# plottr

**Wolfgang Pfaff**

**Feb 11, 2022**

# CONTENTS:

CHAPTER

ONE

# ELEMENTARY CONCEPTS

## 1.1 Data formats

The main format we're using within plottr is the `DataDict`. While most of the actual numeric data will typically live in numpy arrays (or lists, or similar), they don't typically capture easily arbitrary metadata and relationships between arrays. Say, for example, we have some data `z` that depends on two other variables, `x` and `y`. This information has be stored somewhere, and numpy doesn't offer readily a solution here. There are various extensions, for example xarray or the MetaArray class. Those however typically have a grid format in mind, which we do not want to impose. Instead, we use a wrapper around the python dictionary that contains all the required meta information to infer the relevant relationships, and that uses numpy arrays internally to store the numeric data. Additionally we can story any other arbitrary meta data.

A DataDict container (a *dataset*) can contain multiple *data fields* (or variables), that have values and can contain their own meta information. Importantly, we distinct between independent fields (the *axes*) and dependent fields (the *data*).

Despite the naming, *axes* is not meant to imply that the *data* have to have a certain shape (but the degree to which this is true depends on the class used). A list of classes for different shapes of data can be found below.

The basic structure of data conceptually looks like this (we inherit from *dict*)

```
{
    'data_1' : {
        'axes' : ['ax1', 'ax2'],
        'unit' : 'some unit',
        'values' : [ ... ],
        '__meta__' : 'This is very important data',
        ...
    },
    'ax1' : {
        'axes' : [],
        'unit' : 'some other unit',
        'values' : [ ... ],
        ...,
    },
    'ax2' : {
        'axes' : [],
        'unit' : 'a third unit',
        'values' : [ ... ],
        ...,
    },
    '__globalmeta__' : 'some information about this data set',
    '__moremeta__' : 1234,
    ...
}
```

In this case we have one dependent variable, `data_1`, that depends on two axes, `ax1` and `ax2`. This concept is restricted only in the following way:

- a dependent can depend on any number of independents
- an independent cannot depend on other fields itself
- any field that does not depend on another, is treated as an axis

Note that meta information is contained in entries whose keys start and end with double underscores. Both the DataDict itself, as well as each field can contain meta information.

In the most basic implementation, the only restriction on the data values is that they need to be contained in a sequence (typically as list, or numpy array), and that the length of all values in the data set (the number of *records*) must be equal. Note that this does not preclude nested sequences!

### 1.1.1 Relevant data classes

**DataDictBase** The main base class. Only checks for correct dependencies. Any requirements on data structure is left to the inheriting classes. The class contains methods for easy access to data and metadata.

**DataDict** The only requirement for valid data is that the number of records is the same for all data fields. Contains some tools for expansion of data.

**MeshgridDataDict** For data that lives on a grid (not necessarily regular).

For more information, see the API documentation.

## 1.2 Nodes and Flowcharts

### 1.2.1 Contents



The basic concept of modular data analyis as we use it in plottr consists of *Nodes* that are connected directionally to form a *Flowchart*. This terminology is adopted from the great pyqtgraph project; we currently use their *Node* and *Flowchart* API under the hood as well. Executing the flowchart means that data flows through the nodes via connections that have been made between them, and gets modified in some way by each node along the way. The end product is then the fully processed data. This whole process is typically on-demand: If a modification of the data flow occurs somewhere in the flowchart – e.g., due to user input – then only 'downstream' nodes need to re-process data in order to keep the flowchart output up to date.

### 1.2.2 Setting up flowcharts

TBD.

### 1.2.3 Creating custom nodes

The following are some general notes. For an example see the notebook `Custom nodes` under `doc/examples`.

The class *plottr.node.node.Node* forms the basis for all nodes in plottr. It is an extension of `pyqtgraph`'s Node class with some additional tools, and defaults.

#### Basics:

The actual data processing the node is supposed to do is implemented in *plottr.node.node.Node.process()*.

#### Defaults:

Per default, we use an input terminal (`dataIn`), and one output terminal (`dataOut`). Can be overwritten via the attribute *plottr.node.node.Node.terminals*.

#### User options:

We use `property` for user options. i.e., we implement a setter and getter function (e.g., with the `@property` decorator). The setter can be decorated with *plottr.node.node.updateOption()* to automatically process the option change on assignment.

#### Synchronizing Node and UI:

The UI widget is automatically instantiated when *plottr.node.node.Node.uiClass* is set to an appropriate node widget class, and *plottr.node.node.Node.useUi* is `True`.

Messaging between Node and Node UI is implemented through Qt signals/slots. Any update to a node property is signalled automatically when the property setter is decorated with *plottr.node.node.updateOption()*. A setter decorated with `@updateOption('myOption')` will, on assignment of the new value, call the function assigned to `plottr.node.node.NodeWidget.optSetter['myOption']`.

Vice versa, there are tools to notify the node of changes made through the UI. Any trigger (such as a widget signal) can be connected to the UI by calling the functions *plottr.node.node.NodeWidget.signalOption()* with the option name (say, `myOption`) as argument, or *plottr.node.node.NodeWidget.signalAllOptions()*. In the first case, the value of the option is taken by calling `plottr.node.node.NodeWidget.optGetter['myOption']()`, and then the name of the option and that value are emitted through *plottr.node.node.updateGuiFromNode()*; this is connected to *plottr.node.node.Node.setOption()* by default. Similarly, *plottr.node.node.NodeWidget.signalAllOptions()* results in a signal leading to *plottr.node.node.Node.setOptions()*.

The implementation of the suitable triggers for emitting the option value and assigning functions to entries in `optSetters` and `optGetters` is up to the re-implementation.

### Example implementation:

The implementation of a custom node with GUI can then looks something like this:

```
class MyNode(Node):

    useUi = True
    uiClass = MyNodeGui

    ...

    @property
    def myOption(self):
        return self._myOption

    # the name in the decorator should match the name of the
    # property to make sure communication goes well.
    @myOption.setter
    @updateOption('myOption')
    def myOption(self, value):
        # this could include validation, etc.
        self._myOption = value

    ...
```

That is essentially all that is needed for the Node; only the process function that does something depending on the value of `myOption` is missing here. The UI class might then look like this:

```
class MyNodeGui(NodeWidget):

    def __init__(self, parent=None):
        # this is a Qt requirement
        super().__init__(parent)

        somehowSetUpWidget()

        self.optSetters = {
            'myOption' : self.setMyOption,
        }
        self.optGetters = {
            'myOption' : self.getMyOption,
        }

        # often the trigger will be a valueChanged function or so,
        # that returns a value. Since the signalOption function
        # doesn't require one, we can use a lambda to bypass, if necessary.
        self.somethingChanged.connect(lambda x: self.signalOption('myOption'))

    def setMyOption(self, value):
        doSomething()

    def getMyOption(self):
        return getInfoNeeded()
```

This node can then already be used, with the UI if desired, in a flowchart.

CHAPTER

TWO

# PREDEFINED NODES

## 2.1 Preparing data for plotting

### 2.1.1 Data Selection

To be written. Describe data selector, idea of compatible data.

### 2.1.2 Data Gridding

Converting data from a tabular format to a grid is done by the by the node class *DataGridder*:

**class** plottr.node.grid.**DataGridder**(*name*)

A node that can put data onto or off a grid. Has one property: *grid*. Its possible values are governed by a main option, plus (optional) additional options.

**property grid**

Specification for how to grid the data. Consists of a main option and (optional) additional options.

The main option is of type *GridOption*, and the additional options are given as a dictionary. Assign as tuple, like:

```
>>> dataGridder.grid = GridOption.<option>, dict((**options)
```

All types of *GridOption* are valid main options:

- ***GridOption.noGrid*** – will leave tabular data as is, and flatten gridded data to result in tabular data
- ***GridOption.guessShape*** – use *guess_shape_from_datadict()* and *datadict_to_meshgrid()* to infer the grid, if the input data is tabular.
- ***GridOption.specifyShape*** – reshape the data using a specified shape.
- ***GridOption.metadataShape*** – use the shape specified in the dataset metadata

Some types may required additional options. At the moment, this is only the case for *GridOption.specifyShape*. Manual specification of the shape requires two additional options, *order* and *shape*:

- ***order*** – a list of the input data axis dimension names, in the internal order of the input data array. This order is used to transpose the data before re-shaping with the *shape* information. Often this is simply the axes list; then the transpose has no effect. A different order needed when the the data to be gridded is not in *C* order, i.e., when the axes order given in the DataDict is not from slowest changing to fastest changing.

- ***shape*** – a tuple of integers that can be used to reshape the input data to obtain a grid. Must be in the same order as *order* to work correctly.

See *data.datadict.datadict_to_meshgrid()* for additional notes; *order* will be passed to *inner_axis_order* in that function, and *shape* to *target_shape*.

**Return type** Tuple[*GridOption*, Dict[str, Any]]

**class** plottr.node.grid.**GridOption**

Options for how to grid data.

**guessShape = 1**

guess the shape of the grid

**metadataShape = 3**

read the shape from DataSet Metadata (if available)

**noGrid = 0**

don't put on a grid

**specifyShape = 2**

manually specify the shape of the grid

### 2.1.3 Data slicing and reduction to 1D or 2D

To be written.

CHAPTER

THREE

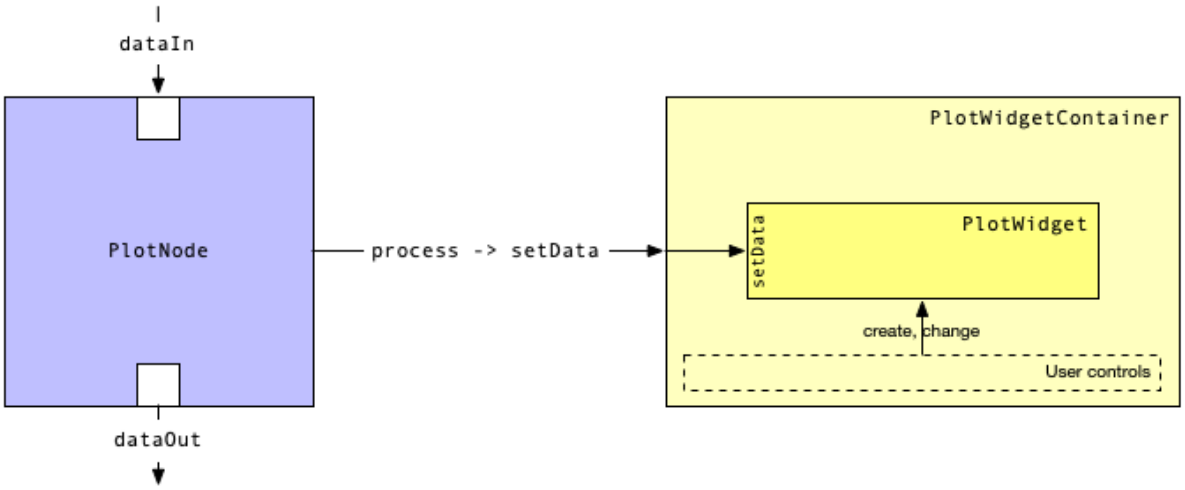
# PLOTTING

## 3.1 Plot Nodes

Plots have a somewhat special role in the node system: We need a node to make plots aware of incoming data, but the node will (typically) not do anything to the data. In the simplest case, `Node.process` will just call a function that triggers plotting, using the just received data. For many applications the base class `PlotNode` will do the job without any need to customize.

## 3.2 Plot Widgets

To make the plot node aware of the presence of a GUI, a suitable widget must be connected to it. This can be done by instantiating `PlotWidgetContainer`, and passing the instance to the node's `setPlotWidgetContainer` method. This will make sure that the container's `setData` is called whenever the node receives data. The container can then in turn host a `PlotWidgetContainer`, which is connected by using `setPlotWidget`. The reason why we don't connect the widget directly to the node is that the container may provide controls to change the widgets through user controls.




See the *API documentation* for more details.

## 3.3 Automatic plotting with Matplotlib

The most commonly used plot widget is based on matplotlib: `AutoPlot`. It determines automatically what an appropriate visualization of the received data is, and then plots that (at least if it can determine a good way to plot). At the same time it gives the user a little bit of control over the appearance (partially through native matplotlib tools). To separate plotting from the GUI elements we use `FigureMaker`.

See the *API documentation* for more details.

CHAPTER

FOUR

# API DOCUMENTATION

## 4.1 Node and Flowchart core elements

### 4.1.1 Node essentials: the node module

node.py

Contains the base class for Nodes.

**class** plottr.node.node.**Node**(*name*)

Base class of the Node we use for plotter.

This class inherits from pyqtgraph's Node, and adds a few additional tools, and some defaults.

**ctrlWidget**()

Returns the node widget, if it exists.

**Return type** Optional[QWidget]

**dataAxesChanged**

signal emitted when available data axes change emits a the list of names of new axes

**dataDependentsChanged**

signal emitted when available dependents change emits a the list of names of new dependents

**dataFieldsChanged**

signal emitted when any available data fields change (dep. and indep.) emits a the list of names of new axes

**dataShapesChanged**

signal emitted when data shapes change

**dataStructureChanged**

signal emitted when data structure changes (fields, or dtype)

**dataTypeChanged**

signal emitted when data type changes

**logger**()

Get a logger for this node

**Return type** Logger

**Returns** logger with a name that can be traced back easily to this node.

**newDataStructure**

when data structure changes, emits (structure, shapes, type)

**nodeName = 'Node'**
Name of the node. used in the flowchart node library.

**optionChangeNotification**
A signal to notify the UI of option changes arguments is a dictionary of options and new values.

**process**(*dataIn=None*)
Process data through this node. This method is called any time the flowchart wants the node to process data. It will be called with one keyword argument corresponding to each input terminal, and must return a dict mapping the name of each output terminal to its new value.

This method is also called with a 'display' keyword argument, which indicates whether the node should update its display (if it implements any) while processing this data. This is primarily used to disable expensive display operations during batch processing.

**Return type** Optional[Dict[str, Optional[*DataDictBase*]]]

**setOption**(*nameAndVal*)
Set an option.

name is the name of the property, not the string used for referencing (which could in principle be different).

**Parameters** **nameAndVal** (Tuple[str, Any]) – tuple of option name and new value

**Return type** None

**setOptions**(*opts*)
Set multiple options.

**Parameters** **opts** (Dict[str, Any]) – a dictionary of property name : value pairs.

**Return type** None

**setupUi**()
setting up the UI widget.

Gets called automatically in the node initialization. Automatically connect the UIs methods to signal option values.

Inheriting classes can use this method to do additional setup of the UI widget (like connecting additional signals/slots between node and node widget).

**Return type** None

**terminals = {'dataIn': {'io': 'in'}, 'dataOut': {'io': 'out'}}**
one input and one output.

**Type** Default terminals

**uiClass: Optional[Type[NodeWidget]] = None**
UI node widget class. If not None, and useUi is True, an instance of the widget is created, and signal/slots are connected.

**uiVisibleByDefault = False**
Whether the ui should be visible by default

**update**(*signal=True*)
Collect all input values, attempt to process new output values, and propagate downstream. Subclasses should call update() whenever thir internal state has changed (such as when the user interacts with the Node's control widget). Update is automatically called when the inputs to the node are changed.

**Return type** None

**useUi = True**
Whether or not to automatically set up a UI widget.

**validateOptions**(*data*)
Validate the user options

Does nothing in this base implementation. Can be reimplemented by any inheriting class.

**Parameters** **data** (*DataDictBase*) – the data to verify the options against.

**Return type** bool

**class** plottr.node.node.**NodeWidget**(*parent=None*, *embedWidgetClass=None*, *node=None*)
Base class for Node control widgets.

For the widget class to set up communication with the Node automatically, make sure to set plottr.node.node.NodeWidget.optGetters and plottr.node.node.NodeWidget.optSetters for a widget class.

**allOptionsToNode**
(object)) all options to the node.

**Type** signal (args

**getAllOptions**()
Return all options as a dictionary

**Return type** Dict[str, Any]

**icon: Optional[PyQt5.QtGui.QIcon] = None**
icon for this node

**optionToNode**
object)) to emit to notify the node of a (changed) user option.

**Type** signal (args

**preferredDockWidgetArea = 1**
preferred location of the widget when used as dock widget

**setOptionFromNode**(*opt*, *value*)
Set an option from the node

Calls the set function specified in the class' optSetters. Decorated with @updateGuiFromNode.

**Parameters**

- **opt** (str) – name of the option
- **value** (Any) – value to set

**Return type** None

**setOptionsFromNode**(*opts*)
Set all options without triggering updates back to the node.

**Return type** None

**signalAllOptions**()
Return all options as a dictionary

Decorated with @emitGuiUpdate('optionToNode').

**Return type** Dict[str, Any]

**signalOption**(*name*)
Returns name and value of an option.

Value is determined from the optGetters. Decorated with @emitGuiUpdate('optionToNode').

**Parameters** **name** (str) – name of the option

**Return type** Tuple[str, Any]

plottr.node.node.**emitGuiUpdate**(*signalName*)
Decorator for UI functions to emit the signal signalName (given as argument the decorator), with the return of the wrapped function.

Signal is only emitted if the flag controlled by updateGuiFromNode is not True, i.e., if the option change was *not* caused by a function decorated with updateGuiFromNode.

**Parameters** **signalName** (str) – name of the signal to emit.

**Return type** Callable[[Callable[..., Any]], Callable[..., None]]

plottr.node.node.**updateGuiFromNode**(*func*)
Decorator for the UI to set an internal flag to during execution of the wrapped function. Prevents recursive updating (i.e., if the node sends a new option value to the UI for updating, the UI will then *not* notify the node back after making the update).

**Return type** Callable[..., ~V]

plottr.node.node.**updateGuiQuietly**(*func*)
Decorator for the UI to set an internal flag to during execution of the wrapped function. Prevents recursive updating (i.e., if the node sends a new option value to the UI for updating, the UI will then *not* notify the node back after making the update).

**Return type** Callable[..., ~V]

plottr.node.node.**updateOption**(*optName=None*)
Decorator for property setters that are handy for user options.

Property setters in nodes that are decorated with this will do two things: * call Node.update, in order to update the flowchart. * if there is a UI, we call the matching optSetter function.

**Parameters** **optName** (Optional[str]) – name of the property.

**Return type** Callable[[Callable[[~R, ~S], ~T]], Callable[[~R, ~S], ~T]]

## 4.2 Plotting elements

### 4.2.1 Base plotting elements

#### Overview

#### Classes for plotting functionality

- *PlotNode* : The base class for a *.Node* with the purpose of receiving data for visualization.
- *PlotWidgetContainer* : A class that contains a *PlotWidget* (and can change it during runtime)
- *PlotWidget* : An abstract widget that can be inherited to implement actual plotting.
- *AutoFigureMaker* : A convenience class for semi-automatic generation of figures. The purpose is to keep actual plotting code out of the plot widget. This is not mandatory, just convenient.

### Data structures

- *PlotDataType* : Enum with types of data that can be plotted.
- *ComplexRepresentation*: Enum with ways to represent complex-valued data.

### Additional tools

- *makeFlowchartWithPlot()* : convenience function for creating a flowchart that leads to a plot node.
- *determinePlotDataType()* : try to infer which type of plot data is in a data set.

### Object Documentation

`plottr.plot.base` – Contains the base classes for plotting nodes and widgets. Everything in here is independent of actual plotting backend, and does not contain plotting commands.

**class** plottr.plot.base.**AutoFigureMaker**

A class for semi-automatic creation of plot figures. It must be inherited to tie it to a specific plotting backend.

The main purpose of this class is to (a) implement actual plotting of plot items, and (b) distribute plot items correctly among subpanels of a figure.

FigureMaker is a context manager. The user should eventually only need to add data and specify what kind of data it is. FigureMaker will then generate plots from that.

In the simplest form, usage looks something like this:

```
>>> with AutoFigureMaker() as fm:
>>>     fm.addData(x, y, [...])
>>>     [...]
```

See *addData()* for details on how to specify data and how to pass plot options to it.

**addData**(**data*, *join=None*, *labels=None*, *plotDataType=<PlotDataType.unknown: 1>*, ***plotOptions*)

Add data to the figure.

**Parameters**

- **data** (Union[ndarray, MaskedArray]) – data arrays describing the plot (one or more independents, one dependent)
- **join** (Optional[int]) – ID of a plot item the new item should be shown together with in the same subplot
- **labels** (Optional[List[str]]) – list of labels for the data arrays
- **plotDataType** (*PlotDataType*) – what kind of plot data the supplied data contains.
- **plotOptions** (Any) – options (as kwargs) to be passed to the actual plot functions (depends on the backend)

**Return type** int

**Returns** ID of the new plot item.

**addSubPlot**()

Add a new subplot.

**Return type** int

**Returns** ID of the new subplot.

**allPlotIds: List = None**
ids of all plot items, incl those who are 'joined' with 'main' plot items.

**combineTraces: bool = None**
whether to combine 1D traces into one plot

**complexRepresentation: ComplexRepresentation = None**
how to represent complex data. must be set before adding data to the plot to have an effect.

**dataDimensionsInSubPlot**(*subPlotId*)
Determine what the data dimensions are in a subplot.

**Parameters** **subPlotId** (int) – ID of the subplot

**Return type** Dict[int, int]

**Returns** dictionary with plot id as key, data dimension (i.e., number of independents) as value.

**findPlotIndexInSubPlot**(*plotId*)
find the index of a plot in its subplot

**Parameters** **plotId** (int) – plot ID to check

**Return type** int

**Returns** index at which the plot is located in its subplot.

**formatSubPlot**(*subPlotId*)
Format a subplot. May be implemented by an inheriting class. By default, does nothing.

**Parameters** **subPlotId** (int) – ID of the subplot.

**Return type** Any

**Returns** Depends on inheriting class.

**makeSubPlots**(*nSubPlots*)
Generate the subplots. Called after all data has been added. Must be implemented by an inheriting class.

**Parameters** **nSubPlots** (int) – number of subplots

**Return type** List[Any]

**Returns** return values of the subplot generation methods.

**nSubPlots**()
Count the subplots in the figure.

**Return type** int

**Returns** number of subplots

**plot**(*plotItem*)
Plot an item. Must be implemented by an inheriting class.

**Parameters** **plotItem** (*PlotItem*) – the item to plot.

**Return type** Any

**Returns** Depends on the inheriting class.

**plotIds: List = None**
ids of all main plot items (does not contain derived/secondary plot items)

**plotIdsInSubPlot**(*subPlotId*)
return all plot IDs in a given subplot

**Parameters** **subPlotId** (int) – ID of the subplot

**Return type** List[int]

**Returns** list of plot IDs

**plotItems: OrderedDictType[int, PlotItem] = None**
items that will be plotted

**previousPlotId**()
Get the ID of the most recently added plot item. :rtype: Optional[int] :return: the ID.

**subPlotItems**(*subPlotId*)
Get items in a given subplot.

**Parameters** **subPlotId** (int) – ID of the subplot

**Return type** OrderedDict[int, *PlotItem*]

**Returns** Dictionary with all plot items and their ids.

**subPlotLabels**(*subPlotId*)
Get the data labels for a given subplot.

**Parameters** **subPlotId** (int) – ID of the subplot.

**Return type** List[List[str]]

**Returns** a list with one element per plot item in the subplot. Each element contains a list of the labels for that item.

**subPlots: OrderedDictType[int, SubPlot] = None**
subplots to create

**class** plottr.plot.base.**ComplexRepresentation**(*label*)
Options for plotting complex-valued data.

**magAndPhase = 4**
magnitude and phase

**real = 1**
only real

**realAndImag = 2**
real and imaginary

**realAndImagSeparate = 3**
real and imaginary, separated

**class** plottr.plot.base.**PlotDataType**
Types of (plottable) data

**grid2d = 5**
grid data with 2 dependents

**line1d = 3**
line data with 1 dependent (data is on a grid)

**scatter1d = 2**
scatter-type data with 1 dependent (data is not on a grid)

**scatter2d = 4**
scatter data with 2 dependents (data is not on a grid)

**unknown = 1**
unplottable data

**class** plottr.plot.base.**PlotItem**(*data: List[Union[numpy.ndarray, numpy.ma.core.MaskedArray]], id: int, subPlot: int, plotDataType: plottr.plot.base.PlotDataType = <PlotDataType.unknown: 1>, labels: Optional[List[str]] = None, plotOptions: Optional[Dict[str, Any]] = None, plotReturn: Optional[Any] = None*)

Data class describing a plot item in *AutoFigureMaker*.

**data: List[Union[np.ndarray, np.ma.MaskedArray]] = None**
List of data arrays (independents and one dependent)

**id: int = None**
unique ID of the plot item

**labels: Optional[List[str]] = None**
labels of the data arrays

**plotDataType: plottr.plot.base.PlotDataType = 1**
type of plot data (unknown is typically OK)

**plotOptions: Optional[Dict[str, Any]] = None**
options to be passed to plotting functions (depends on backend). Could be formatting options, for example.

**plotReturn: Optional[Any] = None**
return value from the plot command (like matplotlib Artists)

**subPlot: int = None**
ID of the subplot the item will be plotted in

**class** plottr.plot.base.**PlotNode**(*name*)

Basic Plot Node, derived from *plottr.node.node.Node*.

At the moment this doesn't do much besides passing data to the plotting widget. Data is just passed through. On receipt of new data, *newPlotData* is emitted.

**newPlotData**
Signal emitted when *process()* is called, with the data passed to it as argument.

**process**(*dataIn=None*)
Emits the *newPlotData* signal when called. Note: does not call the parent method *plottr.node.node.Node.process()*.

**Parameters** **dataIn** (Optional[*DataDictBase*]) – input data

**Return type** Dict[str, Optional[*DataDictBase*]]

**Returns** input data as is: {dataOut: dataIn}

**setPlotWidgetContainer**(*w*)
Set the plot widget container.

Makes sure that newly arriving data is sent to plot GUI elements.

**Parameters** **w** (*PlotWidgetContainer*) – container to connect the node to.

**Return type** None

**class** plottr.plot.base.**PlotWidget**(*parent=None*)

Base class for Plot Widgets, this just defines the API. Derived from *QWidget*.

Implement a child class for actual plotting.

**analyzeData**(*data*)
checks data and compares with previous properties.

**Parameters** **data** (Optional[*DataDictBase*]) – incoming data to compare to already existing data in the object.

**Return type** Dict[str, bool]

**Returns**

dictionary with information on what has changed from previous to new data. contains key/value pairs where the key is the property analyzed, and the value is True of False. Keys are:

- *dataTypeChanged* – has the data class changed?
- *dataStructureChanged* – has the internal structure (data fields, etc) changed?
- *dataShapesChanged* – have the data fields changed shape?
- *dataLimitsChanged* – have the maxima/minima of the data fields changed?

**dataIsComplex**(*dependentName=None*)

Determine whether our data is complex.

**Parameters** **dependentName** (Optional[str]) – name of the dependent to check. if *None*, check all.

**Return type** bool

**Returns** *True* if data is complex, *False* if not.

**setData**(*data*)

Set data. Use this to trigger plotting.

**Parameters** **data** (Optional[*DataDictBase*]) – data to be plotted.

**Return type** None

**class** plottr.plot.base.**PlotWidgetContainer**(*parent=None*)

This is the base widget for Plots, derived from *QWidget*.

This widget does not implement any plotting. It merely is a wrapping widget that contains the actual plot widget in it. This actual plot widget can be set dynamically.

Use *PlotWidget* as base for implementing widgets that can be added to this container.

**setData**(*data*)

set Data. If a plot widget is defined, call the widget's *PlotWidget.setData()* method.

**Parameters** **data** (*DataDictBase*) – input data to be plotted.

**Return type** None

**setPlotWidget**(*widget*)

Set the plot widget.

Makes sure that the added widget receives new data.

**Parameters** **widget** (*PlotWidget*) – plot widget

**Return type** None

**class** plottr.plot.base.**SubPlot**(*id: int*, *axes: Optional[List[Any]] = None*)

Data class describing a subplot in a *AutoFigureMaker*.

**axes: Optional[List[Any]] = None**

list of subplot objects (type depends on backend)

**id: int = None**
ID of the subplot (unique per figure)

plottr.plot.base.**determinePlotDataType**(*data*)
Analyze input data and determine most likely *PlotDataType*.

Analysis is simply based on number of dependents and data type.

**Parameters** **data** (Optional[*DataDictBase*]) – data to analyze.

**Return type** *PlotDataType*

**Returns** type of plot data inferred

plottr.plot.base.**makeFlowchartWithPlot**(*nodes*, *plotNodeName='plot'*)
create a linear FlowChart terminated with a plot node.

**Parameters**

- **nodes** (List[Tuple[str, Type[*Node*]]]) – List of Node classes, in the order they are to be arranged.
- **plotNodeName** (str) – name of the plot node that will be appended.

**Return type** Flowchart

**Returns** the resulting FlowChart instance

### 4.2.2 Matplotlib plotting tools

#### Overview

plottr.plot.mpl – matplotlib plotting system for plottr. Contains the following main objects:

#### Base UI elements

- *widgets.MPLPlot* (matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg) – Figure and Canvas widget. The most elementary Qt widget that contains the the matplotlib figure instance.
- *widgets.MPLPlotWidget* (*plottr.plot.base.PlotWidget*) – A widget that contains the figure/canvas

#### General plotting functionality

- *plotting.PlotType* – Enum for currently implemented plot types in automatic plotting.

#### Automatic plotting

- *autoplot.AutoPlot* (mpl.widgets.PlotWidget) – PlotWidget that allows user selection of plot types and plots using *autoplot.FigureMaker*.
- *autoplot.FigureMaker* (*base.AutoFigureMaker*) – Matplotlib implementation of the figure manager.

### Utilities

- *widgets.figureDialog()* – make a dialog window containing a plot widget.

### Configuration

This module looks for a file *plottr_default.mplstyle* in the plottr config directories and applies it to matplotlib plots using *pyplot.style.use*.

### Object Documentation

### General Widgets

`plottr.plot.mpl.widgets` – This module contains general matplotlib plotting tools.

**class** `plottr.plot.mpl.widgets.`**MPLPlot**(*parent=None*, *width=4.0*, *height=3.0*, *dpi=150*, *constrainedLayout=True*)

This is the basic matplotlib canvas widget we are using for matplotlib plots. This canvas only provides a few convenience tools for automatic sizing, but is otherwise not very different from the class `FCanvas` that comes with matplotlib (and which we inherit). It can be used as any QT widget.

**autosize**()

Sets some default spacings/margins.

**Return type** `None`

**clearFig**()

clear and reset the canvas.

**Return type** `None`

**resizeEvent**(*event*)

Re-implementation of the widget resizeEvent method. Makes sure we resize the plots appropriately.

**Return type** `None`

**setFigureInfo**(*info*)

Display an info string in the figure

**Return type** `None`

**setFigureTitle**(*title*)

Add a title to the figure.

**Return type** `None`

**setRcParams**()

apply matplotlibrc config from plottr configuration files.

**Return type** `None`

**setShowInfo**(*show*)

Whether to show additional info in the plot

**Return type** `None`

**toClipboard**()

Copy the current canvas to the clipboard.

**Return type** `None`

**class** plottr.plot.mpl.widgets.**MPLPlotWidget**(*parent=None*)
Base class for matplotlib-based plot widgets. Per default, add a canvas and the matplotlib NavBar.

**addMplBarOptions**()
Add options for displaying `info` meta data and copying the figure to the clipboard to the plot toolbar.

**Return type** None

**mplBar = None**
the matplotlib toolbar

**plot = None**
the plot widget

**setMeta**(*data*)
Add meta info contained in the data to the figure.

**Parameters** **data** (*DataDictBase*) – data object containing the meta information if meta field `title` or `info` are in the data object, then they will be added as text info to the figure.

**Return type** None

plottr.plot.mpl.widgets.**figureDialog**()
Make a dialog window containing a *MPLPlotWidget*.

**Return type** Tuple[Figure, QDialog]

**Returns** The figure object of the plot, and the dialog window object.

### General plotting tools

plottr.plot.mpl.plotting – Plotting tools (mostly used in Autoplot)

**class** plottr.plot.mpl.plotting.**PlotType**
Plot types currently supported in Autoplot.

**colormesh = 5**
colormesh plot of 2D data

**empty = 1**
no plot defined

**image = 4**
image plot of 2D data

**multitraces = 3**
multiple 1D lines/scatter plots per panel

**scatter2d = 6**
2D scatter plot

**singletraces = 2**
a single 1D line/scatter plot per panel

**class** plottr.plot.mpl.plotting.**SymmetricNorm**(*vmin=None*, *vmax=None*, *vcenter=0*, *clip=False*)
Color norm that's symmetric and linear around a center value.

plottr.plot.mpl.plotting.**colorplot2d**(*ax*, *x*, *y*, *z*, *plotType=<PlotType.image: 4>*, *axLabels=('', '', '')*, ***kw*)
make a 2d colorplot. what plot is made, depends on *plotType*. Any of the 2d plot types in *PlotType* works.

**Parameters**

- **ax** (Axes) – matplotlib subPlots to plot in
- **x** (Union[ndarray, MaskedArray]) – x coordinates (meshgrid)
- **y** (Union[ndarray, MaskedArray]) – y coordinates (meshgrid)
- **z** (Union[ndarray, MaskedArray]) – z data
- **plotType** (*PlotType*) – the plot type
- **axLabels** (Tuple[Optional[str], Optional[str], Optional[str]]) – labels for the x, y subPlots, and the colorbar.

all keywords are passed to the actual plotting functions, depending on the plotType:

- ***PlotType.image*** **–** *plotImage()*
- ***PlotType.colormesh*** **–** *ppcolormesh_from_meshgrid()*
- ***PlotType.scatter2d*** **–** matplotlib's *scatter*

**Return type** Optional[AxesImage]

plottr.plot.mpl.plotting.**plotImage**(*ax*, *x*, *y*, *z*, ***kw*)

Plot 2d meshgrid data as image.

**Parameters**

- **ax** (Axes) – matplotlib subPlots to plot the image in.
- **x** (ndarray) – x coordinates (as meshgrid)
- **y** (ndarray) – y coordinates
- **z** (ndarray) – z values

**Return type** AxesImage

**Returns** the image object returned by *imshow*

All keywords are passed to *imshow*.

plottr.plot.mpl.plotting.**ppcolormesh_from_meshgrid**(*ax*, *x*, *y*, *z*, ***kw*)

Plot a pcolormesh with some reasonable defaults. Input are the corresponding arrays from a 2D MeshgridDataDict.

Will attempt to fix missing points in the coordinates.

**Parameters**

- **ax** (Axes) – subPlots to plot the colormesh into.
- **x** (ndarray) – x component of the meshgrid coordinates
- **y** (ndarray) – y component of the meshgrid coordinates
- **z** (ndarray) – data values

**Return type** Optional[AxesImage]

**Returns** the image returned by *pcolormesh*.

Keywords are passed on to *pcolormesh*.

### Autoplot

`plottr.plot.mpl.autoplot` – This module contains the tools for automatic plotting with matplotlib.

**class** `plottr.plot.mpl.autoplot.`**AutoPlot**(*parent=None*)
A widget for plotting with matplotlib.

When data is set using *setData()* the class will automatically try to determine what good plot options are from the structure of the data.

User options (for different types of plots, styling, etc) are presented through a toolbar.

**setData**(*data*)
Analyses data and determines whether/what to plot.

**Parameters** **data** (Optional[*DataDictBase*]) – input data

**Return type** None

**class** `plottr.plot.mpl.autoplot.`**AutoPlotToolBar**(*name*, *parent=None*)
A toolbar that allows the user to configure AutoPlot.

Currently, the user can select between the plots that are possible, given the data that AutoPlot has.

**complexRepresentationSelected**
signal emitted when the complex data option has been changed

**plotTypeSelected**
signal emitted when the plot type has been changed

**selectComplexType**(*comp*)
makes sure that the selected *comp* is active (checked), all others are not active. This method should be used to catch a trigger from the UI. If the active plot type has been changed by using this method, we emit *complexPolarSelected*.

**Return type** None

**selectPlotType**(*plotType*)
makes sure that the selected *plotType* is active (checked), all others are not active.

This method should be used to catch a trigger from the UI.

If the active plot type has been changed by using this method, we emit *plotTypeSelected*.

**Parameters** **plotType** (*PlotType*) – type of plot

**Return type** None

**setAllowedComplexTypes**(**complexOptions*)
Disable all complex representation choices that are not allowed. If the current selection is now disabled, instead select the first enabled one.

**Return type** None

**setAllowedPlotTypes**(**args*)
Disable all plot type choices that are not allowed. If the current selection is now disabled, instead select the first enabled one.

**Parameters** **args** (*PlotType*) – which types of plots can be selected.

**Return type** None

**class** `plottr.plot.mpl.autoplot.`**FigureMaker**(*fig*)
Matplotlib implementation for *AutoFigureMaker*. Implements plotting routines for data with 1 or 2 dependents, as well as generation and formatting of subplots.

The class tries to lay out the subplots to be generated on a grid that's as close as possible to square. The allocation of plot to subplots depends on the type of plot we're making, and the type of data. Subplots may contain either one 2d plot (image, 2d scatter, etc) or multiple 1d plots.

**addData**(**data*, *join=None*, *labels=None*, *plotDataType=<PlotDataType.unknown: 1>*, ***plotOptions*)

Add data to the figure.

**Parameters**

- **data** (Union[ndarray, MaskedArray]) – data arrays describing the plot (one or more independents, one dependent)
- **join** (Optional[int]) – ID of a plot item the new item should be shown together with in the same subplot
- **labels** (Optional[List[str]]) – list of labels for the data arrays
- **plotDataType** (*PlotDataType*) – what kind of plot data the supplied data contains.
- **plotOptions** (Any) – options (as kwargs) to be passed to the actual plot functions (depends on the backend)

**Return type** int

**Returns** ID of the new plot item.

**formatSubPlot**(*subPlotId*)

Format a subplot. Parses the plot items that go into that subplot, and attaches axis labels and legend handles.

**Parameters** **subPlotId** (int) – ID of the subplot.

**Return type** None

**makeSubPlots**(*nSubPlots*)

Create subplots (*Axes*). They are arranged on a grid that's close to square.

**Parameters** **nSubPlots** (int) – number of subplots to make

**Return type** List[Axes]

**Returns** list of matplotlib axes.

**plot**(*plotItem*)

Plots data in a PlotItem.

**Parameters** **plotItem** (*PlotItem*) – the item to plot.

**Return type** Union[Artist, List[Artist], None]

**Returns** matplotlib Artist(s), or None if nothing was plotted.

**plotType = None**

what kind of plot we're making. needs to be set before adding data. Incompatibility with the data provided will result in failure.

## 4.3 Data format: DataDict

datadict.py :

Data classes we use throughout the plottr package, and tools to work on them.

**class** plottr.data.datadict.**DataDict**(**kw*)

The most basic implementation of the DataDict class.

It only enforces that the number of *records* per data field must be equal for all fields. This refers to the most outer dimension in case of nested arrays.

The class further implements simple appending of datadicts through the `DataDict.append` method, as well as allowing addition of DataDict instances.

**add_data**(**kw*)

Add data to all values. new data must be valid in itself.

This method is useful to easily add data without needing to specify meta data or dependencies, etc.

**Parameters** **kw** (`Any`) – one array per data field (none can be omitted).

**Return type** `None`

**Returns** None

**append**(*newdata*)

Append a datadict to this one by appending data values.

**Parameters** **newdata** (*DataDict*) – DataDict to append.

**Raises** `ValueError`, if the structures are incompatible.

**Return type** `None`

**expand**()

Expand nested values in the data fields.

Flattens all value arrays. If nested dimensions are present, all data with non-nested dims will be repeated accordingly – each record is repeated to match the size of the nested dims.

**Return type** *DataDict*

**Returns** The flattened dataset.

**Raises** `ValueError` if data is not expandable.

**is_expandable**()

Determine if the DataDict can be expanded.

Expansion flattens all nested data values to a 1D array. For doing so, we require that all data fields that have nested/inner dimensions (i.e, inside the *records* level) shape the inner shape. In other words, all data fields must be of shape (N,) or (N, (shape)), where shape is common to all that have a shape not equal to (N,).

**Return type** `bool`

**Returns** `True` if expandable. `False` otherwise.

**is_expanded**()

Determine if the DataDict is expanded.

**Return type** `bool`

**Returns** `True` if expanded. `False` if not.

**nrecords**()

**Return type** Optional[int]

**Returns** The number of records in the dataset.

**remove_invalid_entries**()
Remove all rows that are None or np.nan in *all* dependents.

**Return type** *DataDict*

**Returns** the cleaned DataDict.

**sanitize**()
Clean-up.

Beyond the tasks of the base class DataDictBase: * remove invalid entries as far as reasonable.

**Return type** *DataDict*

**Returns** sanitized DataDict

**validate**()
Check dataset validity.

Beyond the checks performed in the base class DataDictBase, check whether the number of records is the same for all data fields.

**Return type** bool

**Returns** True if valid.

**Raises** ValueError if invalid.

**class** plottr.data.datadict.**DataDictBase**(**kw*)
Simple data storage class that is based on a regular dictionary.

This base class does not make assumptions about the structure of the values. This is implemented in inheriting classes.

**add_meta**(*key*, *value*, *data=None*)
Add meta info to the dataset.

If the key already exists, meta info will be overwritten.

**Parameters**

- **key** (str) – Name of the meta field (without underscores)
- **value** (Any) – Value of the meta information
- **data** (Optional[str]) – if None, meta will be global; otherwise assigned to data field data.

**Return type** None

**astype**(*dtype*)
Convert all data values to given dtype.

**Parameters** **dtype** (dtype) – np dtype.

**Return type** ~T

**Returns** copy of the dataset, with values as given type.

**axes**(*data=None*)
Return a list of axes.

**Parameters** **data** (Union[Sequence[str], str, None]) – if None, return all axes present in the dataset, otherwise only the axes of the dependent data.

**Return type** List[str]

**Returns** the list of axes

**axes_are_compatible**()
Check if all dependent data fields have the same axes.

This includes axes order.

**Return type** bool

**Returns** True or False

**clear_meta**(*data=None*)
Delete meta information.

**Parameters** **data** (Optional[str]) – if this is not None, delete onlymeta information from data field *data*. Else, delete all top-level meta, as well as meta for all data fields.

**Return type** None

**copy**()
Make a copy of the dataset.

**Return type** ~T

**Returns** A copy of the dataset.

**data_items**()
Generator for data field items.

Like dict.items(), but ignores meta data.

**Return type** Iterator[Tuple[str, Dict[str, Any]]]

**data_vals**(*key*)
Return the data values of field key.

Equivalent to DataDict['key'].values.

**Parameters** **key** (str) – name of the data field

**Return type** ndarray

**Returns** values of the data field

**delete_meta**(*key*, *data=None*)
Remove meta data.

**Parameters**

- **key** (str) – name of the meta field to remove.
- **data** (Optional[str]) – if None, this affects global meta; otherwise remove from data field data.

**Return type** None

**dependents**()
Get all dependents in the dataset.

**Return type** List[str]

**Returns** a list of the names of dependents (data fields that have axes)

**extract**(*data*, *include_meta=True*, *copy=True*, *sanitize=True*)
Extract data from a dataset.

Return a new datadict with all fields specified in `data` included. Will also take any axes fields along that have not been explicitly specified.

**Parameters**

- **data** (`List[str]`) – data field or list of data fields to be extracted
- **include_meta** (`bool`) – if `True`, include the global meta data. data meta will always be included.
- **copy** (`bool`) – if `True`, data fields will be deep copies of the original.
- **sanitize** (`bool`) – if `True`, will run DataDictBase.sanitize before returning.

**Return type** ~T

**Returns** new DataDictBase containing only requested fields.

**has_meta**(*key*)
Check whether meta field exists in the dataset.

**Return type** `bool`

**label**(*name*)
Get a label for a data field.

If label is present, use the label for the data; otherwise fallback to use data name as the label. If a unit is present, this is the name with the unit appended in brackets: `name (unit)`; if no unit is present, just the name.

**Parameters** **name** (`str`) – name of the data field

**Return type** `Optional[str]`

**Returns** labelled name

**mask_invalid**()
Mask all invalid data in all values. :rtype: ~T :return: copy of the dataset with invalid entries (nan/None) masked.

**meta_items**(*data=None*, *clean_keys=True*)
Generator for meta items.

Like dict.items(), but yields *only* meta entries. The keys returned do not contain the underscores used internally.

**Parameters**

- **data** (`Optional[str]`) – if `None` iterate over global meta data. if it's the name of a data field, iterate over the meta information of that field.
- **clean_keys** (`bool`) – if *True*, remove the underscore pre/suffix

**Return type** `Iterator[Tuple[str, Dict[str, Any]]]`

**meta_val**(*key*, *data=None*)
Return the value of meta field `key` (given without underscore).

**Parameters**

- **key** (`str`) – name of the meta field
- **data** (`Optional[str]`) – `None` for global meta; name of data field for data meta.

**Return type** Any

**Returns** the value of the meta information.

**remove_unused_axes**()
Removes axes not associated with dependents.

**Return type** ~T

**Returns** cleaned dataset.

**reorder_axes**(*data_names=None*, ***pos*)
Reorder data axes.

**Parameters**

- **data_names** (Union[Sequence[str], str, None]) – data name(s) for which to reorder the axes if None, apply to all dependents.
- **pos** (int) – new axes position in the form axis_name = new_position. non-specified axes positions are adjusted automatically.

**Return type** ~T

**Returns** dataset with re-ordered axes.

**reorder_axes_indices**(*name*, ***pos*)
Get the indices that can reorder axes in a given way.

**Parameters**

- **name** (str) – name of the data field of which we want to reorder axes
- **pos** (int) – new axes position in the form axis_name = new_position. non-specified axes positions are adjusted automatically.

**Return type** Tuple[Tuple[int, ...], List[str]]

**Returns** the tuple of new indices, and the list of axes names in the new order.

**static same_structure**(**data*, *check_shape=False*)
Check if all supplied DataDicts share the same data structure (i.e., dependents and axes).

Ignores meta info and values. Checks also for matching shapes if *check_shape* is *True*.

**Parameters**

- **data** (~*T*) – the data sets to compare
- **check_shape** (bool) – whether to include a shape check in the comparison

**Return type** bool

**Returns** True if the structure matches for all, else False.

**sanitize**()
Clean-up tasks: * removes unused axes.

**Return type** ~T

**Returns** sanitized dataset.

**set_meta**(*key*, *value*, *data=None*)
Add meta info to the dataset.

If the key already exists, meta info will be overwritten.

**Parameters**

- **key** (str) – Name of the meta field (without underscores)
- **value** (Any) – Value of the meta information
- **data** (Optional[str]) – if None, meta will be global; otherwise assigned to data field data.

**Return type** None

**shapes**()
Get the shapes of all data fields.

**Return type** Dict[str, Tuple[int, ...]]

**Returns** a dictionary of the form {key : shape}, where shape is the np.shape-tuple of the data with name key.

**structure**(*add_shape=False*, *include_meta=True*, *same_type=False*)
Get the structure of the DataDict.

Return the datadict without values (*value* omitted in the dict).

**Parameters**

- **add_shape** (bool) – Deprecated – ignored.
- **include_meta** (bool) – if *True*, include the meta information in the returned dict, else clear it.
- **same_type** (bool) – if *True*, return type will be the one of the object this is called on. Else, DataDictBase.

**Return type** Optional[~T]

**Returns** The DataDict containing the structure only. The exact type is the same as the type of self

**static to_records**(***data*)
Convert data to rows that can be added to the DataDict. All data is converted to np.array, and the first dimension of all resulting arrays has the same length (chosen to be the smallest possible number that does not alter any shapes beyond adding a length-1 dimension as first dimesion, if necessary).

If a field is given as None, it will be converted to numpy.array([numpy.nan]).

**Return type** Dict[str, ndarray]

**validate**()
Check the validity of the dataset.

**Checks performed:**

- all axes specified with dependents must exist as data fields.

**Other tasks performed:**

- unit keys are created if omitted
- label keys are created if omitted
- shape meta information is updated with the correct values (only if present already).

**Return type** bool

**Returns** True if valid.

**Raises** ValueError if invalid.

**exception** plottr.data.datadict.**GriddingError**

**class** plottr.data.datadict.**MeshgridDataDict**(***kw*)

A dataset where the axes form a grid on which the dependent values reside.

This is a more special case than `DataDict`, but a very common scenario. To support flexible grids, this class requires that all axes specify values for each datapoint, rather than a single row/column/dimension.

For example, if we want to specify a 3-dimensional grid with axes x, y, z, the values of x, y, z all need to be 3-dimensional arrays; the same goes for all dependents that live on that grid. Then, say, x[i,j,k] is the x-coordinate of point i,j,k of the grid.

This implies that a `MeshgridDataDict` can only have a single shape, i.e., all data values share the exact same nesting structure.

For grids where the axes do not depend on each other, the correct values for the axes can be obtained from np.meshgrid (hence the name of the class).

Example: a simple uniform 3x2 grid might look like this; x and y are the coordinates of the grid, and z is a function of the two:

```
x = [[0, 0],
     [1, 1],
     [2, 2]]

y = [[0, 1],
     [0, 1],
     [0, 1]]

z = x * y =
    [[0, 0],
     [0, 1],
     [0, 2]]
```

Note: Internally we will typically assume that the nested axes are ordered from slow to fast, i.e., dimension 1 is the most outer axis, and dimension N of an N-dimensional array the most inner (i.e., the fastest changing one). This guarantees, for example, that the default implementation of np.reshape has the expected outcome. If, for some reason, the specified axes are not in that order (e.g., we might have `z` with `axes = ['x', 'y']`, but `x` is the fast axis in the data). In such a case, the guideline is that at creation of the meshgrid, the data should be transposed such that it conforms correctly to the order as given in the `axis = [...]` specification of the data. The function `datadict_to_meshgrid` provides options for that.

**reorder_axes**(*data_names=None*, ***pos*)

Reorder the axes for all data.

This includes transposing the data, since we're on a grid.

**Parameters** **pos** (`int`) – new axes position in the form `axis_name = new_position`. non-specified axes positions are adjusted automatically.

**Return type** *MeshgridDataDict*

**Returns** Dataset with re-ordered axes.

**shape**()

Return the shape of the meshgrid.

**Return type** `Optional[Tuple[int, ...]]`

**Returns** the shape as tuple. None if no data in the set.

**validate**()

Validation of the dataset.

Performs the following checks: * all dependents must have the same axes * all shapes need to be identical

**Return type** `bool`

**Returns** `True` if valid.

**Raises** `ValueError` if invalid.

`plottr.data.datadict.`**`combine_datadicts`**(**dicts*)

Try to make one datadict out of multiple.

Basic rules:

- we try to maintain the input type
- return type is 'downgraded' to DataDictBase if the contents are not compatible (i.e., different numbers of records in the inputs)

**Return type** `Union[DataDictBase, DataDict]`

**Returns** combined data

`plottr.data.datadict.`**`datadict_to_meshgrid`**(*data*, *target_shape=None*, *inner_axis_order=None*, *use_existing_shape=False*)

Try to make a meshgrid from a dataset.

**Parameters**

- **`data`** (*DataDict*) – input DataDict.
- **`target_shape`** (`Optional[Tuple[int, ...]]`) – target shape. if `None` we use `guess_shape_from_datadict` to infer.
- **`inner_axis_order`** (`Optional[List[str]]`) – if axes of the datadict are not specified in the 'C' order (1st the slowest, last the fastest axis) then the 'true' inner order can be specified as a list of axes names, which has to match the specified axes in all but order. The data is then transposed to conform to the specified order. **Note**: if this is given, then *target_shape* needs to be given in in the order of this inner_axis_order. The output data will keep the axis ordering specified in the *axes* property.
- **`use_existing_shape`** (`bool`) – if `True`, simply use the shape that the data already has. For numpy-array data, this might already be present. if `False`, flatten and reshape.

**Raises** GriddingError (subclass of ValueError) if the data cannot be gridded.

**Return type** *MeshgridDataDict*

**Returns** the generated `MeshgridDataDict`.

`plottr.data.datadict.`**`datasets_are_equal`**(*a*, *b*, *ignore_meta=False*)

Check whether two datasets are equal.

Compares type, structure, and content of all fields.

**Parameters**

- **`a`** (*DataDictBase*) – first dataset
- **`b`** (*DataDictBase*) – second dataset
- **`ignore_meta`** (`bool`) – if `True`, do not verify if metadata matches.

**Return type** `bool`

**Returns** `True` or `False`

plottr.data.datadict.**datastructure_from_string**(*description*)
Construct a DataDict from a string description.

### Examples

- `"data[mV](x, y)"` results in a datadict with one dependent `data` with unit `mV` and two independents, `x` and `y`, that do not have units.
- `"data_1[mV](x, y); data_2[mA](x); x[mV]; y[nT]"` results in two dependents, one of them depening on `x` and `y`, the other only on `x`. Note that `x` and `y` have units. We can (but do not have to) omit them when specifying the dependencies.
- `"data_1[mV](x[mV], y[nT]); data_2[mA](x[mV])"`. Same result as the previous example.

We recognize descriptions of the form `field1[unit1](ax1, ax2, ...); field1[unit2](...); ...`.

- field names (like `field1` and `field2` above) have to start with a letter, and may contain word characters
- field descriptors consist of the name, optional unit (presence signified by square brackets), and optional dependencies (presence signified by round brackets).
- dependencies (axes) are implicitly recognized as fields (and thus have the same naming restrictions as field names)
- axes are separated by commas
- axes may have a unit when specified as dependency, but besides the name, square brackets, and commas no other characters are recognized within the round brackets that specify the dependency
- in addition to being specified as dependency for a field, axes may be specified also as additional field without dependency, for instance to specify the unit (may simplify the string). For example, `z1[x, y]; z2[x, y]; x[V]; y[V]`
- units may only consist of word characters
- use of unexpected characters will result in the ignoring the part that contains the symbol
- the regular expression used to find field descriptors is: `((?<=\A)|(?<=\;))[a-zA-Z]+\w*(\[\w*\])?(\(([a-zA-Z]+\w*(\[\w*\])?\,?)*\))?`

**Return type** *DataDict*

plottr.data.datadict.**guess_shape_from_datadict**(*data*)
Try to guess the shape of the datadict dependents from the axes values.

**Parameters** **data** (*DataDict*) – dataset to examine.

**Return type** `Dict[str, Optional[Tuple[List[str], Tuple[int, ...]]]]`

**Returns** a dictionary with the dependents as keys, and inferred shapes as values. value is None, if the shape could not be inferred.

plottr.data.datadict.**meshgrid_to_datadict**(*data*)
Make a DataDict from a MeshgridDataDict by reshaping the data.

**Parameters** **data** (*MeshgridDataDict*) – input `MeshgridDataDict`

**Return type** *DataDict*

**Returns** flattened `DataDict`

plottr.data.datadict.**str2dd**(*description*)
shortcut to *datastructure_from_string()*.

**Return type** *DataDict*

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

# INDEX

## P



## R



## S

## T



## U



## V