Python bindings for NIX

Release 1.4.8 Release

May 22, 2019

Contents

The nixpy module contains the python binding to the nix c++ library. In order to use it you also have to have the nix library installed.

CHAPTER 1

Getting started

The following sections should help you getting started with nixpy.

1.1 Installation Guidelines

1.1.1 Dependencies

NIXPy is a reimplementation of the NIX library and file format for Python. Since NIX is built upon HDF5, NIXPy depends on h5py, the Python interface to the HDF5 data format.

NIXPy can also be used as an interface for the original NIX library. NIX is therefore an optional dependency, if NIXPy is to be used in this mode.

The installation instructions below describe how to build or install NIXPy as a standalone, pure Python version of NIX. To use NIXPy as a Python interface for NIX, see the *advanced installation* instructions.

Dependencies:

- h5py: http://docs.h5py.org/en/latest/build.html
- numpy: https://docs.scipy.org/doc/numpy-1.10.1/user/install.html

Optional dependencies (see *advanced installation* instructions):

- NIX: https://github.com/G-Node/nix

1.1.2 Installation instructions

The latest stable release of NIXPy is available on PyPi as nixio. Therefore, the simplest way to install NIXPy is to use pip:

```
pip install nixio
```

1.1.3 Advanced installation

To use NIXPy as a Python interface for the NIX library it is necessary to install the NIX library before you can install or build NIXPy on your system. The following two sections explain how NIX can be installed on Windows and Linux.

Linux

If you use the latest Ubuntu LTS you can install NIX from our PPA on launchpad. First you have to add the PPA to your system:

```
sudo add-apt-repository ppa:gnode/nix
sudo apt-get update
```

Afterwards you can use apt to install the NIX package:

```
sudo apt-get install libnix-dev
```

If you want to use NIX on other distributions you have to compile and install NIX from source (*see below*).

Windows

To install NIX under Windows it is recommended to use the latest installer. The installer can be downloaded from the nix releases on GitHub.

Build NIX from Source

In order to build and install NIX from source please follow the build instructions in the NIX repository. Comprehensive build instructions for Linux can be found in the nix README.rst. For Windows this information can be found in the nix Win32.md file.

Install NIXPy

Once the NIX library is installed on your system you can proceed with the installation of the python bindings.

Compatibility

The NIX library as well as NIXPy undergo continuous development and improvement. Although most changes do not affect the NIX API, the compatibility between the NIX library and their bindings might still break from time to time. Therefore it is worth mentioning which assumptions can be made concerning compatibility between versions of the NIX projects.

- The head of the master branches of the NIX library and the bindings are usually compatible to each other.
- Nix releases of the same version and their corresponding tags in the repositories are always compatible with each other e.g. NIXPy 1.0.x is compatible with libnix 1.0.x etc.

Linux

If you use the latest Ubuntu LTS you can install NIXPy the same was as shown above for NIX from our PPA on launchpad. If the PPA was not already added to your system, you can do so by executing the following commands::

```
sudo add-apt-repository ppa:gnode/nix
sudo apt-get update
```

Once the PPA was added NIXPy can be installed via apt-get:

```
sudo apt-get install python-nix
```

If you want to use NIXPy on other distributions, please follow the instructions for building NIXPy from source (*see below*).

Windows

To install NIXPy under Windows it is recommended to use the Windows installer. Download the installer with the same version as your NIX installation from the NIXPy releases and execute the installer. In addition NIXPy requires numpy to be installed.

Build NIXPy from Source

If you want to use the latest development version or in cases where the provided installers or packages can't be used, it is possible to build and install NIXPy from sources. Instructions for building NIX on Linux can be found in the NIXPy README.rst file. For the Windows platform those instructions are described in the NIXPy Win32.md file.

1.2 Overview

1.2.1 Design Principles

The design of the data model tries to draw on similarities of different data types and structures and and come up with *entities* that are as generic and versatile as meaningful. At the same time we aim for clearly established links between different entities to keep the model as expressive as possible.

Most entities of the NIX-model have a *name* and a *type* field which are meant to provide information about the entity. While the name can be freely chose, the type is meant to provide semantic information about the entity and we aim at definitions of different types. Via the type, the generic entities can become domain specific.

For the electrophysiology disciplines of the neuroscience, an INCF working groups has set out to define such data types. For more information see here

Creating a file

So far we have implemented the nix model only for the HDF5 file format. In order to store data in a file we need to create one.

```
import nixio as nix

nix_file = nix.File.open('example.h5', nix.FileMode.Overwrite)
```

The **File** entity is the root of this document and it has only two children the *data* and *metadata* nodes. You may want to use the hdfview tool to open the file and look at it. Of course you can access both parts using the **File** API.

All information directly related to a chunk of data is stored in the *data* node as children of a top-level entity called **Block**. A **Block** is a grouping element that can represent many things. For example it can take up everything that was recorded in the same *session*. Therefore, the **Block** has a *name* and a *type*.

```
block = nix_file.create_block("Test block", "nix.session")
```

Names can be freely chosen. Duplication of names on the same hierarchy-level is not allowed. In this example creating a second **Block** with the very same name leads to an error. Names must not contain '/' characters since they are path separators in the HDF5 file. To avoid collisions across files every created entity has an unique id (UUID).

```
block.id
'017d7764-173b-4716-a6c2-45f6d37ddb52'
```

Storing data

The heart of our data model is an entity called **DataArray**. This is the entity that actually stores all data. It can take n-dimensional arrays and provides sufficient information to create a basic plot of the data. To achieve this, one essential part is to define what kind of data is stored. Hence, every dimension of the stored data **must** be defined using the available Dimension descriptors (below). The following code snippets show how to create a **DataArray** and how to store data in it.

```
# create a DataArray and store data in it
data = block.create_data_array("my data", "nix.sampled", data=some_numpy_array)
```

Using this call will create a **DataArray**, set name and type, set the *dataType* according to the dtype of the passed data, and store the data in the file. You can also create empty **DataArrays** to take up data-to-be-recorded. In this case you have to provide the space that will be needed in advance.

```
import numpy as np
# create an empty DataArray to store 2x1000 values
data = block.create_data_array("my data", "nix.sampled", dtype=nix.DataType.Double,␣
↪shape=(2, 1000))
some_numpy_array = np.random.randn(2, 1000)
data.write_direct(some_numpy_array)
```

If you do not know the size of the data in advance, you can append data to an already existing **DataArray**. **Beware:** Though it is possible to extend the data, it is not possible to change the dimensionality (rank) of the data afterwards.

```
# create an empty DataArray to store 2x1000 values
data = block.create_data_array("my data", "nix.sampled", dtype=nix.DataType.Double,␣
↪shape=(2, 1000))
some_numpy_array = np.random.randn(2, 1000)
data[:, :] = some_numpy_array
some_more_data = np.random.randn(2, 10)
data.data_extent((2, 1010))
data[:, 1000:] = some_more_data
```

Dimension descriptors

In the above examples we have created **DataArray** entities that are used to store the data. The goal of our model design is that the data containing structures carry enough information to create a basic plot. Let's assume a time-series of data needs to be stored: The data is just a vector of measurements (e.g. voltages). The data would be plotted as a line-plot. We thus need to define the x- and the y-axis of the plot. The y- or value axis is defined by setting the label and the unit properties of the **DataArray**, the x-axis needs a dimension descriptor. In the nix model three different dimension descriptors are defined. **SampledDimension**, **RangeDimension**, and **SetDimension** which are used for (i) data that has been sampled in space or time in regular intervals, (ii) data that has been sampled in irregular intervals, and (iii) data that belongs to categories.

```
sample_interval = 0.001 # s
sinewave = np.sin(np.arange(0, 1.0, sample_interval) * 2 * np.pi)
data = block.create_data_array("sinewave", "nix.regular_sampled", data=sinewave)
data.label = "voltage"
data.unit = "mV"
# define the time dimension of the data
dim = data.append_sampled_dimension(sample_interval)
dim.label = "time"
dim.unit = "s"
```

The **SampledDimension** can also be used to desribe space dimensions, e.g. in case of images.

If the data was sampled at irregular intervals the sample points of the x-axis are defined using the *ticks* property of a **RangeDimension**.

```
sample_times = [1.0, 3.0, 4.2, 4.7, 9.6]
dim = data.append_range_dimension(sample_times)
dim.label = "time"
dim.unit = "s"
```

Finally, some data belongs into categories which do not necessarily have a natural order. In these cases a **SetDimension** is used. This descriptor can store for each category an optional label.

```
observations = [0, 0, 5, 20, 45, 40, 28, 12, 2, 0, 1, 0]
categories = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
              'Jul', 'Aug','Sep','Oct','Nov', 'Dec']
data = block.create_data_array("observations", "nix.histogram", data=observations)
dim = data.append_set_dimension()
dim.labels = categories
```

Annotate regions in the data

Annotating points of regions of interest is one of the key features of the nix data-model. There are two entities for this purpose: (i) the **Tag** is used for single points or regions while the (ii) **MultiTag** is used to mark multiple of these. Tags have one or many *positions* and *extents* which define the point or the region in the *referenced* DataArray. Further they can have **Features** to store additional information about the positions (see tutorials below).

Tag

The tag is a relatively simple structure directly storing the *position* the tag points and the, optional, *extent* of this region. Each of these are vectors of a length matching the dimensionality of the referenced data.

```
position = [10, 10]
extent = [5, 20]
tag = block.create_tag('interesting part', 'nix.roi', position)
tag.extent = extent
# finally, add the referenced data to this tag
tag.references.append(data)
```

MuliTag

MultiTags are made to tag multiple points (regions) at once. The main difference to the **Tag** is that position and extent are stored in **DataArray** entities. These entities **must** be 2-D. Both dimensions are *SetDimensions*. The

first dimension represents the individual positions, the second dimension takes the coordinates in the referenced n-dimensional **DataArray**.

```
# fake data
frame = np.random.randn(100, 100)
data = block.create_data_array('random image', 'nix.image', data=frame)
dim_x = data.append_sampled_dimension(1.0)
dim_x.label = 'x'
dim_y = data.append_sampled_dimension(1.0)
dim_y.label = 'y'
# positions array must be 2D
p = np.zeros((3, 2)) # 1st dim, represents the positions, 2nd the coordinates
p[1, :] = [10, 10]
p[2, :] = [20, 10]
positions = block.create_data_array('special points', 'nix.positions', data=p)
positions.append_set_dimension()
dim = positions.append_set_dimension()
dim.labels = ['x', 'y']
# create a multi tag
tag = block.create_multi_tag('interesting points', 'nix.multiple_roi', positions)
tag.references.append(data)
```

Adding further information

The tags establish links between datasets. If one needs to attach further information to each of the regions defined by the tag, one can add **Features** to them. A **Feature** references a **DataArray** as its *data* and specifies with the *link_type* how the link has to be interpreted. The *link_type* can either be *tagged*, *indexed*, or *untagged* indicating that the tag should be applied also to the feature data (*tagged*), for each position given in the tag, a slice of the feature data (ith index along the first dimension) is the feature (*indexed*), or all feature data applies for all positions (*untagged*).

Let's say we want to give each point a name, we can create a feature like this:

```
spot_names = block.create_data_array('spot ids', 'nix.feature', dtype=nix.DataType.
↪Int8, data=[1, 2])
spot_names.append_set_dimension()
feature = tag.create_feature(spot_names, nix.LinkType.Indexed)
```

We could also say that each point in the tagged data (e.g. a matrix of measurements) has a corresponding point in an input matrix.

```
input_matrix = np.random.random(data.shape)
input_data = block.create_data_array('input matrix', 'nix.feature', data=input_matrix)
dim_x = input_data.append_sampled_dimension(1.0)
dim_x.label = 'x'
dim_y = input_data.append_sampled_dimension(1.0)
dim_y.label = 'y'
tag.create_feature(input_data, nix.LinkType.Tagged)
```

Finally, one could need to attach the same information to all positions defined in the tag. In this case the feature is *untagged*

```
common_feature = block.create_data_array('common feature', 'nix.feature', data=some_
↪common_data)
tag.create_feature(common_feature, nix.LinkType.Untagged)
```

Defining the Source of the data

In cases in which we want to store where the data originates **Source** entities can be used. Almost all entities of the NIX-model can have **Sources**. For example, if the recorded data originates from experiments done with one specific experimental subject. **Sources** have a name and a type and can have some definition.

```
subject = block.create_source('subject A', 'nix.experimental_subject')
subject.definition = 'The experimental subject used in this experiment'
data.sources.append(subject)
```

Sources may depend on other **Sources**. For example, in an electrophysiological experiment we record from different cells in the same brain region of the same animal. To represent this hierarchy, **Sources** can be nested, create a tree-like structure.

```
subject = block.create_source('subject A', 'nix.experimental_subject')
brain_region = subject.create_source('hippocampus', 'nix.experimental_subject')
cell_a = brain_region.create_source('Cell 1', 'nix.experimental_subject')
cell_b = brain_region.create_source('Cell 2', 'nix.experimental_subject')
```

Arbitrary metadata

The entities discussed so far carry just enough information to get a basic understanding of the stored data. Often much more information than that is required. Storing additional metadata is a central part of the NIX concept. We use a slightly modified version of the *odML* data model for metadata to store additional information. In brief: the model consists of **Sections** that contain **Properties** which in turn contain one or more **Values**. Again, **Sections** can be nested to represent logical dependencies in the hierarchy of a tree. While all data entities discussed above are children of **Block** entities, the metadata lives parallel to the **Blocks**. The idea behind this is that several blocks may refer to the same metadata, or, the other way round the metadata applies to data entities in several blocks. The *types* used for the **Sections** in the following example are defined in the odml terminologies

Most of the data entities can link to metadata sections.

```
sec = nix_file.create_section('recording session', 'odml.recording')
sec.create_property('experimenter', nix.Value('John Doe'))
sec.create_property('recording date', nix.Value('2014-01-01'))
subject = sec.create_section('subject', 'odml.subject')
subject.create_property('id', nix.Value('mouse xyz'))
cell = subject.create_section('cell', 'odml.cell')
v = nix.Value(-64.5)
v.uncertainty = 2.25
p = cell.create_property('resting potential', v)
p.unit = 'mV'
# set the recording block metadata
block.metadata = sec
```

Units

In NIX we accept only SI units (plus dB, %) wherever units can be given. We also accept compound units like *mV/cm*. Units are most of the times handled transparently. That is, when you tag a region of data that has been specified with a time axis in seconds and use e.g. the *tag.retrieve_data* method to get this data slice, the API will handle unit scaling. The correct data will be returned even if the tag's position is given in *ms*.

```
x_positions=[2, 4, 6, 8, 10, 12]
tag=block.create_tag('unit example', 'nix.sampled', x_positions)
```

(continues on next page)

(continued from previous page)

```
#single SI unit is supported like mV,cm etc.
tag.units=["cm"]

#for compound units we can do
tag.units=["mV/cm"]
```

1.3 Tutorials

The following tutorials show how to work with NIX files, to store different kinds of data, tag points or regions of interest and add further information to the data.

1.3.1 List of Tutorials

- Working with files
  - *Working with Files*
- Basic data structures
  - *Regularly sampled data*
  - *Irregularly sampled data*
  - *Event data*
  - *Series of signals*
  - *Image data*
- Tagging points and regions-of-interest
  - *Single point or region*
  - *Multiple points or regions*
  - *Tagging spikes in membrane potential*
- Features
  - *Untagged Feature*
  - *Tagged Feature*
  - *Indexed Feature*
- Retrieve data
  - *Retrieving tagged regions*
  - *Retrieving feature data*
- Additional information
  - *Storing the origin of data*
  - *Adding arbitrary metadata*

1.3.2 Working with Files

The following code shows how to create a new nix-file, close it and re-open them with different access rights (examples/fileCreate.py).

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function

import nixio as nix


file_name = 'file_create_example.h5'

# create a new file overwriting any existing content
nixfile = nix.File.open(file_name, nix.FileMode.Overwrite)
print(nixfile.format, nixfile.version, nixfile.created_at)

# close file
nixfile.close()

# re-open file for read-only access
nixfile = nix.File.open(file_name, nix.FileMode.ReadOnly)

# this command will fail putting out HDF5 Errors
try:
    nixfile.create_block("test block", "test")
except ValueError:
    print("Error caught: cannot create a new group in nix.FileMode.ReadOnly mode")


nixfile.close()

# re-open for read-write access
nixfile = nix.File.open(file_name, nix.FileMode.ReadWrite)

# the following command now works fine
nixfile.create_block("test block", "test")

nixfile.close()
```

Source code of this example: fileCreate.py.

Selecting a backend

The *open* method supports specifying a NIX backend with the *backend* argument. The H5Py backend is always available.

```
file = nix.File.open(file_name, nix.FileMode.Overwrite, backend="h5py")
```

Alternatively, if NIX is installed and NIXPy was built with NIX support, the HDF5 backend can be specified.

```
file = nix.File.open(file_name, nix.FileMode.Overwrite, backend="hdf5")
```

See the *Advanced installation* instructions for details on installing NIXPy with NIX HDF5 backend support.

When no backend is specified, HDF5 is used if available, otherwise the library defaults to H5Py.

List of Tutorials

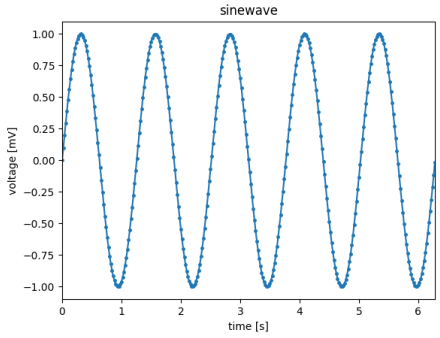
1.3.3 Basic data structures

In this section we will show how different kinds of data are stored in nix files. We will start with simple regularly and irregularly sampled signals, turn to series of such signals and end with images stacks.

Regularly sampled data

Regularly sampled data is everything which is sampled in regular intervals in space, time, or something else. Let's consider a signal that has been digitized using an AD-Converter at a fixed sampling rate. In this case the axis representing time has to be described using a **SampledDimension**. This dimension descriptor contains as mandatory element the *sampling_interval*. The *sampling_interval* has to be given because it also applies e.g. to spatial sampling, it is the more general term than the sampling rate which may appear appropriate for time discretization. Further, the *unit* in which this number has to be interpreted and a *label* for the axis can be specified. The following code illustrates how this is stored in nix files.

```
    # create a 'Block' that represents a grouping object. Here, the recording session.
    # it gets a name and a type
    block = file.create_block("block name", "nix.session")

    # create a 'DataArray' to take the sinewave, add some information about the signal
    data = block.create_data_array("sinewave", "nix.regular_sampled", data=y)
    data.unit = "mV"
    data.label = "voltage"
    # add a descriptor for the xaxis
    dim = data.append_sampled_dimension(stepsize)
    dim.unit = "s"
    dim.label = "time"
    dim.offset = 0.0 # optional
```



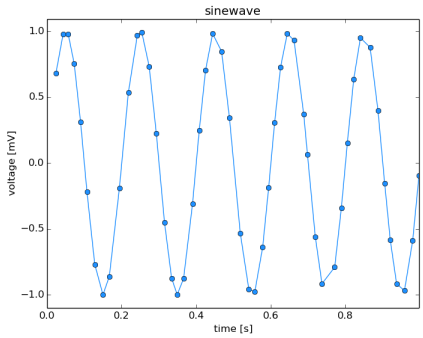
Source code for this example `regularlySampledData.py`.

List of Tutorials

Irregularly sampled data

Irregularly sampled data is sampled at irregular intervals. The dimension which is sampled in this way has to be described using a **RangeDimension**. This dimension descriptor stores besides the *unit* and *label* of the axis the ticks, e.g. time-stamps of the instances at which the samples were taken.

```
    data = block.create_data_array("sinewave", "nix.irregular_sampled", data=y)
    data.unit = "mV"
    data.label = "voltage"
    # add a descriptor for the xaxis
    dim = data.append_range_dimension(times)
    dim.unit = "s"
    dim.label = "time"
```



Source code for this example `irregularlySampledData.py`.

List of Tutorials

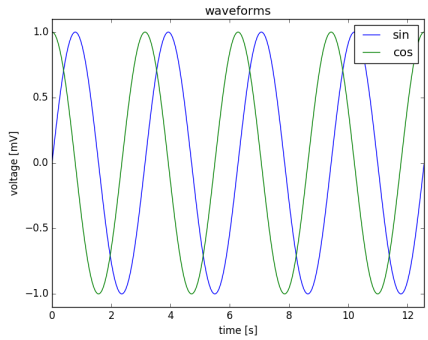
Event data

TODO

Series of signals

It is possible to store multiple signals that have the same shape and logically belong together in the same *DataArray* object. In this case, the data is two-dimensional and two dimension-descriptors are needed. Depending on the layout of the data one dimension represents time and is described with a *SampledDimension* while the other represents the various signals. This is described with a *SetDimension*. A *SetDimension* can have labels for each entry along this dimension of the data.

```
    y = np.vstack((sine, cosine))
    data = block.create_data_array("waveforms", "nix.regular_sampled.multiple_series",
↪ data=y)
    data.unit = "mV"
    data.label = "voltage"
    # descriptor for first dimension is a set
    set_dim = data.append_set_dimension()
    set_dim.labels = ['sin', 'cos']
    # add a descriptor for the xaxis
    dim = data.append_sampled_dimension(stepsize)
    dim.unit = "s"
    dim.label = "time"
```



Source code for this example: `multipleTimeSeries.py`.

List of Tutorials

Image data

Color images can be stored as 3-D data in a *DataArray*. The first two dimensions represent *width* and *height* of the image while the 3rd dimension represents the color channels. Accordingly, we need three dimension descriptors. The first two are *SampledDimensions* since the pixels of the image are regularly sampled in space. The third dimension is a *SetDimension* with labels for each of the channels. In this tutorial the "Lenna" image is used. Please see the author attribution in the code.

```
    # create a 'DataArray' to take the sinewave, add some information about the signal
    data = block.create_data_array("lenna", "nix.image.rgb", data=img_data)
    # add descriptors for width, height and channels
    height_dim = data.append_sampled_dimension(1)
    height_dim.label = "height"
    width_dim = data.append_sampled_dimension(1)
    width_dim.label = "width"
    color_dim = data.append_set_dimension()
```

if the image is not shown install *imagemagick* or *xv* tools (Linux) Source code for this example: `imageData.py`.

List of Tutorials

1.3.4 Tagging regions

One key feature of the nix-model is its ability to annotate, or "tag", points or regions-of-interest in the stored data. This feature can be used to state the occurrence of events during the recording, to state the intervals of a certain condition, e.g. a stimulus presentation, or to mark the regions of interests in image data. In the nix data-model two types of Tags are discriminated. (1) the **Tag** for single points or regions, and (2) the **MultiTag** to annotate multiple points or regions using the same entity.

Single point or region

Single points of regions-of-interest are annotated using a **Tag** object. The Tag contains the start *position* and, optional, the *extent* of the point or region. The link to the data is established by adding the **DataArray** that contains the data to the list of references. It is important to note that *position* and *extent* are arrays with the length matching the dimensionality of the referenced data. The same Tag can be applied to many references as long as *position* and *extent* can be applied to these.

```
# create a Tag, position and extent must be 3-D since the data is 3-D
position = [250, 250, 0]
extent = [30, 100, 3]
tag = block.create_tag('Region of interest', 'nix.roi', position)
tag.extent = extent
```

Source code for this example `singleROI.py`.

List of Tutorials

Multiple points or regions

For tagging multiple regions of interest in the same data the **MultiTag** object is used. Unlike the simple **Tag** from the previous example, the multiple *positions* and *extents* can be given. These are stored in **DataArray** objects. The tagged dataset is linked via the references. There are some restrictions regarding the **DataArrays** storing positions and extents. The data stored in them **must** be 2-dimensional. Both dimensions are **SetDimensions** representing the individual positions and the positions in the referenced data, respectively. Thus, the second dimension has as many entries as the referenced data has dimensions.

In the following example we will declare multiple ROIs in a image. The image as a spatial extent and three color channels, is hence 3-D. The same mechanism can, of course, be used to tag other event in different kinds of data. For example in the neuroscience context: the detection of action potentials in a recorded membrane potential.

```
    # some space for three regions-of-interest
    roi_starts = np.zeros((3, 3), dtype=int)
    roi_starts[0, :] = [250, 245, 0]
    roi_starts[1, :] = [250, 315, 0]
    roi_starts[2, :] = [340, 260, 0]

    roi_extents = np.zeros((3, 3), dtype=int)
    roi_extents[0, :] = [30, 45, 3]
    roi_extents[1, :] = [30, 40, 3]
    roi_extents[2, :] = [25, 65, 3]

    # create the positions DataArray
    positions = block.create_data_array("ROI positions", "nix.positions", data=roi_
↪starts)
```

Source code for this example `multipleROIs.py`.

List of Tutorials

Tagging spikes in membrane potential

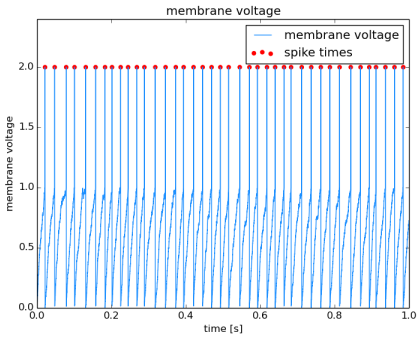
Neuroscience example. The same construct as above is used to mark the times at which action potentials were detected in the recording of a neuron's membrane potential.

```
    data = block.create_data_array("membrane voltage", "nix.regular_sampled.time_
↪series", data=voltage)
    data.label = "membrane voltage"
    # add descriptors for time axis
    time_dim = data.append_sampled_dimension(time[1]-time[0])
    time_dim.label = "time"
    time_dim.unit = "s"

    # create the positions DataArray
    positions = block.create_data_array("times", "nix.positions", data=spike_times)
    positions.append_set_dimension() # these can be empty
    positions.append_set_dimension()

    # create a MultiTag
```

(continues on next page)

(continued from previous page)

```
    multi_tag = block.create_multi_tag("spike times", "nix.events.spike_times",␣
↪positions)
    multi_tag.references.append(data)
```



Source code for this example: `spikeTagging.py`.

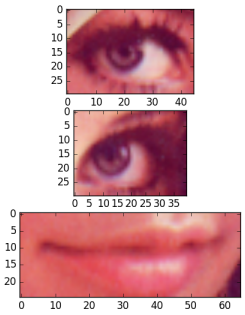
List of Tutorials

Retrieving tagged regions

Tagging regions of interest in one thing but retrieving the tagged data slice is another. The **Tag** and **MultiTag** entities offer a function for this. Consider the image example from above:

Three regions were tagged. To retrieve the respective data the following code has to be executed:

```
def plot_roi_data(tag):
    position_count = tag.positions.shape[0]
    for p in range(position_count):
```



Source code for this example: `multipleROIs.py`.

List of Tutorials

Unit support in tagging

TODO

List of Tutorials

1.3.5 Features

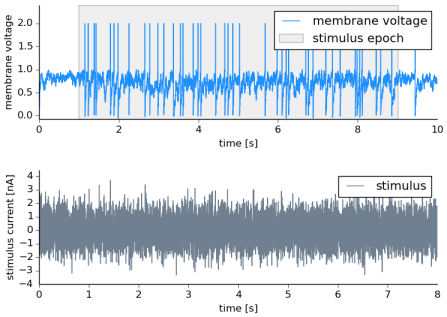
The following code shows how to use the **Features** of the NIX-model. Suppose that we have the recording of a signal in which a set of events is detected. Each event may have certain characteristics one wants to store. These are stored as **Features** of the events. There are three different link-types between the features and the events stored in the tag. *nix.LinkType.Untagged* indicates that the whole data stored in the **Feature** applies to the points defined in the tag. *nix.LinkType.Tagged* on the other side implies that the *position* and *extent* have to be applied also to the data stored in the **Feature**. Finally, the *nix.LinkType.Indexed* indicates that there is one point (or slice) in the **Feature** data that is related to each position in the Tag.

The following examples show how this works.

Untagged Feature

Let's say we record the activity of a neuron and at a certain epoch of that recording a stimulus was presented. This time interval is annotated using a **Tag**. This inidicates the time in which the stimulus was on but we may also want to link the stimulus trace to it. The stimulus is also stored as a **DataArray** in the file and can be linked to the stimulus interval as an *untagged* **Feature** of it.

```
    time_dim.label = "time"
    time_dim.unit = "s"

    # create a Tag
    tag = block.create_tag("stimulus presentation", "nix.epoch.stimulus_presentation",
↪ [stim_onset])
    tag.extent = [stim_duration]
    tag.references.append(data)
```



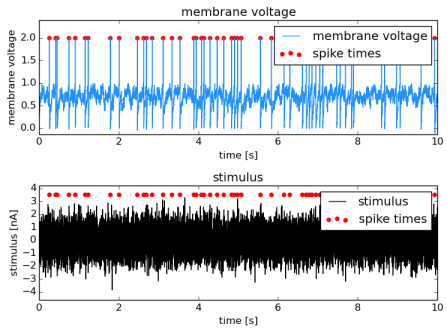
Source code for this example: `untaggedFeature.py`.

List of Tutorials

Tagged Feature

Tagged **Features** are used in cases in which the positions and extents of a tag also apply to another linked dataset. In the following example the spike times should also be applied to the stimulus that led to the responses. The stimulus is saved in an additional **DataArray** and is linked to the spike times as a **Feature** setting the **LinkType** to *tagged*.

```
    stimulus_array = block.create_data_array("stimulus", "nix.regular_sampled",␣
↪data=stimulus)
    stimulus_array.label = "stimulus"
    stimulus_array.unit = "nA"
    # add a descriptor for the time axis
    dim = stimulus_array.append_sampled_dimension(stepsize)
    dim.unit = "s"
    dim.label = "time"

    # set stimulus as a tagged feature of the multi_tag
    multi_tag.create_feature(stimulus_array, nix.LinkType.Tagged)

    # let's plot the data from the stored information
    plot_data(multi_tag)
    file.close()
```



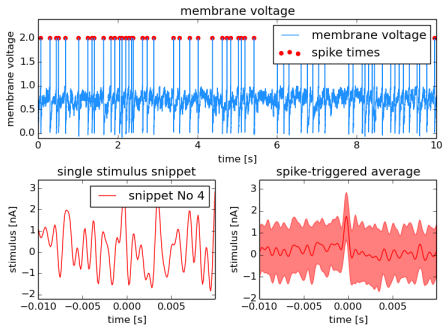
Source code for this example: `taggedFeature.py`.

List of Tutorials

Indexed Feature

In the example, the signal is the membrane potential of a (model) neuron which was stimulated with some stimulus. The events are again the action potentials (or spikes) fired by that neuron. A typical analysis performed on such data is the Spike triggered average which represent the average stimulus that led to a spike. For each spike, a snippet of the respective stimulus is cut out and later averaged. In this example we store these stimulus snippets and link them to the events by adding a **Feature** to the **MultiTag**. There are three different flags that define how this link has to be interpreted. In this case there is one snippet for each spike. The index of each position has to be used as an index in the first dimension of the Feature data. The **LinkType** has to be set to *indexed*.

```
    snippets.unit = "nA"
    set_dim = snippets.append_set_dimension()
    # add a descriptor for the time axis
    dim = snippets.append_sampled_dimension(stepsize)
    dim.unit = "s"
    dim.label = "time"
    dim.offset = -sta_offset * stepsize

    # set snippets as an indexed feature of the multi_tag
    multi_tag.create_feature(snippets, nix.LinkType.Indexed)

    # let's plot the data from the stored information
    plot_data(multi_tag)
```



Source code for this example: `spikeFeatures.py`.

List of Tutorials

Retrieving feature data

The above sections have shown how to attach features to tagged regions. To get the feature data back there are two ways. (i) You can access the data via the selected feature as it is shown in the example above (:download: *spikeFeatures.py <examples/spikeFeatures.py>*) (line 61).

```
    snippets = tag.features[0].data[:]
```

With this line of code you get all the data stored in the Feature as one numpy array. If you want to get the feature data that is related to a singe point (or region) one can call (line 62):

```
    single_snippet = tag.retrieve_feature_data(3, 0)[:]
```

with the first argument being the index of the position and the second one that of the feature. In case of **Tag** entities, there is only one argument that is the index of the feature you want the data from.

List of Tutorials

1.3.6 Storing the origin of data

Let's assume we want to note the origin of the data. For example they have been obtained from a certain experiment or an experimental subject. For this purpose **Source** entities are used. Sources can be nested to reflect dependencies between different sources. For example One may record data from different neurons in the same brain region of the same animal.

```
# create some source entities
subject.block.create_source('mouese A', 'nix.experimental_subject')
brain_region = subject.create_source('hippocampus', 'nix.experimental_subject')
cell_1 = brain_region.create_source('CA1 1', 'nix.experimental_subject')
cell_2 = brain_region.create_source('CA1 2', 'nix.experimental_subject')
# add them to the data.
da1 = block.create_data_array("cell1 response", "nix.regular_sampled", data=response_
↪1)
da1.sources.append(cell_1)
da2 = block.create_data_array("cell2 response", "nix.regular_sampled", data=response_
↪2)
da2.sources.append(cell_2)
```

The **Sources** can be used to indicate links between data that cannot be reflected by the data itself.

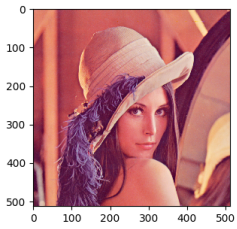
List of Tutorials

1.3.7 Adding arbitrary metadata

Almost all entities allow to attach arbitray metadata. The basic concept of the metadata model is that **Properties** are oragnized in **Sections** which in turn can be nested to represent hierarchical structures. The **Sections** basically act like python dictionaries. How to create sections and properties is demonstrated by attaching information about the 'Lenna' image used above.

```
    if len(cell_text) > 0:
        nrows, ncols = len(cell_text)+1, len(columns)
        ax.axis('off')
        the_table = ax.table(cellText=cell_text,
                             colLabels=columns,
                             loc='center')
        for cell in the_table.get_children():
            cell.set_height(.075)
            cell.set_fontsize(16)
```

Source code for this example: `imageWithMetadata.py`.



Name	Value
Original name	lenna
Journal	Playboy Magazine
Year	1972
Month	November
Author	Dwight Hooker
Source	http://en.wikipedia.org/wiki/T...
Comment	512x512 electronic/mechanical ...
Model	Lena Soederberg

List of Tutorials

CHAPTER 2

API Documentation

TODO write something

2.1 API Documentation for Data

TODO write something about the data model

2.1.1 File

A File represents a specific data source of a NIX back-end for example an NIX HDF5 file. All entities of the nix data model (except the Value entity) must exist in the context of an open File object. Therefore NIX entities can't be initialized via their constructors but only through the factory methods of their respective parent entity.

Working with files

```
file = File.open("test.h5", FileMode.ReadWrite)
# do some work
file.close()
```

File open modes

class nixio.**FileMode**

Overwrite = 'w'

ReadOnly = 'r'

ReadWrite = 'a'

File API

class nixio.**File**(*h5file*, *compression*)

blocks
A property containing all blocks of a file. Blocks can be obtained by their id or their index. Blocks can be deleted from the list, when a block is deleted all its content (data arrays, tags and sources) will be also deleted from the file. Adding new Block is done via the create_block method of File. This is a read-only attribute.

Type ProxyList of Block entities.

close()
Closes an open file.

create_block(*name*, *type_*)
Create a new block inside the file.

Parameters

- **name** (*str*) – The name of the block to create.
- **type** (*str*) – The type of the block.

Returns The newly created block.

Return type *Block*

create_section(*name*, *type_*)
Create a new metadata section inside the file.

Parameters

- **name** (*str*) – The name of the section to create.
- **type** (*str*) – The type of the section.

Returns The newly created section.

Return type *Section*

created_at
The creation time of the file. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

find_sections(*filtr=<function FileMixin.<lambda>>*, *limit=None*)
Get all sections and their child sections recursively.

This method traverses the trees of all sections. The traversal is accomplished via breadth first and can be limited in depth. On each node or section a filter is applied. If the filter returns true the respective section will be added to the result list. By default a filter is used that accepts all sections.

Parameters

- **filtr** (*function*) – A filter function
- **limit** (*int*) – The maximum depth of traversal

Returns A list containing the matching sections.

Return type list of Section

flush()

force_created_at(*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set

force_updated_at(*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set (default: now)

format
The format of the file. This read only property should always have the value 'nix'.

Type str

is_open()
Checks whether a file is open or closed.

Returns True if the file is open, False otherwise.

Return type bool

classmethod open(*path*, *mode='a'*, *backend=None*, *compression='Auto'*)
Open a NIX file, or create it if it does not exist.

Parameters

- **path** – Path to file
- **mode** – FileMode ReadOnly, ReadWrite, or Overwrite. (default: ReadWrite)
- **backend** – Either "hdf5" or "h5py". Defaults to "hdf5" if available, or "h5py" otherwise

Returns nixio.File object

sections
A property containing all root sections of a file. Specific root sections can be obtained by their id or their index. Sections can be deleted from this list. Notice: when a section is deleted all its child section and properties will be removed too. Adding a new Section is done via the crate_section method of File. This is a read-only property.

Type ProxyList of Section entities.

updated_at
The time of the last update of the file. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

validate()
Checks if the File is a valid NIX file. This method is only available when using the "hdf5" backend.

Returns Result object

version
The file format version.

Type tuple

2.1.2 Block

The Block entity is a top-level, summarizing element that allows to group the other data entities belonging for example to the same recording session. All data entities such as Source, DataArray, Tag and MultiTag have to be associated with one Block.

Create a new Block

A block can only be created on an existing file object. Do not use the blocks constructors for this purpose.

```
block = file.create_block("session one", "recordingsession");
```

Working with blocks

After a block was created it can be used to create further entities. See the documentation of Source, DataArray, Tag and MultiTag for more information. The next example shows how some properties of a block can be accessed.

```
block = file.blocks[some_id]

# add metadata to a block
section = file.sections[sec_id]
block.metadata = section

# get associated metadata from a block
block.metadata

# remove associated metadata from a block
block.metadata = None
```

Deleting a block

When a block is deleted from a file all contained data e.g. sources, data arrays and tags will be removed too.

```
del file.blocks[some_id]
```

Block API

class nixio.pycore.**Block**(*nixparent*, *h5group*, *compression='Auto'*)

create_data_array(*name*, *array_type*, *dtype=None*, *shape=None*, *data=None*, *compression='Auto'*)

Create a new data array for this block. Either `shape` or `data` must be given. If both are given their shape must agree. If `dtype` is not specified it will default to 64-bit floating points.

Parameters

- **name** (*str*) – The name of the data array to create.
- **array_type** (*str*) – The type of the data array.
- **dtype** (`numpy.dtype`) – Which data-type to use for storage
- **shape** (*tuple of int or long*) – Layout (dimensionality and extent)
- **data** (*array-like data*) – Data to write after storage has been created
- **compression** (`Compression`) – En-/disable dataset compression.

Returns The newly created data array.

Return type `DataArray`

create_group(*name*, *type_*)
Create a new group on this block.

Parameters

- **name** (*str*) – The name of the group to create.
- **type** (*str*) – The type of the group.

Returns The newly created group.

Return type *Group*

create_multi_tag(*name*, *type_*, *positions*)
Create a new multi tag for this block.

Parameters

- **name** (*str*) – The name of the tag to create.
- **type** (*str*) – The type of tag.
- **positions** (DataArray) – A data array defining all positions of the tag.

Returns The newly created tag.

Return type *MultiTag*

create_source(*name*, *type_*)
Create a new source on this block.

Parameters

- **name** (*str*) – The name of the source to create.
- **type** (*str*) – The type of the source.

Returns The newly created source.

Return type *Source*

create_tag(*name*, *type_*, *position*)
Create a new tag for this block.

Parameters

- **name** (*str*) – The name of the tag to create.
- **type** (*str*) – The type of tag.
- **position** – Coordinates of the start position in units of the respective data dimension.

Returns The newly created tag.

Return type *Tag*

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

data_arrays
A property containing all data arrays of a block. DataArray entities can be obtained via their index or by their id. Data arrays can be deleted from the list. Adding a data array is done using the Blocks create_data_array method. This is a read only attribute.

Type ProxyList of DataArray entities.

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

find_sources (*filtr=<function BlockMixin.<lambda>>*, *limit=None*)
Get all sources in this block recursively.

This method traverses the tree of all sources in the block. The traversal is accomplished via breadth first and can be limited in depth. On each node or source a filter is applied. If the filter returns true the respective source will be added to the result list. By default a filter is used that accepts all sources.

Parameters

- **filtr** (*function*) – A filter function
- **limit** (*int*) – The maximum depth of traversal

Returns A list containing the matching sources.

Return type list of Source

force_created_at (*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at (*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

groups
A property containing all groups of a block. Group entities can be obtained via their index or by their id. Groups can be deleted from the list. Adding a Group is done using the Blocks create_group method. This is a read only attribute.

Type ProxyList of Group entities.

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

metadata
Associated metadata of the entity. Sections attached to the entity via this attribute can provide additional annotations. This is an optional read-write property, and can be None if no metadata is available.

Type *Section*

multi_tags
A property containing all multi tags of a block. MultiTag entities can be obtained via their index or by their id. Tags can be deleted from the list. Adding tags is done using the Blocks create_multi_tag method. This is a read only attribute.

Type ProxyList of MultiTag entities.

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

sources
A property containing all sources of a block. Sources can be obtained via their index or by their id. Sources can be deleted from the list. Adding sources is done using the Blocks create_source method. This is a read only attribute.

Type ProxyList of Source entities.

tags
A property containing all tags of a block. Tag entities can be obtained via their index or by their id. Tags can be deleted from the list. Adding tags is done using the Blocks create_tag method. This is a read only attribute.

Type ProxyList of Tag entities.

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

2.1.3 DataArray

The DataArray is the core entity of the NIX data model, its purpose is to store arbitrary n-dimensional data. In addition to the common fields id, name, type, and definition the DataArray stores sufficient information to understand the physical nature of the stored data.

A guiding principle of the data model is provides enough information to create a plot of the stored data. In order to do so, the DataArray defines a property dataType which provides the physical type of the stored data (for example 16 bit integer or double precision IEEE floatingpoint number). The property unit specifies the SI unit of the values stored in the DataArray whereas the label defines what is given in this units. Together, both specify what corresponds to the the y-axis of a plot.

In some cases it is much more efficient or convenient to store data not as floating point numbers but rather as (16 bit) integer values as, for example read from a data acquisition board. In order to convert such data to the correct values, we follow the approach taken by the comedi data-acquisition library (http://www.comedi.org) and provide polynomCoefficients and an expansionOrigin.

Create and delete a DataArray

A DataArray can only be created at an existing block. Do not use the DataArrays constructors for this purpose.

```
1 data_array = block.crate_data_array("matrix", "data");
2 del block.data_arrays[data_array]
```

DataArray API

class nixio.pycore.**DataArray**(*nixparent*, *h5group*)

append(*data*, *axis=0*)
Append `data` to the DataSet along the `axis` specified.

Parameters **data** – The data to append. Shape must agree except for the

specified axis :param axis: Along which axis to append the data to

append_alias_range_dimension()
Append a new RangeDimension that uses the data stored in this DataArray as ticks. This works only(!) if the DataArray is 1-D and the stored data is numeric. A ValueError will be raised otherwise.

Returns The created dimension descriptor.

Return type RangeDimension

append_range_dimension(*ticks*)
Append a new RangeDimension to the list of existing dimension descriptors.

Parameters **ticks** (*list of float*) – The ticks of the RangeDimension to create.

Returns The newly created RangeDimension.

Return type RangeDimension

append_sampled_dimension(*sampling_interval*)
Append a new SampledDimension to the list of existing dimension descriptors.

Parameters **sampling_interval** (*float*) – The sampling interval of the SetDimension to create.

Returns The newly created SampledDimension.

Return type SampledDimension

append_set_dimension()
Append a new SetDimension to the list of existing dimension descriptors.

Returns The newly created SetDimension.

Return type SetDimension

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

data
DEPRECATED DO NOT USE ANYMORE! Returns self

Type DataArray

data_extent
The size of the data.

Type set of int

data_type
The data type of the data stored in the DataArray. This is a read only property.

Type DataType

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

delete_dimensions()
Delete all the dimension descriptors for this DataArray.

dimensions
A property containing all dimensions of a DataArray. Dimensions can be obtained via their index. Adding dimensions is done using the respective append methods for dimension descriptors. This is a read only attribute.

Type ProxyList of dimension descriptors.

dtype
The data type of the data stored in the DataArray. This is a read only property.

Returns DataType

expansion_origin
The expansion origin of the calibration polynomial. This is a read-write property and can be set to None. The default value is 0.

Type float

force_created_at (*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at (*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

get_slice (*positions*, *extents=None*, *mode=<DataSliceMode.Index: 1>*)

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

label
The label of the DataArray. The label corresponds to the label of the x-axis of a plot. This is a read-write property and can be set to None.

Type str

len ()
Length of the first dimension. Equivalent to *DataSet.shape[0]*.

Type int or long

metadata
Associated metadata of the entity. Sections attached to the entity via this attribute can provide additional annotations. This is an optional read-write property, and can be None if no metadata is available.

Type *Section*

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

polynom_coefficients
The polynomial coefficients for the calibration. By default this is set to a {0.0, 1.0} for a linear calibration with zero offset. This is a read-write property and can be set to None

Type list of float

read_direct(*data*)
Directly read all data stored in the DataSet into data. The supplied data must be a numpy.ndarray that matches the DataSet's shape, must have C-style contiguous memory layout and must be writeable (see numpy.ndarray.flags and ndarray for more information).

Parameters **data** (numpy.ndarray) – The array where data is being read into

shape

Type tuple of data array dimensions.

size
Number of elements in the DataSet, i.e. the product of the elements in shape.

Type int

sources
Getter for sources.

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

unit
The unit of the values stored in the DataArray. This is a read-write property and can be set to None.

Type str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

write_direct(*data*)
Directly write all of data to the DataSet. The supplied data must be a numpy.ndarray that matches the DataSet's shape and must have C-style contiguous memory layout (see numpy.ndarray.flags and ndarray for more information).

Parameters **data** (numpy.ndarray) – The array which contents is being written

2.1.4 DataSet

The DataSet object is used for data input/output to the underlying storage.

class nixio.pycore.data_array.**DataSet**

append(*data*, *axis=0*)
Append data to the DataSet along the axis specified.

Parameters **data** – The data to append. Shape must agree except for the

specified axis :param axis: Along which axis to append the data to

data_extent
The size of the data.

Type set of int

data_type
The data type of the data stored in the DataArray. This is a read only property.

Type DataType

dtype

Type numpy.dtype object holding type infromation about the data stored in the DataSet.

len()
Length of the first dimension. Equivalent to *DataSet.shape[0]*.

Type int or long

read_direct(*data*)
Directly read all data stored in the DataSet into data. The supplied data must be a numpy.ndarray that matches the DataSet's shape, must have C-style contiguous memory layout and must be writeable (see numpy.ndarray.flags and ndarray for more information).

Parameters **data** (numpy.ndarray) – The array where data is being read into

shape

Type tuple of data array dimensions.

size
Number of elements in the DataSet, i.e. the product of the elements in shape.

Type int

write_direct(*data*)
Directly write all of data to the DataSet. The supplied data must be a numpy.ndarray that matches the DataSet's shape and must have C-style contiguous memory layout (see numpy.ndarray.flags and ndarray for more information).

Parameters **data** (numpy.ndarray) – The array which contents is being written

2.1.5 Tags

Besides the DataArray the tag entities can be considered as the other core entities of the data model. They are meant to attach annotations directly to the data and to establish meaningful links between different kinds of stored data. Most importantly tags allow the definition of points or regions of interest in data that is stored in other DataArray entities. The data array entities the tag applies to are defined by its property references.

Further the referenced data is defined by an origin vector called position and an optional extent vector that defines its size. Therefore position and extent of a tag, together with the references field defines a group of points or regions of interest collected from a subset of all available DataArray entities.

Further tags have a field called features which makes it possible to associate other data with the tag. Semantically a feature of a tag is some additional data that contains additional information about the points of hyperslabs defined by the tag. This could be for example data that represents a stimulus (e.g. an image or a signal) that was applied in a certain interval during the recording.

Tag API

class nixio.pycore.**Tag**(*nixparent*, *h5group*)

create_feature(*data*, *link_type*)
Create a new feature.

Parameters

- **data** (DataArray) – The data array of this feature.

- **link_type** (*LinkType*) – The link type of this feature.

Returns The created feature object.

Return type Feature

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

extent
The extent defined by the tag. This is an optional read-write property and may be set to None.

Type list of float

features
A property containing all features of the tag. Features can be obtained via their index or their id. Features can be deleted from the list. Adding new features to the tag is done using the create_feature method. This is a read only attribute.

Type ProxyList of Feature.

force_created_at(*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at(*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

metadata
Associated metadata of the entity. Sections attached to the entity via this attribute can provide additional annotations. This is an optional read-write property, and can be None if no metadata is available.

Type *Section*

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

position
The position defined by the tag. This is a read-write property.

Type list of float

references
A property containing all data arrays referenced by the tag. Referenced data arrays can be obtained by

index or their id. References can be removed from the list, removing a referenced DataArray will not remove it from the file. New references can be added using the append method of the list. This is a read only attribute.

Type RefProxyList of DataArray

retrieve_data(*refidx*)

retrieve_feature_data(*featidx*)

sources
Getter for sources.

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

units
Property containing the units of the tag. The tag must provide a unit for each dimension of the position or extent vector. This is a read-write property.

Type list of str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

MultiTag API

class nixio.pycore.**MultiTag**(*nixparent*, *h5group*)

create_feature(*data*, *link_type*)
Create a new feature.

Parameters

- **data** (DataArray) – The data array of this feature.
- **link_type** (*LinkType*) – The link type of this feature.

Returns The created feature object.

Return type Feature

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

extents
The extents defined by the tag. This is an optional read-write property and may be set to None.

Type *DataArray* or None

features
A property containing all features of the tag. Features can be obtained via their index or their id. Features can be deleted from the list. Adding new features to the tag is done using the create_feature method. This is a read only attribute.

Type ProxyList of Feature.

force_created_at (*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at (*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

metadata
Associated metadata of the entity. Sections attached to the entity via this attribute can provide additional annotations. This is an optional read-write property, and can be None if no metadata is available.

Type *Section*

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

positions
The positions defined by the tag. This is a read-write property.

Type *DataArray*

references
A property containing all data arrays referenced by the tag. Referenced data arrays can be obtained by index or their id. References can be removed from the list, removing a referenced DataArray will not remove it from the file. New references can be added using the append method of the list. This is a read only attribute.

Type RefProxyList of DataArray

retrieve_data (*posidx*, *refidx*)

retrieve_feature_data (*posidx*, *featidx*)

sources
Getter for sources.

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

units
Property containing the units of the tag. The tag must provide a unit for each dimension of the position or extent vector. This is a read-write property.

Type list of str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

2.1.6 Source

class nixio.pycore.**Source**(*nixparent*, *h5group*)

create_source(*name*, *type_*)
Create a new source as a child of the current Source.

Parameters

- **name** (*str*) – The name of the source to create.
- **type** (*str*) – The type of the source.

Returns The newly created source.

Return type *Source*

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

find_sources(*filtr=<function SourceMixin.<lambda>>*, *limit=None*)
Get all child sources of this source recursively.

This method traverses the tree of all sources. The traversal is accomplished via breadth first and can be limited in depth. On each node or source a filter is applied. If the filter returns true the respective source will be added to the result list. By default a filter is used that accepts all sources.

Parameters

- **filtr** (*function*) – A filter function
- **limit** (*int*) – The maximum depth of traversal

Returns A list containing the matching sources.

Return type list of Source

force_created_at(*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at (*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

metadata
Associated metadata of the entity. Sections attached to the entity via this attribute can provide additional annotations. This is an optional read-write property, and can be None if no metadata is available.

Type *Section*

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

referring_data_arrays

referring_multi_tags

referring_objects

referring_tags

sources
A property containing all sources of a block. Sources can be obtained via their index or by their id. Sources can be deleted from the list. Adding sources is done using the Blocks create_source method. This is a read only attribute.

Type ProxyList of Source entities.

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

2.1.7 Group

Groups establish a simple way of grouping entities that in some way belong together. The Group exists inside a Block and can contain (link) DataArrays, Tags, and MultiTags. As any other nix-entity, the Groups is named, has a type, and a definition property. Additionally, it contains data_arrays, tags, and multi_tags lists. As indicated before, the group does only link the entities. Thus, deleting elements from the lists does not remove them from file, it merely removes the link from the group.

```
data_array = block.crate_data_array("matrix", "data");
tag = block.create_tag("a tag", "event", [0.0, 1.0])
group = block.create_group("things that belong together", "group")
```

(continues on next page)

(continued from previous page)

```
group.tags.append(tag)
group.data_arrays.append(data_array)

del group.data_arrays[data_array]
del group.tags[tag]
```

Group API

class nixio.pycore.**Group**(*nixparent*, *h5group*)

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

data_arrays
A property containing all data arrays referenced by the group. Referenced data arrays can be obtained by index or their id. References can be removed from the list, removing a referenced DataArray will not remove it from the file. New references can be added using the append method of the list. This is a read only attribute.

Type DataArrayProxyList of DataArray

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

force_created_at(*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at(*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

metadata
Associated metadata of the entity. Sections attached to the entity via this attribute can provide additional annotations. This is an optional read-write property, and can be None if no metadata is available.

Type *Section*

multi_tags
A property containing all MultiTags referenced by the group. MultiTags can be obtained by index or their id. Tags can be removed from the list, removing a referenced MultiTag will not remove it from the file. New MultiTags can be added using the append method of the list. This is a read only attribute.

Type MultiTagProxyList of MultiTags

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

sources
Getter for sources.

tags
A property containing all tags referenced by the group. Tags can be obtained by index or their id. Tags can be removed from the list, removing a referenced Tag will not remove it from the file. New Tags can be added using the append method of the list. This is a read only attribute.

Type TagProxyList of Tags

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

2.2 API Documentation for Metadata

The model for storing metadata is largely equivalent to the odML (open metadata Markup Laguage) model. In brief: the model consists of so called Properties that contain Values much like a key-value pair (plus some additional fields). These Properties can be grouped into Sections which themselves can be nested to built a tree-structure. Sections are defined by a name and a type (e.g. a stimulus-type section will contain information that is related to a stimulus). The basic feature of the odML approach is that it defines the model but not the items that are described or the terms that are used in this. On the other hand where standardization is required each Section can be based on an odML-terminology that standardizes without restricting to the terms defined within the terminology.

2.2.1 Section

Metadata stored in a NIX file can be accessed directly from an open file.

Create and delete sub sections

```
sub = section.create_section("a name", "type")
del section.sections[sub]
```

Add and remove properties

Properties can be created using the create_property method. Existing properties can be accessed and deleted directly from the respective section.

```
section.create_property("one", [Value(1)])
section.create_property("two", [Value(2)])

# iterate over properties
for p in section:
   print(p)

# access by name
one = section["one"]

# convert properties into a dict
dct = dict(section.items())

# delete properties
del section["one"]
del section["two"]
```

Section API

class nixio.pycore.**Section**(*nixparent*, *h5group*)

create_property(*name*, *values*)
Add a new property to the section.

Parameters

- **name** (*str*) – The name of the property to create.
- **values** (*list of Value*) – The values of the property.

Returns The newly created property.

Return type *Property*

create_section(*name*, *type_*)
Creates a new subsection that is a child of this section entity.

Parameters

- **name** (*str*) – The name of the section to create.
- **type** (*str*) – The type of the section.

Returns The newly created section.

Return type *Section*

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

definition
The definition of the entity. The definition can contain a textual description of the entity. This is an optional read-write property, and can be None if no definition is available.

Type str

file
Root file object.

Type *File*

find_related(*filtr=<function SectionMixin.<lambda>>*)
Get all related sections of this section.

The result can be filtered. On each related section a filter is applied. If the filter returns true the respective section will be added to the result list. By default a filter is used that accepts all sections.

Parameters **filtr** (*function*) – A filter function

Returns A list containing the matching related sections.

Return type list of Section

find_sections(*filtr=<function SectionMixin.<lambda>>*, *limit=None*)
Get all child sections recursively. This method traverses the trees of all sections. The traversal is accomplished via breadth first and can be limited in depth. On each node or section a filter is applied. If the filter returns true the respective section will be added to the result list. By default a filter is used that accepts all sections.

Parameters

- **filtr** (*function*) – A filter function
- **limit** (*int*) – The maximum depth of traversal

Returns A list containing the matching sections.

Return type list of Section

force_created_at(*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at(*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

get_property_by_name(*name*)
Get a property by its name.

Parameters **name** (*str*) – The name to check.

Returns The property with the given name.

Return type *Property*

has_property_by_name(*name*)
Checks whether a section has a property with a certain name.

Parameters **name** (*str*) – The name to check.

Returns True if the section has a property with the given name, False otherwise.

Return type bool

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

inherited_properties()

items()

link
Link to another section. If a section is linked to another section, the linking section inherits all properties from the target section. This is an optional read-write property and may be set to None.

Type *Section*

mapping

name
The name of an entity. The name serves as a human readable identifier. This is a read-only property; entities cannot be renamed.

Type str

parent
The parent section. This is a read-only property. For root sections this property is always None.

Accessing this property can be slow when the metadata tree is large.

Type *Section*

pprint(*max_depth=1*, *indent=2*, *max_length=80*, *current_depth=0*)

props
A property containing all Property entities associated with the section. Properties can be accessed by index of via their id. Properties can be deleted from the list. Adding new properties is done using the create_property method. This is a read-only attribute.

Type ProxyList of Property

referring_blocks

referring_data_arrays

referring_groups

referring_multi_tags

referring_objects

referring_sources

referring_tags

repository
URL to the terminology repository the section is associated with. This is an optional read-write property and may be set to None.

Type str

sections
A property providing all child sections of a section. Child sections can be accessed by index or by their id. Sections can also be deleted: if a section is deleted, all its properties and child sections are removed from the file too. Adding new sections is achieved using the create_section method. This is a read-only attribute.

Type ProxyList of Section

type
The type of the entity. The type is used in order to add semantic meaning to the entity. This is a read-write property, but it can't be set to None.

Type str

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

2.2.2 Property

class nixio.pycore.**Property**(*nixparent*, *h5dataset*)

created_at
The creation time of the entity. This is a read-only property. Use *force_created_at* in order to change the creation time.

Return type int

data_type

definition

delete_values()

force_created_at(*t=None*)
Sets the creation time *created_at* to the given time (default: current time).

Parameters **t** (*int*) – The time to set.

force_updated_at(*t=None*)
Sets the update time *updated_at* to the given time. (default: current time)

Parameters **t** (*int*) – The time to set.

id
A property providing the ID of the Entity. The id is generated automatically, therefore the property is read-only.

Return type str

mapping

name

pprint(*indent=2*, *max_length=80*, *current_depth=-1*)

unit

updated_at
The time of the last update of the entity. This is a read-only property. Use *force_updated_at* in order to change the update time.

Return type int

values

2.2.3 Value

class nixio.**Value**(*value*)

to_string(*unit=''*)

Index

P

R

S

T

U

V

W