter plotters.

To plot data from a netCDF file type:

```python
>>> psy.plot.lineplot(filename, name=['my_variable'], ...)
```

Possible format options are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>color</th>
<th>coord</th>
<th>err_calc</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>erroralpha</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid</td>
<td>labelprops</td>
</tr>
<tr>
<td>labelsize</td>
<td>labelweight</td>
<td>legend</td>
<td>legendlabels</td>
</tr>
<tr>
<td>linewidth</td>
<td>marker</td>
<td>markersize</td>
<td>maskbetween</td>
</tr>
<tr>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
</tr>
<tr>
<td>mean</td>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
</tr>
<tr>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transpose</td>
<td>xlabel</td>
<td>xlim</td>
</tr>
<tr>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
</tr>
<tr>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
<td>yticklabels</td>
</tr>
<tr>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.12. API Reference
Examples
To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple format options
>>> psy.plot.lineplot.keys('labels')

# show the summaries of a group of format options or of a format option
>>> psy.plot.lineplot.summaries('title')

# show the full documentation
>>> psy.plot.lineplot.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.lineplot.plot
```

**mapcombined** (*args, **kwargs)
Plot a 2D scalar field with an overlying vector field on a map

This plotting method adds data arrays and plots them via psy_maps.plotters. CombinedPlotter plotters

To plot data from a netCDF file type:

```python
>>> psy.plot.mapcombined(filename, name=[['my_variable', ['u_var', 'v_var']], ...])
```

Possible format options are
Examples

To explore the format options and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple format options
>>> psy.plot.mapcombined.keys('labels')

# show the summaries of a group of format options or of a format option
>>> psy.plot.mapcombined.summaries('title')

# show the full documentation
>>> psy.plot.mapcombined.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.mapcombined.plot
```

`mapplot(*args, **kwargs)`

Plot a 2D scalar field on a map

This plotting method adds data arrays and plots them via `psy_maps.plotters.FieldPlotter` plotters.

To plot data from a netCDF file type:

```python
>>> psy.plot.mapplot(filename, name=['my_variable'], ...)
```

Possible format options are

```
<table>
<thead>
<tr>
<th>arrowsize</th>
<th>arrowstyle</th>
<th>bounds</th>
<th>cbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>clabelsize</td>
</tr>
<tr>
<td>cmap</td>
<td>color</td>
<td>cticklabels</td>
<td>ctickprops</td>
</tr>
<tr>
<td>ccticks</td>
<td>cticksize</td>
<td>ctickweight</td>
<td>datagrid</td>
</tr>
<tr>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid_color</td>
<td>grid_labels</td>
</tr>
<tr>
<td>grid_labelsize</td>
<td>grid_settings</td>
<td>interp_bounds</td>
<td>levels</td>
</tr>
<tr>
<td>linewidth</td>
<td>lonlatbox</td>
<td>lsm</td>
<td>map_extent</td>
</tr>
<tr>
<td>maskbetween</td>
<td>maskgeq</td>
<td>maskgreater</td>
<td>maskleq</td>
</tr>
<tr>
<td>maskless</td>
<td>miss_color</td>
<td>plot</td>
<td>post</td>
</tr>
<tr>
<td>post_timing</td>
<td>projection</td>
<td>stock_img</td>
<td>text</td>
</tr>
<tr>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
</tr>
<tr>
<td>titleweight</td>
<td>transform</td>
<td>vbounds</td>
<td>vbar</td>
</tr>
<tr>
<td>vcbarspacing</td>
<td>vclabel</td>
<td>vclabelprops</td>
<td>vclabelsize</td>
</tr>
<tr>
<td>vclabelweight</td>
<td>vcmap</td>
<td>vcticklabels</td>
<td>vctickprops</td>
</tr>
<tr>
<td>vcticks</td>
<td>vcticksize</td>
<td>vctickweight</td>
<td>vplot</td>
</tr>
<tr>
<td>xgrid</td>
<td>ygrid</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Examples

To explore the formatoptions and their documentations, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple formatoptions
>>> psy.plot.mapplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> psy.plot.mapplot.summaries('title')

# show the full documentation
>>> psy.plot.mapplot.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.mapplot.plot
```

**mapvector** (*args, **kwargs)

Plot a 2D vector field on a map

This plotting method adds data arrays and plots them via psy_maps.plotters.VectorPlotter plotters.

To plot data from a netCDF file type:

```python
>>> psy.plot.mapvector(filename, name=['u_var', 'v_var'], ...)
```

Possible formatoptions are
Examples

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple format options
>>> psy.plot.mapvector.keys('labels')

# show the summaries of a group of format options or of a format option
>>> psy.plot.mapvector.summaries('title')

# show the full documentation
>>> psy.plot.mapvector.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.mapvector.plot
```

`plot2d(*args, **kwargs)`

Make a simple plot of a 2D scalar field

This plotting method adds data arrays and plots them via psy_simple.plotters.

To plot data from a netCDF file type:

```python
>>> psy.plot.plot2d(filename, name=['my_variable'], ...)  
```

Possible format options are

<table>
<thead>
<tr>
<th>arrowsize</th>
<th>arrowstyle</th>
<th>bounds</th>
<th>cbar</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbarspacing</td>
<td>clabel</td>
<td>clabelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>clabelweight</td>
<td>clat</td>
<td>clip</td>
<td>clon</td>
</tr>
<tr>
<td>cmap</td>
<td>color</td>
<td>cticklabels</td>
<td>ctickprops</td>
</tr>
<tr>
<td>cticks</td>
<td>cticksizes</td>
<td>ctickweights</td>
<td>datagrid</td>
</tr>
<tr>
<td>density</td>
<td>extend</td>
<td>figtitle</td>
<td>figtitleprops</td>
</tr>
<tr>
<td>figtitlesize</td>
<td>figtitleweight</td>
<td>grid_color</td>
<td>grid_labels</td>
</tr>
<tr>
<td>grid_labelsize</td>
<td>grid_settings</td>
<td>linewidth</td>
<td>lonlatbox</td>
</tr>
<tr>
<td>lsm</td>
<td>map_extent</td>
<td>maskbetween</td>
<td>maskleq</td>
</tr>
<tr>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
<td>plot</td>
</tr>
<tr>
<td>post</td>
<td>post_timing</td>
<td>projection</td>
<td>stock_img</td>
</tr>
<tr>
<td>text</td>
<td>tight</td>
<td>title</td>
<td>titleprops</td>
</tr>
<tr>
<td>titlesize</td>
<td>titleweight</td>
<td>transform</td>
<td>xgrid</td>
</tr>
<tr>
<td>ygrid</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.12. API Reference
Examples

To explore the format options and their documents, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy
# show the keys corresponding to a group or multiple # formatoptions
>>> psy.plot.plot2d.keys('labels')
# show the summaries of a group of formatoptions or of a # formatoption
>>> psy.plot.plot2d.summaries('title')
# show the full documentation
>>> psy.plot.plot2d.docs('plot')
# or access the documentation via the attribute
>>> psy.plot.plot2d.plot
```

project

show_plot_methods()
Print the plot methods of this instance

vector(*args, **kwargs)
Make a simple plot of a 2D vector field

This plotting method adds data arrays and plots them via psy_simple.plotters. SimpleVectorPlotter plotters

To plot data from a netCDF file type:

```python
>>> psy.plot.vector(filename, name=['u_var', 'v_var'], ...)  
```

Possible format options are

<table>
<thead>
<tr>
<th>axiscolor</th>
<th>bounds</th>
<th>cbar</th>
<th>cbarspacing</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbar</td>
<td>clabelprops</td>
<td>clabelsize</td>
<td>clabelweight</td>
</tr>
<tr>
<td>cmap</td>
<td>cticklabels</td>
<td>ctickprops</td>
<td>cticks</td>
</tr>
<tr>
<td>cticksize</td>
<td>ctickweight</td>
<td>datagrid</td>
<td>extend</td>
</tr>
<tr>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
</tr>
<tr>
<td>grid</td>
<td>interp_bounds</td>
<td>labelprops</td>
<td>labelsize</td>
</tr>
<tr>
<td>labelweight</td>
<td>levels</td>
<td>maskbetween</td>
<td>maskgeq</td>
</tr>
<tr>
<td>maskgreater</td>
<td>maskleq</td>
<td>maskless</td>
<td>miss_color</td>
</tr>
<tr>
<td>plot</td>
<td>post</td>
<td>post_timing</td>
<td>sym_lims</td>
</tr>
<tr>
<td>text</td>
<td>ticksize</td>
<td>tickweight</td>
<td>tight</td>
</tr>
<tr>
<td>title</td>
<td>titleprops</td>
<td>titlesize</td>
<td>titleweight</td>
</tr>
<tr>
<td>transpose</td>
<td>xlabel</td>
<td>xlim</td>
<td>xrotation</td>
</tr>
<tr>
<td>xticklabels</td>
<td>xtickprops</td>
<td>xticks</td>
<td>ylabel</td>
</tr>
<tr>
<td>ylim</td>
<td>yrotation</td>
<td>yticklabels</td>
<td>ytickprops</td>
</tr>
<tr>
<td>yticks</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Examples

To explore the format options and their documentations, use the `keys`, `summaries` and `docs` methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple format options
>>> psy.plot.vector.keys('labels')

# show the summaries of a group of format options or of a format option
>>> psy.plot.vector.summaries('title')

# show the full documentation
>>> psy.plot.vector.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.vector.plot
```

### violinplot

(*args, **kwargs)

Make a violin plot of your data

This plotting method adds data arrays and plots them via `psy_simple.plotters.ViolinPlotter` plotters.

To plot data from a netCDF file type:

```python
>>> psy.plot.violinplot(filename, name=['my_variable'], ...)
```

Possible format options are

<table>
<thead>
<tr>
<th>arrowsize</th>
<th>arrowstyle</th>
<th>axiscolor</th>
<th>bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbar</td>
<td>cbarstyles</td>
<td>clabel</td>
<td>clabelprops</td>
</tr>
<tr>
<td>clabels</td>
<td>clabelweight</td>
<td>cmap</td>
<td>clabelprops</td>
</tr>
<tr>
<td>cctick</td>
<td>cctickprops</td>
<td>cticks</td>
<td>cticksize</td>
</tr>
<tr>
<td>ctickweight</td>
<td>datagrid</td>
<td>density</td>
<td>extend</td>
</tr>
<tr>
<td>figtitle</td>
<td>figtitleprops</td>
<td>figtitlesize</td>
<td>figtitleweight</td>
</tr>
<tr>
<td>grid</td>
<td>labelprops</td>
<td>labelsize</td>
<td>labelweight</td>
</tr>
<tr>
<td>linewidth</td>
<td>maskbetween</td>
<td>maskgeq</td>
<td>maskgreater</td>
</tr>
<tr>
<td>maskleq</td>
<td>maskless</td>
<td>plot</td>
<td>post</td>
</tr>
<tr>
<td>post_timing</td>
<td>sym_lims</td>
<td>text</td>
<td>ticksize</td>
</tr>
<tr>
<td>tickweight</td>
<td>title</td>
<td>titleprops</td>
<td></td>
</tr>
<tr>
<td>titlesize</td>
<td>titleweight</td>
<td>transpose</td>
<td>xlabel</td>
</tr>
<tr>
<td>xlim</td>
<td>xrotation</td>
<td>xticklabels</td>
<td>xtickprops</td>
</tr>
<tr>
<td>xticks</td>
<td>ylabel</td>
<td>ylim</td>
<td>yrotation</td>
</tr>
<tr>
<td>yticklabels</td>
<td>ytickprops</td>
<td>yticks</td>
<td></td>
</tr>
</tbody>
</table>
Examples

To explore the format options and their documentations, use the keys, summaries and docs methods. For example:

```python
>>> import psyplot.project as psy

# show the keys corresponding to a group or multiple formatoptions
>>> psy.plot.violinplot.keys('labels')

# show the summaries of a group of formatoptions or of a formatoption
>>> psy.plot.violinplot.summaries('title')

# show the full documentation
>>> psy.plot.violinplot.docs('plot')

# or access the documentation via the attribute
>>> psy.plot.violinplot.plot
```

`psyplot.project.close(num=None, figs=True, data=True, ds=True, remove_only=False)`

Close the project

This method closes the current project (figures, data and datasets) or the project specified by `num`

Parameters

- `num` *(int, None or 'all')* – if `int`, it specifies the number of the project, if `None`, the current subproject is closed, if `'all'`, all open projects are closed
- `figs` *(bool)* – Close the figures
- `data` *(bool)* – delete the arrays from the (main) project
- `ds` *(bool)* – If True, close the dataset as well
- `remove_only` *(bool)* – If True and `figs` is True, the figures are not closed but the plotters are removed

See also:

- `Project.close()`

`psyplot.project.gcp(main=False)`

Get the current project
Parameters `main` *(bool)* – If True, the current main project is returned, otherwise the current subproject is returned.

See also:

`scp()`  Sets the current project

`project()`  Creates a new project

`psyplot.project.get_project_nums()`  Returns the project numbers of the open projects

`psyplot.project.multiple_subplots(rows=1, cols=1, maxplots=None, n=1, delete=True, for_maps=False, *args, **kwargs)`  Function to create subplots.

This function creates so many subplots on so many figures until the specified number `n` is reached.

Parameters

- `rows` *(int)* – The number of subplots per rows
- `cols` *(int)* – The number of subplots per column
- `maxplots` *(int)* – The number of subplots per figure (if None, it will be row*cols)
- `n` *(int)* – number of subplots to create
- `delete` *(bool)* – If True, the additional subplots per figure are deleted
- `for_maps` *(bool)* – If True this is a simple shortcut for setting `subplot_kw=dict(projection=cartopy.crs.PlateCarree())` and is useful if you want to use the `mapplot, mapvector or mapcombined` plotting methods
- and `**kwargs` *(+args)* – anything that is passed to the `matplotlib.pyplot.subplots()` function

Returns  list of matplotlib.axes.SubplotBase instances

Return type  list

`psyplot.project.plot = <psyplot.project.ProjectPlotter object>`  `ProjectPlotter` of the current project. See the class documentation for available plotting methods

`psyplot.project.project(num=None, *args, **kwargs)`  Create a new main project

Parameters

- `num` *(int)* – The number of the project
- `iterable` *(iterable)* – The iterable (e.g. another list) defining this list
- `attrs` *(dict-like or iterable, optional)* – Global attributes of this list
- `auto_update` *(bool)* – Default: None. A boolean indicating whether this list shall automatically update the contained arrays when calling the `update()` method or not. See also the `no_auto_update` attribute. If None, the value from the 'lists.auto_update' key in the psyplot.rcParams dictionary is used.
- `new_name` *(bool or str)* – If False, and the `arr_name` attribute of the new array is already in the list, a ValueError is raised. If True and the `arr_name` attribute of the new array is not already in the list, the name is not changed. Otherwise, if the array name is already in use, `new_name` is set to ‘arr{0}’. If not True, this will be used for renaming (if the array name of `arr` is in use or not). '{0}' is replaced by a counter
• **main** (*Project*) – The main project this subproject belongs to (or None if this project is the main project)

**Returns**
The with the given *num* (if it does not already exist, it is created)

**Return type** *Project*

**See also:**

-scp() Sets the current project

gcp() Returns the current project

`psyplot.project.register_plotter`(*identifier*, *module*, *plotter_name*, *plotter_cls=None*, *sorter=True*, *plot_func=True*, *import_plotter=None*, **kwargs)

Register a `psyplot.plotter.Plotter` for the projects

This function registers plotters for the *Project* class to allow a dynamical handling of different plotter classes.

**Parameters**

- **identifier** (*str*) – Name of the attribute that is used to filter for the instances belonging to this plotter

- **module** (*str*) – The module from where to import the *plotter_name*

- **plotter_name** (*str*) – The name of the plotter class in *module*

- **sorter** (*bool*, optional) – If True, the *Project* class gets a new property with the name of the specified *identifier* which allows you to access the instances that are plotted by the specified *plotter_name*

- **plot_func** (*bool*, optional) – If True, the *ProjectPlotter* (the class that holds the plotting method for the *Project* class and can be accessed via the *Project.plot* attribute) gets an additional method to plot via the specified *plotter_name* (see Other Parameters below.)

- **import_plotter** (*bool*, optional) – If True, the plotter is automatically imported, otherwise it is only imported when it is needed. If *import_plotter* is None, then it is determined by the *psyplot.rcParams'*project.auto_import' item.

**Other Parameters**

- **prefer_list** (*bool*) – Determines the *prefer_list* parameter in the from_dataset method. If True, the plotter is expected to work with instances of `psyplot.InteractiveList` instead of `psyplot.InteractiveArray`.

- **default_slice** (*indexer*) – Index (e.g. 0 if method is ‘isel’) that shall be used for dimensions not covered by *dims* and *furtherdims*. If None, the whole slice will be used.

- **default_dims** (*dict*) – Default dimensions that shall be used for plotting (e.g. {'x': slice(None), 'y': slice(None)} for longitude-latitude plots)

- **show_examples** (*bool*, optional) – If True, examples how to access the plotter documentation are included in class documentation

- **example_call** (*str*, optional) – The arguments and keyword arguments that shall be included in the example of the generated plot method. This call will then appear as >>> psy.plot.%(*identifier*)s(%(*example_call*)s) in the documentation

- **plugin** (*str*) – The name of the plugin
psyplot.project.

```python
scp(project)
```

Set the current project

Parameters % (Project.scp.parameters) s -

See also:

- `gcp()` Returns the current project
- `project()` Creates a new project

psyplot.project.

```python
unregister_plotter(identifier, sorter=True, plot_func=True)
```

Unregister a psyplot.plotter.Plotter for the projects

Parameters

- `identifier (str)` – Name of the attribute that is used to filter for the instances belonging to this plotter or to create plots with this plotter
- `sorter (bool)` – If True, the identifier will be unregistered from the Project class
- `plot_func (bool)` – If True, the identifier will be unregistered from the ProjectPlotter class

**psyplot.utils module**

Miscallaneous utility functions for the psyplot package

**Classes**

- `DefaultOrderedDict([default_factory])` An ordered collections.defaultdict

**Functions**

- `check_key(key, possible_keys[, raise_error, ...])` Checks whether the key is in a list of possible keys
- `hashable(val)` Test if `val` is hashable and if not, get it’s string representation
- `is_iterable(iterable)` Test if an object is iterable
- `is_remote_url(path)`
- `join_dicts(dict[, delimiter, keep_all])` Join multiple dictionaries into one
- `sort_kwargs(kwargs, *param_lists)` Function to sort keyword arguments and sort them into dictionaries
- `unique_everseen(iterable[, key])` List unique elements, preserving order.

**class psyplot.utils.DefaultOrderedDict**(default_factory=None, *a*, **kw)**

Bases: collections.OrderedDict

An ordered collections.defaultdict

Taken from http://stackoverflow.com/a/6190500/562769

**Methods**

- `copy()` Return a shallow copy of the dictionary

```
copy()
```

Return a shallow copy of the dictionary
psyplot.utils.check_key(key, possible_keys, raise_error=True, name='formatoption keyword', msg='See show_fmtkeys function for possible formatoption keywords', *args, **kwargs)

Checks whether the key is in a list of possible keys

This function checks whether the given key is in possible_keys and if not looks for similar sounding keys

Parameters

- **key** *(str)* – Key to check
- **possible_keys** *(list of strings)* – A list of possible keys to use
- **raise_error** *(bool)* – If not True, a list of similar keys is returned
- **name** *(str)* – The name of the key that shall be used in the error message
- **msg** *(str)* – The additional message that shall be used if no close match to key is found
- ***args, **kwargs** – They are passed to the difflib.get_close_matches() function (i.e. n to increase the number of returned similar keys and cutoff to change the sensibility)

Returns

- **str** – The key if it is a valid string, else an empty string
- **list** – A list of similar formatoption strings (if found)
- **str** – An error message which includes

Raises **KeyError** – If the key is not a valid formatoption and raise_error is True

psyplot.utils.hashable(val)

Test if val is hashable and if not, get its string representation

Parameters **val** *(object)* – Any (possibly not hashable) python object

Returns The given val if it is hashable or its string representation

Return type **val or string**

psyplot.utils.is_iterable(iterable)

Test if an object is iterable

Parameters **iterable** *(object)* – The object to test

Returns True, if the object is an iterable object

Return type **bool**

psyplot.utils.is_remote_url(path)

psyplot.utils.join_dicts(dicts, delimiter=None, keep_all=False)

Join multiple dictionaries into one

Parameters

- **dicts** *(list of dict)* – A list of dictionaries
- **delimiter** *(str)* – The string that shall be used as the delimiter in case that there are multiple values for one attribute in the arrays. If None, they will be returned as sets
- **keep_all** *(bool)* – If True, all formatoptions are kept. Otherwise only the intersection

Returns The combined dictionary

Return type **dict**
psyplot.utils.sort_kwargs(kwars, *param_lists)

Function to sort keyword arguments and sort them into dictionaries

This function returns dictionaries that contain the keyword arguments from *kwargs corresponding given iterables
in *params

Parameters

• *kwars (dict) – Original dictionary
• *param_lists – iterables of strings, each standing for a possible key in kwars

Returns

len(params) + 1 dictionaries. Each dictionary contains the items of kwars corresponding
to the specified list in *param_lists. The last dictionary contains the remaining items

Return type list

psyplot.utils.unique_everseen(iterable, key=None)

List unique elements, preserving order. Remember all elements ever seen.

Function taken from https://docs.python.org/2/library/itertools.html

psyplot.version module

psyplot.warning module

Warning module of the psyplot python module

This module controls the warning behaviour of the module via the python builtin warnings module and introduces
three new warning classes:

.. autosummary::
   :toctree: ..

   PsPylotRuntimeWarning
   PsyPlotWarning
   PsyPlotCritical

Exceptions

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PsyPlotCritical</td>
<td>Critical UserWarning for psyplot module</td>
</tr>
<tr>
<td>PsyPlotRuntimeWarning</td>
<td>Runtime warning that appears only ones</td>
</tr>
<tr>
<td>PsyPlotWarning</td>
<td>Normal UserWarning for psyplot module</td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>critical</td>
<td>wrapper around the warnings.warn function for critical warnings.</td>
</tr>
<tr>
<td>customwarn</td>
<td>Use the psyplot.warning logger for categories being out of PsyPlotWarning and PsyPlotCritical and the default</td>
</tr>
<tr>
<td></td>
<td>warnings.showwarning function for all the others.</td>
</tr>
<tr>
<td>disable_warnings</td>
<td>Function that disables all warnings and all critical warnings (if critical evaluates to True) related to the psyplot</td>
</tr>
<tr>
<td></td>
<td>Module.</td>
</tr>
<tr>
<td>warn</td>
<td>wrapper around the warnings.warn function for non-critical warnings.</td>
</tr>
</tbody>
</table>

exception psyplot.warning.PsyPlotCritical
Bases: UserWarning

Critical UserWarning for psyplot module

deprecated

exception psyplot.warning.PsyPlotRuntimeWarning
    Bases: RuntimeWarning

Runtime warning that appears only ones

deprecated

exception psyplot.warning.PsyPlotWarning
    Bases: UserWarning

Normal UserWarning for psyplot module

psyplot.warning.critical(message, category=<class 'psyplot.warning.PsyPlotCritical'>, logger=None)
    wrapper around the warnings.warn function for critical warnings. logger may be a logging.Logger instance

psyplot.warning.customwarn(message, category, filename, lineno, *args, **kwargs)
    Use the psyplot.warning logger for categories being out of PsyPlotWarning and PsyPlotCritical and the default
    warnings.showwarning function for all the others.

psyplot.warning.disable_warnings(critical=False)
    Function that disables all warnings and all critical warnings (if critical evaluates to True) related to the psy-
    plot Module. Please note that you can also configure the warnings via the psyplot.warning logger (log-
    ging.getLogger(psyplot.warning)).

psyplot.warning.warn(message, category=<class 'psyplot.warning.PsyPlotWarning'>, logger=None)
    wrapper around the warnings.warn function for non-critical warnings. logger may be a logging.Logger instance

1.13 ToDos

Todo: Implement the visualization for UGrid data shown on the edge of the triangles

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/psyplot/checkouts/latest/psyplot/data.py:docstring of psyplot.data.UGridDecoder.get_triangles, line 32.)

1.14 Changelog

1.14.1 v1.2.1

This patch fixes compatibility issues with xarray 0.12 and cdo 1.5. Additionally we now officially drop support for
python 2.7.

1.14.2 v1.2.0

Added

• The psyplot.plotter.Plotter.initialize_plot method now takes a priority keyword to only
  initialize only formatoptions of a certain priority
Removed

- The installers from the `psyplot-conda` repositories have been deprecated. Instead, now download the latest `miniconda` and install psyplot and the plugins via `conda install -c conda-forge psy-maps psyplot-gui psy-reg`

Changed

- We generalized the handling of unstructured data as lined out in issue #6. The new method `psyplot.data.CFDecoder.get_cell_node_coord` returns the coordinates of the nodes for a given grid cell. These informations are used by the psy-simple and psy-maps plugins for displaying any unstructured data. See also the example on the visualization of unstructured grids
- We removed the inplace parameter for the CFDecoder methods since it is deprecated with xarray 0.12 (see issue #8). The `CFDecoder.decode_ds` method now always decodes inplace

1.14.3 v1.1.0

This new release mainly adds new xarray accossors (`psy`) for DataArrays and Datasets. Additionally we provide methods to calculate the spatially weighted mean, such as `fldmean`, `fldstd` and `fldpctl`

Added

- The `yaxis_inverted` and `xaxis_inverted` is now considered when loading and saving a matplotlib axes
- Added the `seaborn-style` command line argument
- Added the `concat_dim` command line argument
- Added the plot attribute to the DataArray and Dataset accessors. It is now possible to plot directly from the dataset and the data array
- Added `requires_replot` attribute for the `Formatoption` class. If this attribute is True and the formatoption is contained in an update, it is the same as calling `Plotter.update(replot=True)`.
- We added support for multifile datasets when saving a project. Multifile datasets are datasets that have been opened with, e.g. `psyplot.data.open_mfdataset` or `psyplot.project.plot.<plotmethod>(..., mfmode=True)`. This however does not always work with datasets opened with `xarray.open_mfdataset`. In these cases, you have to set the `Dataset.psy._concat_dim` attribute manually
- Added the `chname` parameter when loading a project. This parameter can be used to display another variable from the dataset than the one stored in the psyplot project file
- Added the `gridweights`, `fldmean`, `fldstd` and `fldpctl` methods to the `psy` DataArray accessor to calculate weighted means, standard deviations and percentiles over the spatial dimensions (x- and y).
- Added the `additional_children` and `additional_dependencies` parameters to the `Formatoption` initialization. These parameters can be used to provide additional children for a formatoption for one plotter class
- We added the `psyplot.plotter.Formatoption.get_fmt_widget` method which can be implemented to insert widgets in the formatoptions widget of the graphical user interface
114.4 v1.0.0

Added

- Changelog

Changed

- When creating new plots using the `psyplot.project.Project.plot` attribute, `scp` for the newly created subproject is only called when the corresponding `Project` is the current main project (gcp(True))

- The `alternate_paths` keyword in the `psyplot.project.Project.save_project` and `psyplot.data.ArrayList.array_info` methods has been changed to `alternative_paths`

- The `psyplot.project.Cdo` class does not accept any of the keywords `returnDA`, `returnMaps` or `returnLine` anymore. Instead it takes the `plot_method` keyword and several others.

- The `psyplot.project.close` method by default now removes the data from the current project and closes attached datasets

- The modules in the `psyplot.plotter` modules have been moved to separate packages to make the debugging and testing easier
  - The `psyplot.plotter.simple`, baseplotter and colors modules have been moved to the `psy-simple` package
  - The `psyplot.plotter.maps` and boxes modules have been moved to the `psy-maps` package
  - The `psyplot.plotter.linreg` module has been moved to the `psy-reg` package

- The endings of the yaml configuration files are now all `.yml`. Hence,
  - the configuration file name is now `psyplotrc.yml` instead of `psyplotrc.yaml`
  - the default logging configuration file name is now `logging.yml` instead of `logging.yaml`

- Under osx, the configuration directory is now also expected to be in `$HOME/.config/psyplot` (as it is for linux)
Fig. 1: *Usage of Climate Data Operators*
Fig. 2: Sharing format options

Fig. 3: Applying your own post processing
Fig. 4: Bar plot demo

Fig. 5: Line plot demo
Fig. 6: 2D plots

Fig. 7: Vector plot
Fig. 8: Violin plot demo

Fig. 9: Visualizing circumpolar data
Fig. 10: Basic data visualization on a map

Fig. 11: Visualizing unstructured data
Fig. 12: Creating and accessing a fit

Fig. 13: Plot a fit over a density plot
Howard to cite psyplot

When using psyplot, you should at least cite the publication in the Journal of Open Source Software:


BibTex - EndNote

Furthermore, each release of psyplot and its subprojects is associated with a DOI using zenodo.org. If you want to cite a specific version or plugin, please refer to the releases page of psyplot or the releases page of the corresponding subproject.
This package has been developed by Philipp Sommer.

I want to thank the matplotlib, xarray and cartopy developers for their great packages and of course the python developers for their fascinating work on this beautiful language.

A special thanks to Stefan Hagemann and Tobias Stacke from the Max-Planck-Institute of Meteorology in Hamburg, Germany for the motivation on this project and to the people of the Not yet visible agency for their advice in designing the logo and webpage.

Finally the author thanks the Swiss National Science Foundation (SNF) for their support. Funding for the author came from the ACACIA grant (CR1012_146314) and the HORNET grant (200021_169598).
INDICES AND TABLES

• genindex
• modindex
• search
p
psyplot, 68
psyplot.compat, 70
psyplot.compat.pycompat, 70
psyplot.config, 70
psyplot.config.logsetup, 71
psyplot.config.rcsetup, 72
psyplot.data, 83
psyplot.docstring, 126
psyplot.gdal_store, 127
psyplot.plotter, 128
psyplot.plugin_template, 151
psyplot.project, 151
psyplot.sphinxext, 81
psyplot.sphinxext.extended_napoleon, 81
psyplot.utils, 207
psyplot.version, 209
psyplot.warning, 209
INDEX

A
AbsoluteTimeDecoder (class in psyplot.data), 84
AbsoluteTimeEncoder (class in psyplot.data), 84
add_base_str () (psyplot.config.rcsetup.SubDict method), 78
all_dims (psyplot.data.ArrayList attribute), 86
all_names (psyplot.data.ArrayList attribute), 86
any_decoder (psyplot.plotter.Formatoption attribute), 132
append () (psyplot.data.ArrayList method), 86
append () (psyplot.data.InteractiveList method), 113
append () (psyplot.project.Project method), 181
append_original_doc () (in module psyplot.docstring), 127
arr_name (psyplot.data.InteractiveBase attribute), 111
arr_names (psyplot.data.ArrayList attribute), 86
arr_names (psyplot.project.Project attribute), 181
array_info () (psyplot.data.ArrayList method), 86
ArrayList (class in psyplot.data), 84
arrays (psyplot.data.ArrayList attribute), 86
ax (psyplot.data.InteractiveBase attribute), 111
ax (psyplot.plotter.Formatoption attribute), 132
ax (psyplot.plotter.Plotter attribute), 138
axes (psyplot.project.Project attribute), 182

B
barplot (psyplot.project.DataArrayPlotter attribute), 153
barplot (psyplot.project.DatasetPlotter attribute), 164
barplot (psyplot.project.ProjectPlotter attribute), 182
base (psyplot.config.rcsetup.SubDict attribute), 78
base (psyplot.data.InteractiveArray attribute), 106
base_str (psyplot.config.rcsetup.SubDict attribute), 78
base_variables (psyplot.data.InteractiveArray attribute), 106
base_variables (psyplot.plotter.Plotter attribute), 138
BEFOREPLOTTING (in module psyplot.plotter), 129
block_signals (psyplot.data.InteractiveBase attribute), 111
block_signals (psyplot.project.Project attribute), 182

C
can_decode () (psyplot.data.CFDecoder class method), 95
can_decode () (psyplot.data.UGridDecoder class method), 117
Cdo (class in psyplot.project), 152
CFDecoder (class in psyplot.data), 95
changed (psyplot.plotter.Formatoption attribute), 132
changed (psyplot.plotter.Plotter attribute), 138
check_and_set () (psyplot.plotter.Formatoption method), 132
check_data () (psyplot.plotter.Plotter class method), 138
check_data () (psyplot.project.DataArrayPlotterInterface method), 164
check_data () (psyplot.project.PlotterInterface method), 177
check_key () (in module psyplot.utils), 207
check_key () (psyplot.plotter.Plotter method), 138
children (psyplot.plotter.Formatoption attribute), 132
children (psyplot.plotter.PostProcessing attribute), 148
close () (in module psyplot.project), 204
close () (psyplot.project.Project method), 182
combined (psyplot.project.DataArrayPlotter attribute), 155
combined (psyplot.project.DatasetPlotter attribute), 166
combined (psyplot.project.Project attribute), 182
combined (psyplot.project.ProjectPlotter attribute), 194
config_path (in module psyplot.config), 71
connect () (psyplot.config.rcsetup.RcParams method), 73
connect () (psyplot.data.Signal method), 116
connections (psyplot.plotter.Formatoption attribute), 132
coords (psyplot.data.ArrayList attribute), 88
figs2draw (psyplot.plotter.Plotter attribute), 139
filename (psyplot.data.DatasetAccessor attribute), 104
find_all() (psyplot.config.rcsetup.RcParams method), 74
find_and_replace() (psyplot.config.rcsetup.RcParams method), 74
finish_update() (psyplot.plotter.Formatoption method), 133
fldmean (psyplot.project.DataArrayPlotter attribute), 156
fldmean (psyplot.project.DatasetPlotter attribute), 167
fldmean (psyplot.project.ProjectPlotter attribute), 196
fldmean() (psyplot.data.InteractiveArray method), 106
fldpctl1() (psyplot.data.InteractiveArray method), 106
fldstd() (psyplot.data.InteractiveArray method), 107
fmt_groups (psyplot.plotter.Plotter attribute), 139
format_time() (in module psyplot.plotter), 150
Formatoption (class in psyplot.plotter), 130
FormatoptionMeta (class in psyplot.plotter), 136
from_dataset() (psyplot.data.ArrayList class method), 89
from_dataset() (psyplot.data.InteractiveList class method), 114
from_dataset() (psyplot.project.Project class method), 184
from_dict() (psyplot.data.ArrayList class method), 90
get_enhanced_attrs() (psyplot.plotter.Plotter method), 139
get_filename() (in module psyplot.data), 119
get_fmt_widget() (psyplot.plotter.Formatoption method), 133
get_fmt_widget() (psyplot.plotter.PostTiming method), 150
get_fname_funcs (in module psyplot.data), 120
get_idims() (psyplot.data.CFDecoder method), 97
get_index_from_coord() (in module psyplot.data), 120
get_mesh() (psyplot.data.UGridDecoder method), 118
get_nodes() (psyplot.data.UGridDecoder method), 118
get_plotbounds() (psyplot.data.CFDecoder method), 97
get_project_nums() (in module psyplot.project), 205
get_sections() (psyplot.docstring.PsyplotDocstringProcessor method), 127
gget_t() (psyplot.data.CFDecoder method), 98
get_tdata() (in module psyplot.data), 121
get_tname() (psyplot.data.CFDecoder method), 98
gget_triangles() (psyplot.data.CFDecoder method), 98
get_variable_by_axis() (psyplot.data.CFDecoder method), 99
get_variables() (psyplot.gdal_store.GdalStore method), 128
get_versions() (in module psyplot), 69
get_vfunc() (psyplot.plotter.Plotter method), 139
get_x() (psyplot.data.CFDecoder method), 99
get_x() (psyplot.data.UGridDecoder method), 119
gget_xname() (psyplot.data.CFDecoder method), 100
get_y() (psyplot.data.CFDecoder method), 100
gget_y() (psyplot.data.UGridDecoder method), 119
gget_yname() (psyplot.data.CFDecoder method), 100
get_z() (psyplot.data.CFDecoder method), 101
gget_zname() (psyplot.data.CFDecoder method), 101
ggetcwd() (in module psyplot.compat.pycompat), 70
gridweights() (psyplot.data.InteractiveArray method), 107
gget_coord_idims() (psyplot.data.CFDecoder method), 97
gget_decoder() (psyplot.data.CFDecoder class method), 97
gget_default_value() (in module psyplot.compat.pycompat), 70
gget_dim() (psyplot.data.InteractiveArray method), 107
G
gcp() (in module psyplot.project), 204
GdalStore (class in psyplot.gdal_store), 128
gget attrs() (psyplot.gdal_store.GdalStore method), 128
gget_cell_node_coord() (psyplot.data.CFDecoder method), 96
gget_cell_node_coord() (psyplot.data.UGridDecoder method), 117
gget_configdir() (in module psyplot.config.rcsetup), 79
gget_coord() (psyplot.data.InteractiveArray method), 107
gget_coord_idims() (psyplot.data.CFDecoder method), 97
gget_decoder() (psyplot.data.CFDecoder class method), 97
gget_default_value() (in module psyplot.compat.pycompat), 70
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>psyplot.comapt</td>
<td>module</td>
<td>70</td>
</tr>
<tr>
<td>psyplot.comapt.pycmapt</td>
<td>module</td>
<td>70</td>
</tr>
<tr>
<td>psyplot.config</td>
<td>module</td>
<td>70</td>
</tr>
<tr>
<td>psyplot.config.logsetup</td>
<td>module</td>
<td>71</td>
</tr>
<tr>
<td>psyplot.config.rcsetup</td>
<td>module</td>
<td>72</td>
</tr>
<tr>
<td>psyplot.data</td>
<td>module</td>
<td>83</td>
</tr>
<tr>
<td>psyplot.docstring</td>
<td>module</td>
<td>126</td>
</tr>
<tr>
<td>psyplot.gdal_store</td>
<td>module</td>
<td>127</td>
</tr>
<tr>
<td>psyplot.plotter</td>
<td>module</td>
<td>128</td>
</tr>
<tr>
<td>psyplot.plugin_template</td>
<td>module</td>
<td>151</td>
</tr>
<tr>
<td>psyplot.project</td>
<td>module</td>
<td>151</td>
</tr>
<tr>
<td>psyplot.sphinxext.extended_napoleon</td>
<td>module</td>
<td>81</td>
</tr>
<tr>
<td>psyplot.sphinxext.extended_napoleon</td>
<td>module</td>
<td>81</td>
</tr>
<tr>
<td>psyplot.utils</td>
<td>module</td>
<td>207</td>
</tr>
<tr>
<td>psyplot.version</td>
<td>module</td>
<td>209</td>
</tr>
<tr>
<td>psyplot.warning</td>
<td>module</td>
<td>209</td>
</tr>
<tr>
<td>psyplot.fname</td>
<td>in module psyplot.config.rcsetup</td>
<td>79</td>
</tr>
<tr>
<td>PsyPlotCritical</td>
<td></td>
<td>209</td>
</tr>
<tr>
<td>PsyPlotDocstringProcessor</td>
<td>class in psyplot.docstring</td>
<td>126</td>
</tr>
<tr>
<td>PsyPlotRuntimeWarning</td>
<td></td>
<td>210</td>
</tr>
<tr>
<td>PsyPlotWarning</td>
<td></td>
<td>210</td>
</tr>
<tr>
<td>raw_data</td>
<td>psyplot.plotter.Formatoption attribute</td>
<td>134</td>
</tr>
<tr>
<td>rc</td>
<td>psyplot.plotter.Plotter attribute</td>
<td>142</td>
</tr>
<tr>
<td>RcParams</td>
<td>class in psyplot.config.rcsetup</td>
<td>72</td>
</tr>
<tr>
<td>rcParams</td>
<td>in module psyplot.config.rcsetup</td>
<td>80</td>
</tr>
<tr>
<td>register_decoder</td>
<td>psyplot.data.CFDecoder static method</td>
<td>102</td>
</tr>
<tr>
<td>register_plotter</td>
<td>in module psyplot.project</td>
<td>206</td>
</tr>
<tr>
<td>reinit</td>
<td>psyplot.plotter.Plotter method</td>
<td>142</td>
</tr>
<tr>
<td>remove</td>
<td>psyplot.config.rcsetup.RcParams method</td>
<td>75</td>
</tr>
<tr>
<td>remove</td>
<td>psyplot.data.ArrayList method</td>
<td>93</td>
</tr>
<tr>
<td>remove</td>
<td>psyplot.plotter.Formatoption method</td>
<td>134</td>
</tr>
<tr>
<td>rename</td>
<td>psyplot.data.List method</td>
<td>93</td>
</tr>
<tr>
<td>replace</td>
<td>psyplot.config.rcsetup.SubDict attribute</td>
<td>79</td>
</tr>
<tr>
<td>requires_clearing</td>
<td>psyplot.plotter.Formatoption attribute</td>
<td>134</td>
</tr>
<tr>
<td>requires_replot</td>
<td>psyplot.plotter.Formatoption attribute</td>
<td>135</td>
</tr>
<tr>
<td>safe_list</td>
<td>in module psyplot.config.rcsetup</td>
<td>80</td>
</tr>
<tr>
<td>save_project</td>
<td>psyplot.project.Project method</td>
<td>190</td>
</tr>
<tr>
<td>scp</td>
<td>in module psyplot.project</td>
<td>206</td>
</tr>
<tr>
<td>scp</td>
<td>psyplot.project.Project class method</td>
<td>191</td>
</tr>
<tr>
<td>sel</td>
<td>psyplot.data.InteractiveArray method</td>
<td>108</td>
</tr>
<tr>
<td>set_data</td>
<td>psyplot.plotter.Formatoption method</td>
<td>135</td>
</tr>
<tr>
<td>set_decoder</td>
<td>psyplot.plotter.Formatoption method</td>
<td>135</td>
</tr>
<tr>
<td>set_value</td>
<td>psyplot.plotter.DictFormatoption method</td>
<td>129</td>
</tr>
<tr>
<td>set_value</td>
<td>psyplot.plotter.Formatoption method</td>
<td>135</td>
</tr>
<tr>
<td>setup</td>
<td>in module psyplot.sphinxext.extended_napoleon</td>
<td>83</td>
</tr>
<tr>
<td>setup_coords</td>
<td>in module psyplot.data</td>
<td>124</td>
</tr>
<tr>
<td>setup_logging</td>
<td>in module psyplot.config.logsetup</td>
<td>71</td>
</tr>
<tr>
<td>show</td>
<td>psyplot.plotter.Plotter method</td>
<td>143</td>
</tr>
<tr>
<td>show</td>
<td>psyplot.plotter.Plotter method</td>
<td>143</td>
</tr>
<tr>
<td>show</td>
<td>psyplot.project.Project static method</td>
<td>192</td>
</tr>
<tr>
<td>show_docs</td>
<td>psyplot.plotter.Plotter class method</td>
<td>143</td>
</tr>
<tr>
<td>show_keys</td>
<td>psyplot.plotter.Plotter class method</td>
<td>144</td>
</tr>
<tr>
<td>show_plot_methods</td>
<td>psyplot.project.ProjectPlotter method</td>
<td>202</td>
</tr>
<tr>
<td>show_summaries</td>
<td>psyplot.plotter.Plotter class method</td>
<td>144</td>
</tr>
<tr>
<td>Signal</td>
<td>class in psyplot.data</td>
<td>116</td>
</tr>
<tr>
<td>simple</td>
<td>psyplot.project.Project attribute</td>
<td>192</td>
</tr>
<tr>
<td>sort_kwargs</td>
<td>in module psyplot.utils</td>
<td>208</td>
</tr>
<tr>
<td>standardize_dims</td>
<td>psyplot.data.CFDecoder method</td>
<td>102</td>
</tr>
<tr>
<td>START</td>
<td>in module psyplot.plotter</td>
<td>150</td>
</tr>
<tr>
<td>start_update</td>
<td>psyplot.data.ArrayList method</td>
<td>93</td>
</tr>
<tr>
<td>start_update</td>
<td>psyplot.data.InteractiveArray method</td>
<td>108</td>
</tr>
<tr>
<td>start_update</td>
<td>psyplot.data.InteractiveBase method</td>
<td>111</td>
</tr>
<tr>
<td>start_update</td>
<td>psyplot.data.InteractiveList method</td>
<td>116</td>
</tr>
<tr>
<td>start_update</td>
<td>psyplot.plotter.Plotter method</td>
<td>145</td>
</tr>
<tr>
<td>SubDict</td>
<td>class in psyplot.config.rcsetup</td>
<td>76</td>
</tr>
<tr>
<td>summaries</td>
<td>psyplot.project.PlotterInterface method</td>
<td>178</td>
</tr>
<tr>
<td>summaries</td>
<td>psyplot.project.Project method</td>
<td>192</td>
</tr>
<tr>
<td>t_patterns</td>
<td>in module psyplot.data</td>
<td>125</td>
</tr>
<tr>
<td>to_array</td>
<td>psyplot.data.DatasetAccessor method</td>
<td>104</td>
</tr>
</tbody>
</table>
to_dataframe() \hspace{1em} \textit{(psyplot.data.InteractiveList} \hspace{1em} \text{method), 116}

to_interactive_list() \hspace{1em} \textit{(psyplot.data.InteractiveArray} \hspace{1em} \text{method), 109}

to_interactive_list() \hspace{1em} \textit{(psyplot.data.InteractiveBase} \hspace{1em} \text{method), 112}

to_interactive_list() \hspace{1em} \textit{(psyplot.data.InteractiveList} \hspace{1em} \text{method), 116}

to_netcdf() \hspace{1em} \textit{(in module psyplot.data), 125}

to_slice() \hspace{1em} \textit{(in module psyplot.data), 126}

trace \hspace{1em} \textit{(psyplot.config.rcsetup.SubDict} \hspace{1em} \text{attribute), 79}

U

UGridDecoder \hspace{1em} \textit{(class in psyplot.data), 117}

unique_everseen() \hspace{1em} \textit{(in module psyplot.utils), 209}

unregister_plotter() \hspace{1em} \textit{(in module psyplot.project), 207}

unshare() \hspace{1em} \textit{(psyplot.plotter.Plotter} \hspace{1em} \text{method), 145}

unshare() \hspace{1em} \textit{(psyplot.project.Project} \hspace{1em} \text{method), 192}

unshare_me() \hspace{1em} \textit{(psyplot.plotter.Plotter} \hspace{1em} \text{method), 145}

update() \hspace{1em} \textit{(psyplot.config.rcsetup.ReParams} \hspace{1em} \text{method), 76}

update() \hspace{1em} \textit{(psyplot.config.rcsetup.SubDict} \hspace{1em} \text{method), 79}

update() \hspace{1em} \textit{(psyplot.data.ArrayList} \hspace{1em} \text{method), 94}

update() \hspace{1em} \textit{(psyplot.data.InteractiveArray} \hspace{1em} \text{method), 109}

update() \hspace{1em} \textit{(psyplot.data.InteractiveBase} \hspace{1em} \text{method), 112}

update() \hspace{1em} \textit{(psyplot.plotter.Formatoption} \hspace{1em} \text{method), 136}

update() \hspace{1em} \textit{(psyplot.plotter.PostProcessing} \hspace{1em} \text{method), 146}

update() \hspace{1em} \textit{(psyplot.plotter.PostTiming} \hspace{1em} \text{method), 150}

update_after_plot() \hspace{1em} \textit{(psyplot.plotter.Formatoption} \hspace{1em} \text{attribute), 136}

update_from_defaultParams() \hspace{1em} \textit{(psyplot.config.rcsetup.ReParams} \hspace{1em} \text{method), 76}

V

validate() \hspace{1em} \textit{(psyplot.config.rcsetup.ReParams} \hspace{1em} \text{attribute), 76}

validate() \hspace{1em} \textit{(psyplot.plotter.Formatoption} \hspace{1em} \text{attribute), 136}

validate() \hspace{1em} \textit{(psyplot.plotter.PostProcessing} \hspace{1em} \text{static} \hspace{1em} \text{method), 149}

validate() \hspace{1em} \textit{(psyplot.plotter.PostTiming} \hspace{1em} \text{static} \hspace{1em} \text{method), 150}

validate_bool() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 80}

validate_bool_maybe_none() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 80}

validate_dict() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 80}

validate_path_exists() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 80}

validate_str() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 80}

validate_stringlist() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 80}

validate_stringset() \hspace{1em} \textit{(in module psyplot.config.rcsetup), 81}

value() \hspace{1em} \textit{(psyplot.plotter.Formatoption} \hspace{1em} \text{attribute), 136}

value2pickle() \hspace{1em} \textit{(psyplot.plotter.Formatoption} \hspace{1em} \text{attribute), 136}

value2share() \hspace{1em} \textit{(psyplot.plotter.Formatoption} \hspace{1em} \text{attribute), 136}

values() \hspace{1em} \textit{(psyplot.config.rcsetup.ReParams} \hspace{1em} \text{method), 76}

vector() \hspace{1em} \textit{(psyplot.project.DataArrayPlotter} \hspace{1em} \text{attribute), 162}

vector() \hspace{1em} \textit{(psyplot.project.DatasetPlotter} \hspace{1em} \text{attribute), 173}

vector() \hspace{1em} \textit{(psyplot.project.Project} \hspace{1em} \text{attribute), 193}

vector() \hspace{1em} \textit{(psyplot.project.ProjectPlotter} \hspace{1em} \text{attribute), 202}

violinplot() \hspace{1em} \textit{(psyplot.project.DataArrayPlotter} \hspace{1em} \text{attribute), 163}

violinplot() \hspace{1em} \textit{(psyplot.project.DatasetPlotter} \hspace{1em} \text{attribute), 174}

violinplot() \hspace{1em} \textit{(psyplot.project.Project} \hspace{1em} \text{attribute), 193}

violinplot() \hspace{1em} \textit{(psyplot.project.ProjectPlotter} \hspace{1em} \text{attribute), 203}

W

warn() \hspace{1em} \textit{(in module psyplot.warning), 210}

with_gui() \hspace{1em} \textit{(in module psyplot), 69}

with_plotter() \hspace{1em} \textit{(psyplot.data.ArrayList} \hspace{1em} \text{attribute), 95}

with_plotter() \hspace{1em} \textit{(psyplot.project.Project} \hspace{1em} \text{attribute), 193}