legend-pydataobj

unknown

May 08, 2024

CONTENTS

legend-pydataobj is a Python implementation of the LEGEND Data Format Specification.

CHAPTER

ONE

GETTING STARTED

legend-pydataobj is published on the Python Package Index. Install on local systems with pip:

```
$ pip install legend-pydataobj
```

```
$ pip install legend-pydataobj@git+https://github.com/legend-exp/legend-pydataobj@main
```

Get a LEGEND container with *legend-pydataobj* pre-installed on Docker hub or follow instructions on the LEGEND wiki.

If you plan to develop *legend-pydataobj*, refer to the *developer's guide*.

Attention: If installing in a user directory (typically when invoking pip as a normal user), make sure `~/.local/bin` is appended to `PATH`. The `lh5ls` executable is installed there.

CHAPTER

TWO

NEXT STEPS

2.1 User Manual

2.1.1 LH5 Command Line Tools

legend-pydataobj provides some command line utilities to deal with LH5 files.

Inspecting LH5 files with `lh5ls`

The `lh5ls` command uses *lh5.tools.show()* to print a pretty representation of a LH5 file's contents:

```
$ lh5ls -a legend-testdata/data/lh5/LDQTA_r117_20200110T105115Z_cal_geds_raw.lh5
/
└── geds · HDF5 group
    └── raw · table{packet_id,ievt,timestamp,numtraces,tracelist,baseline,energy,channel,
↪wf_max,wf_std,waveform}
        ├── baseline · array<1>{real}
        ├── channel · array<1>{real}
        ├── energy · array<1>{real}
        ├── ievt · array<1>{real}
        ├── numtraces · array<1>{real}
        ├── packet_id · array<1>{real}
        ├── timestamp · array<1>{real} ── {'units': 's'}
        ├── tracelist · array<1>{array<1>{real}}
        │   ├── cumulative_length · array<1>{real}
        │   └── flattened_data · array<1>{real}
        ├── waveform · table{t0,dt,values}
        │   ├── dt · array<1>{real} ── {'units': 'ns'}
        │   ├── t0 · array<1>{real} ── {'units': 'ns'}
        │   └── values · array_of_equalsized_arrays<1,1>{real}
        ├── wf_max · array<1>{real}
        └── wf_std · array<1>{real}
```

For more information, have a look at the command's help section:

```
lh5ls --help
```

Concatenating LGDOs with `lh5concat`

The `lh5concat` command can be used to concatenate LGDO *`Array`*, `VectorOfVectors` and *`Table`* into an output LH5 file.

Concatenate all eligible objects in `file{1,2}.lh5` into `concat.lh5`:

```
lh5concat -o concat.lh5 file1.lh5 file2.lh5
```

Include only the /data/table1 Table:

```
lh5concat -o concat.lh5 -i '/data/table1/*' file1.lh5 file2.lh5
```

Exclude the /data/table1/col1 Table column:

```
lh5concat -o concat.lh5 -e /data/table1/col1 file1.lh5 file2.lh5
```

For more information, have a look at the command's help section:

```
lh5concat --help
```

2.2 Tutorials (Jupyter notebooks)

Tutorials are currently available as Jupyter notebooks on GitHub.

2.2.1 Handling LH5 data

LEGEND stores its data in HDF5 format, a high-performance data format becoming popular in experimental physics. LEGEND Data Objects (LGDO) are represented as HDF5 objects according to a custom specification, documented here.

Reading data from disk

Let's start by downloading a small test LH5 file with the pylegendtestdata package (it takes a while depending on your internet connection):

[1]:
```
from legendtestdata import LegendTestData

ldata = LegendTestData()
lh5_file = ldata.get_path("lh5/LDQTA_r117_20200110T105115Z_cal_geds_raw.lh5")
```

We can use `lgdo.lh5.ls()` [docs] to inspect the file contents:

[2]:
```
from lgdo import lh5

lh5.ls(lh5_file)
```

[2]:
```
['geds']
```

This particular file contains an HDF5 group (they behave like directories). The second argument of `ls()` can be used to inspect a group (without the trailing `/`, only the group name is returned, if existing):

[3]:
```
lh5.ls(lh5_file, "geds/")  # returns ['geds/raw'], which is a group again
lh5.ls(lh5_file, "geds/raw/")
```

[3]:
```
['geds/raw/baseline',
 'geds/raw/channel',
 'geds/raw/energy',
 'geds/raw/ievt',
 'geds/raw/numtraces',
 'geds/raw/packet_id',
 'geds/raw/timestamp',
 'geds/raw/tracelist',
 'geds/raw/waveform',
 'geds/raw/wf_max',
 'geds/raw/wf_std']
```

Note: Alternatively to `ls()`, `show()` [docs] prints a nice representation of the LH5 file contents (with LGDO types) on screen:

---

[4]:
```
lh5.show(lh5_file)
```

```
/
└── geds · HDF5 group
    └── raw · table{packet_id,ievt,timestamp,numtraces,tracelist,baseline,energy,channel,
↪wf_max,wf_std,waveform}
        ├── baseline · array<1>{real}
        ├── channel · array<1>{real}
        ├── energy · array<1>{real}
        ├── ievt · array<1>{real}
        ├── numtraces · array<1>{real}
        ├── packet_id · array<1>{real}
        ├── timestamp · array<1>{real}
        ├── tracelist · array<1>{array<1>{real}}
        │   ├── cumulative_length · array<1>{real}
        │   └── flattened_data · array<1>{real}
        ├── waveform · table{t0,dt,values}
        │   ├── dt · array<1>{real}
        │   ├── t0 · array<1>{real}
        │   └── values · array_of_equalsized_arrays<1,1>{real}
        ├── wf_max · array<1>{real}
        └── wf_std · array<1>{real}
```

The group contains several LGDOs. Let's read them in memory. `read()` [docs] reads an LGDO from disk and returns the object in memory. Let's try to read `geds/raw`:

[5]:
```
lh5.read("geds/raw", lh5_file)
```

[5]:
```
Table(dict={'packet_id': Array([1 2 ... 99 100], attrs={'datatype': 'array<1>{real}'}),
↪'ievt': Array([0 0 ... 3 32], attrs={'datatype': 'array<1>{real}'}), 'timestamp':␣
↪Array([0.79465985 0.7968994 ... 0.974689 0.9786208 ], attrs={'datatype': 'array<1>
↪{real}', 'units': 's'}), 'numtraces': Array([1 1 ... 1 1], attrs={'datatype': 'array<1>
↪{real}'}), 'tracelist': VectorOfVectors(flattened_data=Array([53 60 ... 30 53], attrs={
↪'datatype': 'array<1>{real}'}), cumulative_length=Array([1 2 ... 99 100], attrs={
↪'datatype': 'array<1>{real}'}), attrs={'datatype': 'array<1>{array<1>{real}}'}),
```

(continues on next page)

(continued from previous page)

```
↪'baseline': Array([13722 13044 ... 9931 17013], attrs={'datatype': 'array<1>{real}'}),
↪'energy': Array([3304 8642 ... 6014 3410], attrs={'datatype': 'array<1>{real}'}),
↪'channel': Array([53 60 ... 30 53], attrs={'datatype': 'array<1>{real}'}), 'wf_max':␣
↪Array([16352 20549 ... 14317 19567], attrs={'datatype': 'array<1>{real}'}), 'wf_std':␣
↪Array([1028.1815 3084.8018 ... 1876.9403 1065.3331], attrs={'datatype': 'array<1>{real}
↪'}), 'waveform': WaveformTable(dict={'t0': Array([0. 0. ... 0. 0.], attrs={'datatype':
↪'array<1>{real}', 'units': 'ns'}), 'dt': Array([16. 16. ... 16. 16.], attrs={'datatype
↪': 'array<1>{real}', 'units': 'ns'}), 'values': ArrayOfEqualSizedArrays([[13712 13712␣
↪... 15380 15400] [13072 13072 ... 18819 18806] ... [9962 9962 ... 13326 13269] [16918␣
↪16918 ... 18877 18933]], attrs={'datatype': 'array_of_equalsized_arrays<1,1>{real}'})},
↪ attrs={'datatype': 'table{t0,dt,values}'})}, attrs={'datatype': 'table{packet_id,ievt,
↪timestamp,numtraces,tracelist,baseline,energy,channel,wf_max,wf_std,waveform}'})
```

As shown by the type signature, it is interpreted as a `Table` with 100 rows. Its contents (or "columns") can be therefore viewed as LGDO objects of the same length. For example `timestamp`:

[6]:
```
lh5.read("geds/raw/timestamp", lh5_file)
```

[6]:
```
Array([0.79465985 0.7968994  0.79960424 ... 0.97331905 0.974689
       0.9786208 ], attrs={'datatype': 'array<1>{real}', 'units': 's'})
```

is an LGDO `Array` with 100 elements.

`read()` also allows to perform more advanced data reading. For example, let's read only rows from 15 to 25:

[7]:
```
obj = lh5.read("geds/raw/timestamp", lh5_file, start_row=15, n_rows=10)
print(obj)
```

```
[0.82679445 0.8307392  0.8298773  0.830739   0.8339691  0.83487684
 0.83510256 0.83612865 0.83797085 0.8406608 ] with attrs={'units': 's'}
```

Or, let's read only columns `timestamp` and `energy` from the `geds/raw` table and rows `[1, 3, 7, 9, 10, 15]`:

[8]:
```
obj = lh5.read(
    "geds/raw", lh5_file, field_mask=("timestamp", "energy"), idx=[1, 3, 7, 9, 10, 15]
)
print(obj)
```

```
 timestamp  energy
  0.796899    8642
  0.799604   13015
  0.812317   22085
  0.813282   26636
  0.813520    2648
  0.826794    7799

with attrs['timestamp']={'units': 's'}
```

As you might have noticed, `read()` loads all the requested data in memory at once. This can be a problem when dealing with large datasets. `LH5Iterator` [docs] makes it possible to handle data one chunk at a time (sequentially) to avoid running out of memory:

[9]:
```
from lgdo.lh5 import LH5Iterator
```

(continues on next page)

(continued from previous page)

```
for lh5_obj, entry, n_rows in LH5Iterator(lh5_file, "geds/raw/energy", buffer_len=20):
    print(f"entry {entry}, energy = {lh5_obj} ({n_rows} rows)")
```

```
entry 0, energy = [3304 8642 9177 ... 8289 7091 4084] (20 rows)
entry 20, energy = [11546  2873  3193 ...  4114 29557 12309] (20 rows)
entry 40, energy = [ 6455  3302  5314 ... 37333  4262 15131] (20 rows)
entry 60, energy = [ 6117  3358  3132 ...  2949  3691 10402] (20 rows)
entry 80, energy = [ 4088 41153 34295 ... 37877  6014  3410] (20 rows)
```

If working with many files at the same time, theLH5Store [docs] class might come handy:

[10]:
```
from lgdo.lh5 import LH5Store

store = LH5Store(keep_open=True)  # with keep_open=True, files are kept open inside the␣
↪store
store.read("geds/raw", lh5_file)  # returns a tuple: (obj, n_rows_read)
```

[10]:
```
(Table(dict={'packet_id': Array([1 2 ... 99 100], attrs={'datatype': 'array<1>{real}'}),
↪'ievt': Array([0 0 ... 3 32], attrs={'datatype': 'array<1>{real}'}), 'timestamp':␣
↪Array([0.79465985 0.7968994 ... 0.974689 0.9786208 ], attrs={'datatype': 'array<1>
↪{real}', 'units': 's'}), 'numtraces': Array([1 1 ... 1 1], attrs={'datatype': 'array<1>
↪{real}'}), 'tracelist': VectorOfVectors(flattened_data=Array([53 60 ... 30 53], attrs={
↪'datatype': 'array<1>{real}'}), cumulative_length=Array([1 2 ... 99 100], attrs={
↪'datatype': 'array<1>{real}'}), attrs={'datatype': 'array<1>{array<1>{real}}'}),
↪'baseline': Array([13722 13044 ... 9931 17013], attrs={'datatype': 'array<1>{real}'}),
↪'energy': Array([3304 8642 ... 6014 3410], attrs={'datatype': 'array<1>{real}'}),
↪'channel': Array([53 60 ... 30 53], attrs={'datatype': 'array<1>{real}'}), 'wf_max':␣
↪Array([16352 20549 ... 14317 19567], attrs={'datatype': 'array<1>{real}'}), 'wf_std':␣
↪Array([1028.1815 3084.8018 ... 1876.9403 1065.3331], attrs={'datatype': 'array<1>{real}
↪'}), 'waveform': WaveformTable(dict={'t0': Array([0. 0. ... 0. 0.], attrs={'datatype':
↪'array<1>{real}', 'units': 'ns'}), 'dt': Array([16. 16. ... 16. 16.], attrs={'datatype
↪': 'array<1>{real}', 'units': 'ns'}), 'values': ArrayOfEqualSizedArrays([[13712 13712␣
↪... 15380 15400] [13072 13072 ... 18819 18806] ... [9962 9962 ... 13326 13269] [16918␣
↪16918 ... 18877 18933]], attrs={'datatype': 'array_of_equalsized_arrays<1,1>{real}'})},
↪ attrs={'datatype': 'table{t0,dt,values}'})}, attrs={'datatype': 'table{packet_id,ievt,
↪timestamp,numtraces,tracelist,baseline,energy,channel,wf_max,wf_std,waveform}'}),
 100)
```

Have a look at the API reference for more documentation.

Converting LGDO data to alternative formats

Each LGDO is equipped with a class method called `view_as()` [docs], which allows the user to "view" the data (i.e. avoiding copying data as much as possible) in a different, third-party format.

LGDOs generally support viewing as NumPy (`np`), Pandas (`pd`) or Awkward (`ak`) data structures, with some exceptions. We strongly recommend having a look at the `view_as()` API docs of each LGDO type for more details (for `Table.view_as()` [docs], for example).

Note: To obtain a copy of the data in the selected third-party format, the user can call the appropriate third-party copy method on the view (e.g. `pandas.DataFrame.copy()`, if viewing the data as a Pandas dataframe).

Let's play around with our good old table, can we view it as a Pandas dataframe?

[11]:
```
obj = lh5.read("geds/raw", lh5_file)
df = obj.view_as("pd")
df
```

[11]:
```
    packet_id  ievt  timestamp  numtraces  tracelist  baseline  energy  \
0           1     0   0.794660          1       [53]     13722    3304
1           2     0   0.796899          1       [60]     13044    8642
2           3     0   0.799604          1       [40]     13508    9177
3           4     0   0.799604          1       [41]     11891   13015
4           5     1   0.801617          1       [60]     14353    3794
..        ...   ...        ...        ...        ...       ...     ...
95         96    30   0.965948          1       [53]     15597    5361
96         97    44   0.971051          1       [60]     14599    3748
97         98    31   0.973319          1       [53]     15438   37877
98         99     3   0.974689          1       [30]      9931    6014
99        100    32   0.978621          1       [53]     17013    3410

    channel  wf_max        wf_std  waveform_t0  waveform_dt  \
0        53   16352   1028.181519          0.0         16.0
1        60   20549   3084.801758          0.0         16.0
2        40   20119   2849.593750          0.0         16.0
3        41   21215   4023.562256          0.0         16.0
4        60   17150   1183.935913          0.0         16.0
..      ...     ...           ...          ...          ...
95       53   29471    947.814392          0.0         16.0
96       60   17324   1151.771362          0.0         16.0
97       53   42319  11780.090820          0.0         16.0
98       30   14317   1876.940308          0.0         16.0
99       53   19567   1065.333130          0.0         16.0

                            waveform_values
0   [13712 13712 13683 ... 15445 15380 15400]
1   [13072 13072 12992 ... 18842 18819 18806]
2   [13575 13575 13496 ... 18409 18384 18457]
3   [11862 11862 11808 ... 18946 18861 18834]
4   [14432 14432 14409 ... 16380 16403 16410]
..                                        ...
95  [27818 27818 27755 ... 26594 26562 26493]
96  [14471 14471 14515 ... 16583 16582 16609]
97  [15405 15405 15366 ... 36208 36214 36233]
98     [9962 9962 9949 ... 13269 13326 13269]
99  [16918 16918 16962 ... 18715 18877 18933]

[100 rows x 13 columns]
```

Yes! But how are the nested objects being handled?

Nested tables have been flattened by prefixing their column names with the table object name (`obj.waveform.values` becomes `df.waveform_values`) and multi-dimensional columns are represented by Awkward arrays:

[12]:
```
df.waveform_values
```

[12]:
```
0     [13712 13712 13683 ... 15445 15380 15400]
1     [13072 13072 12992 ... 18842 18819 18806]
```

(continues on next page)

(continued from previous page)

```
2     [13575 13575 13496 ... 18409 18384 18457]
3     [11862 11862 11808 ... 18946 18861 18834]
4     [14432 14432 14409 ... 16380 16403 16410]
                        ...
95    [27818 27818 27755 ... 26594 26562 26493]
96    [14471 14471 14515 ... 16583 16582 16609]
97    [15405 15405 15366 ... 36208 36214 36233]
98       [9962 9962 9949 ... 13269 13326 13269]
99    [16918 16918 16962 ... 18715 18877 18933]
Name: waveform_values, Length: 100, dtype: awkward
```

But what if we wanted to have the waveform values as a NumPy array?

[13]:
```
obj.waveform.values.view_as("np")
```

[13]:
```
array([[13712, 13712, 13683, ..., 15445, 15380, 15400],
       [13072, 13072, 12992, ..., 18842, 18819, 18806],
       [13575, 13575, 13496, ..., 18409, 18384, 18457],
       ...,
       [15405, 15405, 15366, ..., 36208, 36214, 36233],
       [ 9962,  9962,  9949, ..., 13269, 13326, 13269],
       [16918, 16918, 16962, ..., 18715, 18877, 18933]], dtype=uint16)
```

Can we just view the full table as a huge Awkward array? Of course:

[14]:
```
obj.view_as("ak")
```

[14]:
```
<Array [{packet_id: 1, ievt: 0, ...}, ...] type='100 * {packet_id: uint32, ...'>
```

Note that viewing a `VectorOfVector` as an Awkward array is a nearly zero-copy operation and opens a new avenue of fast computational possibilities thanks to Awkward:

[15]:
```
import awkward as ak

# tracelist is a VoV on disk
trlist = obj.tracelist.view_as("ak")
ak.mean(trlist)
```

[15]:
```
54.31
```

Last but not least, we support attaching physical units (that might be stored in the `units` attribute of an LGDO) to data views through Pint, if the third-party format allows it:

[16]:
```
df = obj.view_as("pd", with_units=True)
df.timestamp.dtype
```

[16]:
```
pint[s]
```

Note that we also provide the `read_as()` [docs] shortcut to save some typing, for users that would like to read LH5 data on disk straight into some third-party format:

[17]:
```
lh5.read_as("geds/raw", lh5_file, "pd", with_units=True)
```

[17]:
```
    packet_id  ievt   timestamp  numtraces  tracelist  baseline  energy  \
0           1     0  0.79465985          1       [53]     13722    3304
```

(continues on next page)

(continued from previous page)

```
1           2     0   0.7968994          1       [60]     13044    8642
2           3     0  0.79960424          1       [40]     13508    9177
3           4     0  0.79960424          1       [41]     11891   13015
4           5     1   0.8016167          1       [60]     14353    3794
..        ...   ...         ...        ...        ...       ...     ...
95         96    30  0.96594805          1       [53]     15597    5361
96         97    44   0.9710506          1       [60]     14599    3748
97         98    31  0.97331905          1       [53]     15438   37877
98         99     3    0.974689          1       [30]      9931    6014
99        100    32   0.9786208          1       [53]     17013    3410

    channel  wf_max        wf_std  waveform_t0  waveform_dt  \
0        53   16352   1028.181519          0.0         16.0
1        60   20549   3084.801758          0.0         16.0
2        40   20119   2849.593750          0.0         16.0
3        41   21215   4023.562256          0.0         16.0
4        60   17150   1183.935913          0.0         16.0
..      ...     ...           ...          ...          ...
95       53   29471    947.814392          0.0         16.0
96       60   17324   1151.771362          0.0         16.0
97       53   42319  11780.090820          0.0         16.0
98       30   14317   1876.940308          0.0         16.0
99       53   19567   1065.333130          0.0         16.0

                             waveform_values
0   [13712 13712 13683 ... 15445 15380 15400]
1   [13072 13072 12992 ... 18842 18819 18806]
2   [13575 13575 13496 ... 18409 18384 18457]
3   [11862 11862 11808 ... 18946 18861 18834]
4   [14432 14432 14409 ... 16380 16403 16410]
..                                        ...
95  [27818 27818 27755 ... 26594 26562 26493]
96  [14471 14471 14515 ... 16583 16582 16609]
97  [15405 15405 15366 ... 36208 36214 36233]
98     [9962 9962 9949 ... 13269 13326 13269]
99  [16918 16918 16962 ... 18715 18877 18933]

[100 rows x 13 columns]
```

Writing data to disk

Let's start by creating some LGDOs:

[18]:
```
from lgdo import Array, Scalar, WaveformTable
import numpy as np

rng = np.random.default_rng(12345)

scalar = Scalar("made with legend-pydataobj!")
array = Array(rng.random(size=10))
wf_table = WaveformTable(values=rng.integers(low=1000, high=5000, size=(10, 1000)))
```

The `write()` [docs] function makes it possible to write LGDO objects on disk. Let's start by writing `scalar` with name `message` in a file named `my_data.lh5` in the current directory:

[19]:
```
lh5.write(scalar, name="message", lh5_file="my_objects.lh5", wo_mode="overwrite_file")
```

Let's now inspect the file contents:

[20]:
```
lh5.show("my_objects.lh5")
```

```
/
└── message · string
```

The string object has been written at the root of the file `/`. Let's now write also `array` and `wf_table`, this time in a HDF5 group called `closet`:

[21]:
```
lh5.write(array, name="numbers", group="closet", lh5_file="my_objects.lh5")
lh5.write(wf_table, name="waveforms", group="closet", lh5_file="my_objects.lh5")
lh5.show("my_objects.lh5")
```

```
/
├── closet · HDF5 group
│   ├── numbers · array<1>{real}
│   └── waveforms · table{t0,dt,values}
│       ├── dt · array<1>{real}
│       ├── t0 · array<1>{real}
│       └── values · array_of_equalsized_arrays<1,1>{real}
└── message · string
```

Everything looks right!

Note: `lh5.write()` allows for more advanced usage, like writing only some rows of the input object or appending to existing array-like structures. Have a look at the [docs] for more information.

This page has been automatically generated by nbsphinx and can be run as a Jupyter notebook available in the legend-pydataobj repository.

2.2.2 Data compression

legend-pydataobj gives the user a lot of flexibility in choosing how to compress LGDOs, on disk or in memory, through traditional HDF5 filters or custom waveform compression algorithms.

[1]:
```
import lgdo
from lgdo import lh5
import numpy as np
```

Let's start by creating a dummy LGDO Table:

[2]:
```
data = lgdo.Table(
    size=1000,
    col_dict={
        "col1": lgdo.Array(np.arange(0, 100, 0.1)),
        "col2": lgdo.Array(np.random.rand(1000)),
```

(continues on next page)

(continued from previous page)

```
    },
)
data
```

```
[2]: Table(dict={'col1': Array([ 0.  0.1 ... 99.8 99.9], attrs={'datatype': 'array<1>{real}'}),
↪ 'col2': Array([0.9837063 0.02993754 ... 0.71079581 0.40897818], attrs={'datatype':
↪'array<1>{real}'})}, attrs={'datatype': 'table{col1,col2}'})
```

and writing it to disk with default settings:

```
[3]: lh5.write(data, "data", "data.lh5", wo_mode="of")
lh5.show("data.lh5")
```

```
/
└── data · table{col1,col2}
    ├── col1 · array<1>{real}
    └── col2 · array<1>{real}
```

Let's inspect the data on disk:

```
[4]: import h5py


def show_h5ds_opts(obj):
    with h5py.File("data.lh5") as f:
        print(obj)
        for attr in ["compression", "compression_opts", "shuffle", "chunks"]:
            print(">", attr, ":", f[obj].__getattribute__(attr))
        print("> size :", f[obj].id.get_storage_size(), "B")


show_h5ds_opts("data/col1")
```

```
data/col1
> compression : gzip
> compression_opts : 4
> shuffle : True
> chunks : (1000,)
> size : 494 B
```

Looks like the data is compressed with Gzip (compression level 4) by default! This default setting is stored in the global `DEFAULT_HDF5_SETTINGS` variable:

```
[5]: lh5.DEFAULT_HDF5_SETTINGS
```

```
[5]: {'shuffle': True, 'compression': 'gzip'}
```

Which specifies the default keyword arguments forwarded to h5py.Group.create_dataset() and can be overridden by the user

Examples:

```
[6]: # use another built-in filter
lh5.DEFAULT_HDF5_SETTINGS = {"compression": "lzf"}
```

(continues on next page)

(continued from previous page)

```
# specify filter name and options
lh5.DEFAULT_HDF5_SETTINGS = {"compression": "gzip", "compression_opts": 7}

# specify a registered filter provided by hdf5plugin
import hdf5plugin

lh5.DEFAULT_HDF5_SETTINGS = {"compression": hdf5plugin.Blosc()}

# shuffle bytes before compressing (typically better compression ratio with no␣
↪performance penalty)
lh5.DEFAULT_HDF5_SETTINGS = {"shuffle": True, "compression": "lzf"}
```

Useful resources and lists of HDF5 filters:

- Registered HDF5 filters
- Built-in HDF5 filters from h5py
- Extra filters from hdf5plugin

Let's now re-write the data with the updated default settings:

[7]:
```
lh5.write(data, "data", "data.lh5", wo_mode="of")
show_h5ds_opts("data/col1")
```

```
data/col1
> compression : gzip
> compression_opts : 4
> shuffle : True
> chunks : (1000,)
> size : 494 B
```

Nice. Shuffling bytes before compressing significantly reduced size on disk. Last but not least, `create_dataset()` keyword arguments can be passed to `write()`. They will be forwarded as is, overriding default settings.

[8]:
```
lh5.write(data, "data", "data.lh5", wo_mode="of", shuffle=True, compression="gzip")
show_h5ds_opts("data/col1")
```

```
data/col1
> compression : gzip
> compression_opts : 4
> shuffle : True
> chunks : (1000,)
> size : 494 B
```

Object-specific compression settings are supported via the `hdf5_settings` LGDO attribute:

[9]:
```
data["col2"].attrs["hdf5_settings"] = {"compression": "gzip"}
lh5.write(data, "data", "data.lh5", wo_mode="of")

show_h5ds_opts("data/col1")
show_h5ds_opts("data/col2")
```

```
data/col1
> compression : gzip
> compression_opts : 4
```

(continues on next page)

(continued from previous page)

```
> shuffle : True
> chunks : (1000,)
> size : 494 B
data/col2
> compression : gzip
> compression_opts : 4
> shuffle : True
> chunks : (1000,)
> size : 7021 B
```

We are now storing table columns with different compression settings.

Note: since any h5py.Group.create_dataset() keyword argument can be used in `write()` or set in the `hdf5_settings` attribute, other HDF5 dataset settings can be configured, like the chunk size.

---

[10]:
```
lh5.write(data, "data", "data.lh5", wo_mode="of", chunks=2)
```

Waveform compression

legend-pydataobj implements fast custom waveform compression routines in the lgdo.compression subpackage.

Let's try them out on some waveform test data:

[11]:
```
from legendtestdata import LegendTestData

ldata = LegendTestData()
wfs = lh5.read(
    "geds/raw/waveform",
    ldata.get_path("lh5/LDQTA_r117_20200110T105115Z_cal_geds_raw.lh5"),
)
wfs
```

[11]:
```
WaveformTable(dict={'t0': Array([0. 0. ... 0. 0.], attrs={'datatype': 'array<1>{real}',
↪'units': 'ns'}), 'dt': Array([16. 16. ... 16. 16.], attrs={'datatype': 'array<1>{real}
↪', 'units': 'ns'}), 'values': ArrayOfEqualSizedArrays([[13712 13712 ... 15380 15400]␣
↪[13072 13072 ... 18819 18806] ... [9962 9962 ... 13326 13269] [16918 16918 ... 18877␣
↪18933]], attrs={'datatype': 'array_of_equalsized_arrays<1,1>{real}'})}, attrs={
↪'datatype': 'table{t0,dt,values}'})
```

Let's encode the waveform values with the RadwareSigcompress codec.

Note: samples from these test waveforms must be shifted by -32768 for compatibility reasons, see lgdo.compression.radware.encode().

---

[12]:
```
from lgdo.compression import encode, RadwareSigcompress

enc_values = encode(wfs.values, RadwareSigcompress(codec_shift=-32768))
enc_values
```

[12]:
```
ArrayOfEncodedEqualSizedArrays(encoded_data=VectorOfVectors(flattened_data=Array([0x15␣
↪0xd8 ... 0x8b 0xa4], attrs={'datatype': 'array<1>{real}'}), cumulative_
↪length=Array([6584 12528 ... 609112 615660], attrs={'datatype': 'array<1>{real}'}),␣
↪attrs={'datatype': 'array<1>{array<1>{real}}'}), decoded_size=Scalar(value=5592, attrs=
↪{'datatype': 'real'}), attrs={'datatype': 'array_of_encoded_equalsized_arrays<1,1>
↪{real}', 'codec': 'radware_sigcompress', 'codec_shift': -32768})
```

The output LGDO is an ArrayOfEncodedEqualSizedArrays, which is basically an array of bytes representing the compressed data. How big is this compressed object in bytes?

[13]:
```
enc_values.encoded_data.flattened_data.nda.nbytes
```

[13]:
```
615660
```

How big was the original data structure?

[14]:
```
wfs.values.nda.nbytes
```

[14]:
```
1118400
```

It shrank quite a bit!

Let's now make a `WaveformTable` object wrapping these encoded values, instead of the uncompressed ones, and dump it to disk.

[15]:
```
enc_wfs = lgdo.WaveformTable(
    values=enc_values,
    t0=wfs.t0,
    dt=wfs.dt,
)
lh5.write(enc_wfs, "waveforms", "data.lh5", wo_mode="o")
lh5.show("data.lh5", attrs=True)
```

```
/
├── data · table{col1,col2}
│   ├── col1 · array<1>{real}
│   └── col2 · array<1>{real}
└── waveforms · table{t0,dt,values}
    ├── dt · array<1>{real} — {'units': 'ns'}
    ├── t0 · array<1>{real} — {'units': 'ns'}
    └── values · array_of_encoded_equalsized_arrays<1,1>{real} — {'codec': 'radware_
↪sigcompress', 'codec_shift': -32768}
        ├── decoded_size · real
        └── encoded_data · array<1>{array<1>{real}}
            ├── cumulative_length · array<1>{real}
            └── flattened_data · array<1>{real}
```

The LH5 structure is more complex now. Note how the compression settings are stored as HDF5 attributes.

> **Warning:** HDF5 compression is never applied to waveforms compressed with these custom filters.

Let's try to read the data back in memory:

[16]:
```
obj = lh5.read("waveforms", "data.lh5")
obj.values
```

[16]:
```
ArrayOfEqualSizedArrays([[13712 13712 13683 ... 15445 15380 15400]
                         [13072 13072 12992 ... 18842 18819 18806]
                         [13575 13575 13496 ... 18409 18384 18457]
                         ...
                         [15405 15405 15366 ... 36208 36214 36233]
                         [9962 9962 9949 ... 13269 13326 13269]
                         [16918 16918 16962 ... 18715 18877 18933]], attrs={'codec':
↪'radware_sigcompress', 'codec_shift': -32768, 'datatype': 'array_of_equalsized_arrays
↪<1,1>{real}'})
```

Wait, this is not the compressed data we just wrote to disk, it got decompressed on the fly! It's still possible to just return the compressed data though:

[17]:
```
obj = lh5.read("waveforms", "data.lh5", decompress=False)
obj.values
```

[17]:
```
ArrayOfEncodedEqualSizedArrays(encoded_data=VectorOfVectors(flattened_data=Array([0x15␣
↪0xd8 ... 0x8b 0xa4], attrs={'datatype': 'array<1>{real}'}), cumulative_
↪length=Array([6584 12528 ... 609112 615660], attrs={'datatype': 'array<1>{real}'}),␣
↪attrs={'datatype': 'array<1>{array<1>{real}}'}), decoded_size=Scalar(value=5592, attrs=
↪{'datatype': 'real'}), attrs={'codec': 'radware_sigcompress', 'codec_shift': -32768,
↪'datatype': 'array_of_encoded_equalsized_arrays<1,1>{real}'})
```

And then decompress it manually:

[18]:
```
from lgdo.compression import decode

decode(obj.values)
```

[18]:
```
ArrayOfEqualSizedArrays([[13712 13712 13683 ... 15445 15380 15400]
                         [13072 13072 12992 ... 18842 18819 18806]
                         [13575 13575 13496 ... 18409 18384 18457]
                         ...
                         [15405 15405 15366 ... 36208 36214 36233]
                         [9962 9962 9949 ... 13269 13326 13269]
                         [16918 16918 16962 ... 18715 18877 18933]], attrs={'codec':
↪'radware_sigcompress', 'codec_shift': -32768, 'datatype': 'array_of_equalsized_arrays
↪<1,1>{real}'})
```

Waveform compression settings can also be specified at the LGDO level by attaching a `compression` attribute to the `values` attribute of a `WaveformTable` object:

[19]:
```
from lgdo.compression import ULEB128ZigZagDiff

wfs.values.attrs["compression"] = ULEB128ZigZagDiff()
lh5.write(wfs, "waveforms", "data.lh5", wo_mode="of")

obj = lh5.read("waveforms", "data.lh5", decompress=False)
obj.values.attrs["codec"]
```

[19]:
```
'uleb128_zigzag_diff'
```

Further reading:

- Available waveform compression algorithms
- read() docstring

- write() docstring

This page has been automatically generated by nbsphinx and can be run as a Jupyter notebook available in the legend-pydataobj repository.

2.3 lgdo

2.3.1 lgdo package

LEGEND Data Objects (LGDO) are defined in the LEGEND data format specification. This package serves as the Python implementation of that specification. The general strategy for the implementation is to dress standard Python and NumPy objects with an `attr` dictionary holding LGDO metadata, plus some convenience functions. The basic data object classes are:

- *LGDO*: abstract base class for all LGDOs
- *Scalar*: typed Python scalar. Access data via the `value` attribute
- *Array*: basic `numpy.ndarray`. Access data via the `nda` attribute.
- *FixedSizeArray*: basic `numpy.ndarray`. Access data via the `nda` attribute.
- *ArrayOfEqualSizedArrays*: multi-dimensional `numpy.ndarray`. Access data via the `nda` attribute.
- *VectorOfVectors*: an n-dimensional variable length array of variable length arrays. Implemented as a pair of datasets: `flattened_data` holding the raw data (*Array* or *VectorOfVectors*, if the vector dimension is greater than 2), and `cumulative_length` (always an *Array*) whose i-th element is the sum of the lengths of the vectors with `index <= i`
- *VectorOfEncodedVectors*: an array of variable length *encoded* arrays. Implemented as a *VectorOfVectors* `encoded_data` holding the encoded vectors and an *Array* `decoded_size` specifying the size of each decoded vector. Mainly used to represent a list of compressed waveforms.
- *ArrayOfEncodedEqualSizedArrays*: an array of equal sized encoded arrays. Similar to *VectorOfEncodedVectors* except for `decoded_size`, which is now a scalar.
- *Struct*: a dictionary containing LGDO objects. Derives from `dict`
- *Table*: a *Struct* whose elements ("columns") are all array types with the same length (number of rows)

Currently the primary on-disk format for LGDO object is LEGEND HDF5 (LH5) files. IO is done via the class *lh5_store.LH5Store*. LH5 files can also be browsed easily in python like any HDF5 file using h5py.

Subpackages

lgdo.compression package

Data compression utilities.

This subpackage collects all LEGEND custom data compression (encoding) and decompression (decoding) algorithms.

Available lossless waveform compression algorithms:

- *RadwareSigcompress*, a Python port of the C algorithm *radware-sigcompress* by D. Radford.
- *ULEB128ZigZagDiff* variable-length base-128 encoding of waveform differences.

All waveform compression algorithms inherit from the *WaveformCodec* abstract class.

encode() and *decode()* provide a high-level interface for encoding/decoding *LGDO*s.

```
>>> from lgdo import WaveformTable, compression
>>> wftbl = WaveformTable(...)
>>> enc_wft = compression.encode(wftable, RadwareSigcompress(codec_shift=-23768)
>>> compression.decode(enc_wft) # == wftbl
```

Submodules

lgdo.compression.base module

class lgdo.compression.base.**WaveformCodec**

Bases: object

Base class identifying a waveform compression algorithm.

The *self.codec* property returns a string identifier suitable for labeling encoded data on disk. This identifier is constant for all class instances.

Note: This is an abstract type. The user must provided a concrete subclass.

---

asdict()

Return the dataclass fields as dictionary.

property codec

The waveform codec string identifier.

Will be attached as an attribute to the encoded Waveform values.

lgdo.compression.generic module

lgdo.compression.generic.**_is_codec**(*ident*, *codec*)

Return type
bool

lgdo.compression.generic.**decode**(*obj*, *out_buf=None*)

Decode encoded LGDOs.

Defines decoding behaviors for each implemented waveform encoding algorithm. Expects to find the codec (and its parameters) the arrays where encoded with among the LGDO attributes.

Parameters

- **obj** (*lgdo.VectorOfEncodedVectors | lgdo.ArrayOfEncodedEqualSizedArrays*) – LGDO array type.
- **out_buf** (*lgdo.ArrayOfEqualSizedArrays*) – pre-allocated LGDO for the decoded signals. See documentation of wrapped encoders for limitations.

Return type
lgdo.VectorOfVectors | lgdo.ArrayOfEqualsizedArrays

lgdo.compression.generic.**encode**(*obj*, *codec=None*)

Encode LGDOs with *codec*.

Defines behaviors for each implemented waveform encoding algorithm.

Parameters

- **obj** (*lgdo.VectorOfVectors | lgdo.ArrayOfEqualsizedArrays*) – LGDO array type.
- **codec** (*WaveformCodec | str*) – algorithm to be used for encoding.

Return type
lgdo.VectorOfEncodedVectors | lgdo.ArrayOfEncodedEqualSizedArrays

lgdo.compression.radware module

class lgdo.compression.radware.**RadwareSigcompress**(*codec_shift=0*)

Bases: *WaveformCodec*

radware-sigcompress array codec.

Examples

```
>>> from lgdo.compression import RadwareSigcompress
>>> codec = RadwareSigcompress(codec_shift=-32768)
```

codec_shift: int = 0

Offset added to the input waveform before encoding.

The *radware-sigcompress* algorithm is limited to encoding of 16-bit integer values. In certain cases (notably, with *unsigned* 16-bit integer values), shifting incompatible data by a fixed amount circumvents the issue.

@numba.jit lgdo.compression.radware.**_get_high_u16**(*x*)

Return type
uint16

@numba.jit lgdo.compression.radware.**_get_hton_u16**(*a*, *i*)

Read unsigned 16-bit integer values from an array of unsigned 8-bit integers.

The first two most significant bytes of the values must be stored contiguously in *a* with big-endian order.

Return type
uint16

@numba.jit lgdo.compression.radware.**_get_low_u16**(*x*)

Return type
uint16

@numba.guvectorize lgdo.compression.radware._radware_sigcompress_decode(*sig_in*, *sig_out*, *shift*, *siglen*, *_mask=array([0, 1, 3, ..., 16383, 32767, 65535], dtype=uint16)*)

- *Options:* boundscheck=False, nopython=True, cache=True
- *Precompiled signatures:* BHiIH->, BIiIH->, BLiIH->, BhiIH->, BiiIH->, BliIH->

Deompress a digital signal.

After decoding, the signal values are shifted by -shift to restore the original waveform. The dtype of *sig_out* must be large enough to contain it.

Almost literal translations of decompress_signal() from the *radware-sigcompress* v1.0 C-code by David Radford[1]. See _radware_sigcompress_encode() for a list of changes to the original algorithm.

Parameters

- **sig_in** (*ndarray[Any, dtype[uint8]]*) – array holding the input, compressed signal. In the original code, an array of 16-bit unsigned integers was expected.
- **sig_out** (*ndarray[Any, dtype[_ScalarType_co]]*) – pre-allocated array for the decompressed signal. In the original code, an array of 16-bit integers was expected.
- **shift** (*int32*) – the value the original signal(s) was shifted before compression. The value is *subtracted* from samples in *sig_out* right after decoding.

Returns
length – length of output, decompressed signal.

Return type
None

@numba.guvectorize lgdo.compression.radware._radware_sigcompress_encode(*sig_in*, *sig_out*, *shift*, *siglen*, *_mask=array([0, 1, 3, ..., 16383, 32767, 65535], dtype=uint16)*)

- *Options:* boundscheck=False, nopython=True, cache=True
- *Precompiled signatures:* HBiIH->, IBiIH->, LBiIH->, hBiIH->, iBiIH->, lBiIH->

Compress a digital signal.

Shifts the signal values by +shift and internally interprets the result as numpy.int16. Shifted signals must be therefore representable as numpy.int16, for lossless compression.

Note: The algorithm also computes the first derivative of the input signal, which cannot always be represented as a 16-bit integer. In such cases, overflows occur, but they seem to be innocuous.

Almost literal translations of compress_signal() from the *radware-sigcompress* v1.0 C-code by David Radford[1]. Summary of changes:

- Shift the input signal by *shift* before encoding.
- Store encoded, numpy.uint16 signal as an array of bytes (numpy.ubyte), in big-endian ordering.

[1] radware-sigcompress source code. released under MIT license *[Copyright (c) 2018, David C. Radford <radforddc@ornl.gov>].*

- Declare mask globally to avoid extra memory allocation.
- Enable hardware-vectorization with Numba (`numba.guvectorize()`).
- Add a couple of missing array boundary checks.

Parameters

- **sig_in** (*ndarray[Any, dtype[_ScalarType_co]]*) – array of integers holding the input signal. In the original C code, an array of 16-bit integers was expected.
- **sig_out** (*ndarray[Any, dtype[uint8]]*) – pre-allocated array for the unsigned 8-bit encoded signal. In the original C code, an array of unsigned 16-bit integers was expected.
- **shift** (*int32*) – value to be added to *sig_in* before compression.
- **siglen** (*uint32*) – array that will hold the lengths of the compressed signals.

Returns
length – number of bytes in the encoded signal

Return type
None

@numba.jit lgdo.compression.radware.**_set_high_u16**(*x*, *y*)

Return type
uint32

@numba.jit lgdo.compression.radware.**_set_hton_u16**(*a*, *i*, *x*)

Store an unsigned 16-bit integer value in an array of unsigned 8-bit integers.

The first two most significant bytes from *x* are stored contiguously in *a* with big-endian order.

Return type
int

@numba.jit lgdo.compression.radware.**_set_low_u16**(*x*, *y*)

Return type
uint32

lgdo.compression.radware.**decode**(*sig_in*, *sig_out=None*, *shift=0*)

Decompress digital signal(s) with *radware-sigcompress*.

Wraps `_radware_sigcompress_decode()` and adds support for decoding LGDOs. Resizes the decoded signals to their actual length.

Note: If *sig_in* is a NumPy array, no resizing (along the last dimension) of *sig_out* to its actual length is performed. Not even of the internally allocated one. If a pre-allocated *ArrayOfEqualSizedArrays* is provided, it won't be resized too. The internally allocated *ArrayOfEqualSizedArrays* *sig_out* has instead always the correct size.

Because of the current (hardware vectorized) implementation, providing a pre-allocated *VectorOfVectors* as *sig_out* is not possible.

Parameters

- **sig_in** (*NDArray[ubyte] | lgdo.VectorOfEncodedVectors | lgdo.ArrayOfEncodedEqualSizedArrays*) – array(s) holding the input, compressed signal(s). Output of *encode()*.
- **sig_out** (*NDArray | lgdo.ArrayOfEqualSizedArrays*) – pre-allocated array(s) for the decompressed signal(s). If not provided, will allocate a 32-bit integer array(s) structure.
- **shift** (*int32*) – the value the original signal(s) was shifted before compression. The value is *subtracted* from samples in *sig_out* right after decoding.

Returns

sig_out, nbytes | *LGDO* – given pre-allocated structure or new structure of 32-bit integers, plus the number of bytes (length) of the decoded signal.

Return type

(NDArray, NDArray[uint32]) | lgdo.VectorOfVectors | lgdo.ArrayOfEqualSizedArrays

See also:

`_radware_sigcompress_decode`

lgdo.compression.radware.**encode**(*sig_in*, *sig_out=None*, *shift=0*)

Compress digital signal(s) with *radware-sigcompress*.

Wraps `_radware_sigcompress_encode()` and adds support for encoding LGDO arrays. Resizes the encoded array to its actual length.

Note: If *sig_in* is a NumPy array, no resizing of *sig_out* is performed. Not even of the internally allocated one.

Because of the current (hardware vectorized) implementation, providing a pre-allocated *VectorOfEncodedVectors* or *ArrayOfEncodedEqualSizedArrays* as *sig_out* is not possible.

Note: The compression algorithm internally interprets the input waveform values as 16-bit integers. Make sure that your signal can be safely cast to such a numeric type. If not, you may want to apply a *shift* to the waveform.

Parameters

- **sig_in** (*NDArray | lgdo.VectorOfVectors | lgdo.ArrayOfEqualSizedArrays*) – array(s) holding the input signal(s).
- **sig_out** (*NDArray[ubyte]*) – pre-allocated unsigned 8-bit integer array(s) for the compressed signal(s). If not provided, a new one will be allocated.
- **shift** (*int32*) – value to be added to *sig_in* before compression.

Returns

sig_out, nbytes | *LGDO* – given pre-allocated *sig_out* structure or new structure of unsigned 8-bit integers, plus the number of bytes (length) of the encoded signal. If *sig_in* is an *LGDO*, only a newly allocated *VectorOfEncodedVectors* or *ArrayOfEncodedEqualSizedArrays* is returned.

Return type

(NDArray[ubyte], NDArray[uint32]) | lgdo.VectorOfEncodedVectors | lgdo.ArrayOfEncodedEqualSizedArrays

See also:

`_radware_sigcompress_encode`

lgdo.compression.utils module

lgdo.compression.utils.**str2wfcodec**(*expr*)

Eval strings containing *WaveformCodec* declarations.

Simple tool to avoid using `eval()`. Used to read *WaveformCodec* declarations configured in JSON files.

Return type
WaveformCodec

lgdo.compression.varlen module

Variable-length code compression algorithms.

class lgdo.compression.varlen.**ULEB128ZigZagDiff**(*codec='uleb128_zigzag_diff'*)

Bases: *WaveformCodec*

ZigZag[2] encoding followed by Unsigned Little Endian Base 128 (ULEB128)[3] encoding of array differences.

codec: str = 'uleb128_zigzag_diff'

lgdo.compression.varlen.**decode**(*sig_in*, *sig_out=None*)

Deompress digital signal(s) with a variable-length encoding of its derivative.

Wraps *uleb128_zigzag_diff_array_decode()* and adds support for decoding LGDOs.

Note: If *sig_in* is a NumPy array, no resizing (along the last dimension) of *sig_out* to its actual length is performed. Not even of the internally allocated one. If a pre-allocated *ArrayOfEqualSizedArrays* is provided, it won't be resized too. The internally allocated *ArrayOfEqualSizedArrays* *sig_out* has instead always the correct size.

Because of the current (hardware vectorized) implementation, providing a pre-allocated *VectorOfVectors* as *sig_out* is not possible.

Parameters

- **sig_in** (*(NDArray[ubyte], NDArray[uint32]) | lgdo.VectorOfEncodedVectors | lgdo.ArrayOfEncodedEqualSizedArrays*) – array(s) holding the input, compressed signal(s). Output of *encode()*.
- **sig_out** (*NDArray | lgdo.ArrayOfEqualSizedArrays*) – pre-allocated array(s) for the decompressed signal(s). If not provided, will allocate a 32-bit integer array(s) structure.

Returns
sig_out, nbytes | LGDO – given pre-allocated structure or new structure of 32-bit integers, plus the number of bytes (length) of the decoded signal.

Return type
(NDArray, NDArray[uint32]) | lgdo.VectorOfVectors | lgdo.ArrayOfEqualSizedArrays

See also:

uleb128_zigzag_diff_array_decode

[2] https://wikipedia.org/wiki/Variable-length_quantity#Zigzag_encoding
[3] https://wikipedia.org/wiki/LEB128#Unsigned_LEB128

lgdo.compression.varlen.**encode**(*sig_in*, *sig_out=None*)

Compress digital signal(s) with a variable-length encoding of its derivative.

Wraps *uleb128_zigzag_diff_array_encode()* and adds support for encoding LGDOs.

Note: If *sig_in* is a NumPy array, no resizing of *sig_out* is performed. Not even of the internally allocated one.

Because of the current (hardware vectorized) implementation, providing a pre-allocated *VectorOfEncodedVectors* or *ArrayOfEncodedEqualSizedArrays* as *sig_out* is not possible.

Parameters

- **sig_in** (*NDArray | lgdo.VectorOfVectors | lgdo.ArrayOfEqualSizedArrays*) – array(s) holding the input signal(s).
- **sig_out** (*NDArray[ubyte]*) – pre-allocated unsigned 8-bit integer array(s) for the compressed signal(s). If not provided, a new one will be allocated.

Returns

sig_out, nbytes | LGDO – given pre-allocated *sig_out* structure or new structure of unsigned 8-bit integers, plus the number of bytes (length) of the encoded signal. If *sig_in* is an *LGDO*, only a newly allocated *VectorOfEncodedVectors* or *ArrayOfEncodedEqualSizedArrays* is returned.

Return type

(NDArray[ubyte], NDArray[uint32]) | lgdo.VectorOfEncodedVectors | lgdo.ArrayOfEncodedEqualSizedArrays

See also:

uleb128_zigzag_diff_array_encode

@numba.jit lgdo.compression.varlen.**uleb128_decode**(*encx*)

Decode a variable-length integer into an unsigned integer.

Implements the Unsigned Little Endian Base-128 decoding[Page 25, 3]. Only encoded positive numbers are expected, as no *two's complement* is applied.

Parameters

encx (*NDArray[ubyte]*) – the encoded varint as a NumPy array of bytes.

Returns

x, nread – the decoded value and the number of bytes read from the input array.

Return type

(int, int)

@numba.jit lgdo.compression.varlen.**uleb128_encode**(*x*, *encx*)

Compute a variable-length representation of an unsigned integer.

Implements the Unsigned Little Endian Base-128 encoding[Page 25, 3]. Only positive numbers are expected, as no *two's complement* is applied.

Parameters

- **x** (*int*) – the number to be encoded.
- **encx** (*ndarray[Any, dtype[uint8]]*) – the encoded varint as a NumPy array of bytes.

Returns

nbytes – size of varint in bytes

Return type
int

@numba.guvectorize lgdo.compression.varlen.**uleb128_zigzag_diff_array_decode**(*sig_in*, *nbytes*, *sig_out*, *siglen*)

- *Options:* boundscheck=False, nopython=True, cache=True
- *Precompiled signatures:* BIHI->, BIII->, BILI->, BIhI->, BIiI->, BIlI->

Decode an array of variable-length integers.

The algorithm inverts `uleb128_zigzag_diff_array_encode()` by decoding the variable-length binary data in *sig_in* with *uleb128_decode()*, then reconstructing the original signal derivative with *zigzag_decode()* and finally computing its cumulative (i.e. the original signal).

Parameters

- **sig_in** (*ndarray[Any, dtype[uint8]]*) – the array of bytes encoding the variable-length integers.
- **nbytes** (*int*) – the number of bytes to read from *sig_in* (stored in the first index of this array).
- **sig_out** (*ndarray[Any, dtype[int]]*) – pre-allocated array for the output decoded signal.
- **siglen** (*int*) – the length of the decoded signal, (stored in the first index of this array).

See also:

uleb128_zigzag_diff_array_encode

@numba.guvectorize lgdo.compression.varlen.**uleb128_zigzag_diff_array_encode**(*sig_in*, *sig_out*, *nbytes*)

- *Options:* boundscheck=False, nopython=True, cache=True
- *Precompiled signatures:* HBI->, IBI->, LBI->, hBI->, iBI->, lBI->

Encode an array of integer numbers.

The algorithm computes the derivative (prepending 0 first) of *sig_in*, maps it to positive numbers by applying *zigzag_encode()* and finally computes its variable-length binary representation with *uleb128_encode()*.

The encoded data is stored in *sig_out* as an array of bytes. The number of bytes written is stored in *nbytes*. The actual encoded data can therefore be found in `sig_out[:nbytes]`.

Parameters

- **sig_in** (*ndarray[Any, dtype[int]]*) – the input array of integers.
- **sig_out** (*ndarray[Any, dtype[uint8]]*) – pre-allocated bytes array for the output encoded data.
- **nbytes** (*int*) – pre-allocated output array holding the number of bytes written (stored in the first index).

See also:

uleb128_zigzag_diff_array_decode

@numba.vectorize lgdo.compression.varlen.**zigzag_decode**(*x*)

- *Precompiled signatures:* L->l, I->i, H->h

ZigZag-decode[Page 25, 2] signed integer numbers.

Return type
int | *ndarray*[*Any*, *dtype*[int]]

@numba.vectorize lgdo.compression.varlen.**zigzag_encode**(*x*)

- *Precompiled signatures:* `l->L`, `i->I`, `h->H`

ZigZag-encode[Page 25, 2] signed integer numbers.

Return type
int | *ndarray*[*Any*, *dtype*[int]]

lgdo.lh5 package

Routines from reading and writing LEGEND Data Objects in HDF5 files. Currently the primary on-disk format for LGDO object is LEGEND HDF5 (LH5) files. IO is done via the class *store.LH5Store*. LH5 files can also be browsed easily in python like any HDF5 file using h5py.

Subpackages

lgdo.lh5._serializers package

Subpackages

lgdo.lh5._serializers.read package

Submodules

lgdo.lh5._serializers.read.array module

lgdo.lh5._serializers.read.array.**_h5_read_array**(*name*, *h5f*, ***kwargs*)

lgdo.lh5._serializers.read.array.**_h5_read_array_generic**(*type_*, *name*, *h5f*, ***kwargs*)

lgdo.lh5._serializers.read.array.**_h5_read_array_of_equalsized_arrays**(*name*, *h5f*, ***kwargs*)

lgdo.lh5._serializers.read.array.**_h5_read_fixedsize_array**(*name*, *h5f*, ***kwargs*)

lgdo.lh5._serializers.read.composite module

lgdo.lh5._serializers.read.composite.**_h5_read_lgdo**(*name*, *h5f*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *field_mask=None*, *obj_buf=None*, *obj_buf_start=0*, *decompress=True*)

lgdo.lh5._serializers.read.composite.**_h5_read_struct**(*name*, *h5f*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *field_mask=None*, *decompress=True*)

lgdo.lh5._serializers.read.composite.**_h5_read_table**(*name*, *h5f*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *field_mask=None*, *obj_buf=None*, *obj_buf_start=0*, *decompress=True*)

lgdo.lh5._serializers.read.encoded module

lgdo.lh5._serializers.read.encoded.**_h5_read_array_of_encoded_equalsized_arrays**(*name*, *h5f*, ***kwargs*)

lgdo.lh5._serializers.read.encoded.**_h5_read_encoded_array**(*lgdotype*, *name*, *h5f*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *obj_buf=None*, *obj_buf_start=0*, *decompress=True*)

lgdo.lh5._serializers.read.encoded.**_h5_read_vector_of_encoded_vectors**(*name*, *h5f*, ***kwargs*)

lgdo.lh5._serializers.read.ndarray module

lgdo.lh5._serializers.read.ndarray.**_h5_read_ndarray**(*name*, *h5f*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *obj_buf=None*, *obj_buf_start=0*)

lgdo.lh5._serializers.read.scalar module

lgdo.lh5._serializers.read.scalar.**_h5_read_scalar**(*name*, *h5f*, *obj_buf=None*)

lgdo.lh5._serializers.read.utils module

lgdo.lh5._serializers.read.utils.**check_obj_buf_attrs**(*attrs*, *new_attrs*, *file*, *name*)

lgdo.lh5._serializers.read.vector_of_vectors module

lgdo.lh5._serializers.read.vector_of_vectors.**_h5_read_vector_of_vectors**(*name*, *h5f*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *obj_buf=None*, *obj_buf_start=0*)

@numba.jit lgdo.lh5._serializers.read.vector_of_vectors.**_make_fd_idx**(*starts*, *stops*, *idx*)

lgdo.lh5._serializers.write package

Submodules

lgdo.lh5._serializers.write.array module

lgdo.lh5._serializers.write.array.**_h5_write_array**(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

lgdo.lh5._serializers.write.composite module

lgdo.lh5._serializers.write.composite.**_h5_write_lgdo**(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

lgdo.lh5._serializers.write.composite.**_h5_write_struct**(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

lgdo.lh5._serializers.write.scalar module

lgdo.lh5._serializers.write.scalar.**_h5_write_scalar**(*obj*, *name*, *lh5_file*, *group='/'*, *wo_mode='append'*)

lgdo.lh5._serializers.write.vector_of_vectors module

lgdo.lh5._serializers.write.vector_of_vectors.**_h5_write_vector_of_vectors**(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

Submodules

lgdo.lh5.core module

lgdo.lh5.core.**read**(*name*, *lh5_file*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *field_mask=None*, *obj_buf=None*, *obj_buf_start=0*, *decompress=True*)

Read LH5 object data from a file.

Note: Use the `idx` parameter to read out particular rows of the data. The `use_h5idx` flag controls whether *only* those rows are read from disk or if the rows are indexed after reading the entire object. Reading individual rows can be orders of magnitude slower than reading the whole object and then indexing the desired rows.

The default behavior (`use_h5idx=False`) is to use slightly more memory for a much faster read. See legend-pydataobj/issues/#29 for additional information.

Parameters

- **name** (*str*) – Name of the LH5 object to be read (including its group path).
- **lh5_file** (*str | File | Sequence[str | File]*) – The file(s) containing the object to be read out. If a list of files, array-like object data will be concatenated into the output object.
- **start_row** (*int*) – Starting entry for the object read (for array-like objects). For a list of files, only applies to the first file.
- **n_rows** (*int*) – The maximum number of rows to read (for array-like objects). The actual number of rows read will be returned as one of the return values (see below).
- **idx** (*_SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes] | None*) – For NumPy-style "fancying indexing" for the read to select only some rows, e.g. after applying some cuts to particular columns. Only selection along the first axis is supported, so tuple arguments must be one-tuples. If *n_rows* is not false, *idx* will be truncated to *n_rows* before reading. To use with a list of files, can pass in a list of *idx*'s (one for each file) or use a long contiguous list (e.g. built from a previous identical read). If used in conjunction with *start_row* and *n_rows*, will be sliced to obey those constraints, where *n_rows* is interpreted as the (max) number of *selected* values (in *idx*) to be read out. Note that the `use_h5idx` parameter controls some behaviour of the read and that the default behavior (`use_h5idx=False`) prioritizes speed over a small memory penalty.
- **use_h5idx** (*bool*) – `True` will directly pass the `idx` parameter to the underlying `h5py` call such that only the selected rows are read directly into memory, which conserves memory at the cost of speed. There can be a significant penalty to speed for larger files (1 - 2 orders of magnitude longer time). `False` (default) will read the entire object into memory before performing the indexing. The default is much faster but requires additional memory, though a relatively small amount in the typical use case. It is recommended to leave this parameter as its default.
- **field_mask** (*Mapping[str, bool] | Sequence[str] | None*) – For tables and structs, determines which fields get read out. Only applies to immediate fields of the requested objects. If a dict is used, a default dict will be made with the default set to the opposite of the first element in the dict. This way if one specifies a few fields at `False`, all but those fields will be read out, while if one specifies just a few fields as `True`, only those fields will be read out. If a list is provided, the listed fields will be set to `True`, while the rest will default to `False`.
- **obj_buf** (*LGDO | None*) – Read directly into memory provided in *obj_buf*. Note: the buffer will be expanded to accommodate the data requested. To maintain the buffer length, send in `n_rows = len(obj_buf)`.
- **obj_buf_start** (*int*) – Start location in `obj_buf` for read. For concatenating data to array-like objects.
- **decompress** (*bool*) – Decompress data encoded with LGDO's compression routines right after reading. The option has no effect on data encoded with HDF5 built-in filters, which is always decompressed upstream by HDF5.

Returns
(object, n_rows_read) – *object* is the read-out object *n_rows_read* is the number of rows successfully read out. Essential for arrays when the amount of data is smaller than the object buffer. For scalars and structs *n_rows_read* will be``1``. For tables it is redundant with `table.loc`. If *obj_buf* is `None`, only *object* is returned.

Return type
LGDO | tuple[LGDO, int]

lgdo.lh5.core.**read_as**(*name*, *lh5_file*, *library*, ***kwargs*)

Read LH5 data from disk straight into a third-party data format view.

This function is nothing more than a shortcut chained call to *read()* and to *LGDO.view_as()*.

Parameters

- **name** (*str*) – LH5 object name on disk.
- **lh5_file** (*str | File | Sequence[str | File]*) – LH5 file name.
- **library** (*str*) – string ID of the third-party data format library (`np`, `pd`, `ak`, etc).

Return type
Any

See also:

read, *LGDO.view_as*

lgdo.lh5.core.**write**(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

Write an LGDO into an LH5 file.

If the *obj* *LGDO* has a *compression* attribute, its value is interpreted as the algorithm to be used to compress *obj* before writing to disk. The type of *compression* can be:

string, kwargs dictionary, hdf5plugin filter
interpreted as the name of a built-in or custom HDF5 compression filter (`"gzip"`, `"lzf"`, `hdf5plugin` filter object etc.) and passed directly to `h5py.Group.create_dataset()`.

***WaveformCodec* object**
If *obj* is a *WaveformTable* and `obj.values` holds the attribute, compress `values` using this algorithm. More documentation about the supported waveform compression algorithms at *lgdo.compression*.

If the *obj* *LGDO* has a *hdf5_settings* attribute holding a dictionary, it is interpreted as a list of keyword arguments to be forwarded directly to `h5py.Group.create_dataset()` (exactly like the first format of *compression* above). This is the preferred way to specify HDF5 dataset options such as chunking etc. If compression options are specified, they take precedence over those set with the *compression* attribute.

Note: The *compression* LGDO attribute takes precedence over the default HDF5 compression settings. The *hdf5_settings* attribute takes precedence over *compression*. These attributes are not written to disk.

Note: HDF5 compression is skipped for the *encoded_data.flattened_data* dataset of *VectorOfEncodedVectors* and *ArrayOfEncodedEqualSizedArrays*.

Parameters

- **obj** (LGDO) – LH5 object. if object is array-like, writes *n_rows* starting from *start_row* in *obj*.
- **name** (*str*) – name of the object in the output HDF5 file.
- **lh5_file** (*str | File*) – HDF5 file name or h5py.File object.
- **group** (*str | Group*) – HDF5 group name or h5py.Group object in which *obj* should be written.
- **start_row** (*int*) – first row in *obj* to be written.
- **n_rows** (*int | None*) – number of rows in *obj* to be written.
- **wo_mode** (*str*) –
  - write_safe or w: only proceed with writing if the object does not already exist in the file.
  - append or a: append along axis 0 (the first dimension) of array-like objects and array-like subfields of structs. Scalar objects get overwritten.
  - overwrite or o: replace data in the file if present, starting from *write_start*. Note: overwriting with *write_start* = end of array is the same as append.
  - overwrite_file or of: delete file if present prior to writing to it. *write_start* should be 0 (its ignored).
  - append_column or ac: append columns from an Table *obj* only if there is an existing Table in the *lh5_file* with the same *name* and size. If the sizes don't match, or if there are matching fields, it errors out.
- **write_start** (*int*) – row in the output file (if already existing) to start overwriting from.
- ****h5py_kwargs** – additional keyword arguments forwarded to h5py.Group.create_dataset() to specify, for example, an HDF5 compression filter to be applied before writing non-scalar datasets. **Note: `compression` Ignored if compression is specified as an `obj` attribute.**

lgdo.lh5.datatype module

lgdo.lh5.datatype.**_lgdo_datatype_map: dict[str, *LGDO*] = {<class 'lgdo.types.array.Array'>: '^array<\\d+>\\{.+\\}$', <class 'lgdo.types.arrayofequalsizedarrays.ArrayOfEqualSizedArrays'>: '^array_of_equalsized_arrays<1,1>\\{.+\\}$', <class 'lgdo.types.encoded.ArrayOfEncodedEqualSizedArrays'>: '^array_of_encoded_equalsized_arrays<1,1>\\{.+\\}$', <class 'lgdo.types.encoded.VectorOfEncodedVectors'>: '^array<1>\\{encoded_array<1>\\{.+\\}\\}$', <class 'lgdo.types.fixedsizearray.FixedSizeArray'>: '^fixedsize_array<\\d+>\\{.+\\}$', <class 'lgdo.types.scalar.Scalar'>: '^real$|^bool$|^complex$|^bool$|^string$', <class 'lgdo.types.struct.Struct'>: '^struct\\{.*\\}$', <class 'lgdo.types.table.Table'>: '^table\\{.*\\}$', <class 'lgdo.types.vectorofvectors.VectorOfVectors'>: '^array<1>\\{array<1>\\{.+\\}\\}$'}**

Mapping between LGDO types and regular expression defining the corresponding datatype string

lgdo.lh5.datatype.**datatype**(*expr*)

Return the LGDO type corresponding to a datatype string.

Return type
type

lgdo.lh5.datatype.**get_nested_datatype_string**(*expr*)

Matches the content of the outermost curly brackets.

Return type
str

lgdo.lh5.datatype.**get_struct_fields**(*expr*)

Returns a list of *Struct* fields, given its datatype string.

Return type
list[str]

lgdo.lh5.exceptions module

exception lgdo.lh5.exceptions.**LH5DecodeError**(*message*, *file*, *obj*)

Bases: Exception

exception lgdo.lh5.exceptions.**LH5EncodeError**(*message*, *file*, *group*, *name*)

Bases: Exception

lgdo.lh5.iterator module

class lgdo.lh5.iterator.**LH5Iterator**(*lh5_files*, *groups*, *base_path=''*, *entry_list=None*, *entry_mask=None*, *field_mask=None*, *buffer_len=3200*, *friend=None*)

Bases: Iterator

A class for iterating through one or more LH5 files, one block of entries at a time. This also accepts an entry list/mask to enable event selection, and a field mask.

This class can be used either for random access:

```
>>> lh5_obj, n_rows = lh5_it.read(entry)
```

to read the block of entries starting at entry. In case of multiple files or the use of an event selection, entry refers to a global event index across files and does not count events that are excluded by the selection.

This can also be used as an iterator:

```
>>> for lh5_obj, entry, n_rows in LH5Iterator(...):
>>>     # do the thing!
```

This is intended for if you are reading a large quantity of data but want to limit your memory usage (particularly when reading in waveforms!). The `lh5_obj` that is read by this class is reused in order to avoid reallocation of memory; this means that if you want to hold on to data between reads, you will have to copy it somewhere!

Parameters

- **lh5_files** (*str | list[str]*) – file or files to read from. May include wildcards and environment variables.
- **groups** (*str | list[str]*) – HDF5 group(s) to read. If a list is provided for both lh5_files and group, they must be the same size. If a file is wild-carded, the same group will be assigned to each file found

- **entry_list** (*list[int] | list[list[int]] | None*) – list of entry numbers to read. If a nested list is provided, expect one top-level list for each file, containing a list of local entries. If a list of ints is provided, use global entries.
- **entry_mask** (*list[bool] | list[list[bool]] | None*) – mask of entries to read. If a list of arrays is provided, expect one for each file. Ignore if a selection list is provided.
- **field_mask** (*dict[str, bool] | list[str] | tuple[str] | None*) – mask of which fields to read. See `LH5Store.read()` for more details.
- **buffer_len** (*int*) – number of entries to read at a time while iterating through files.
- **friend** (*Iterator | None*) – a "friend" LH5Iterator that will be read in parallel with this. The friend should have the same length and entry list. A single LH5 table containing columns from both iterators will be returned.

_abc_impl = <_abc._abc_data object>

_get_file_cumentries(*i_file*)

Helper to get cumulative iterator entries in file

Return type
int

_get_file_cumlen(*i_file*)

Helper to get cumulative file length of file

Return type
int

get_file_entrylist(*i_file*)

Helper to get entry list for file

Return type
ndarray

get_global_entrylist()

Get global entry list, constructing it if needed

Return type
ndarray

read(*entry*)

Read the nextlocal chunk of events, starting at entry. Return the LH5 buffer and number of rows read.

Return type
tuple[Array | Scalar | Struct | VectorOfVectors, int]

reset_field_mask(*mask*)

Replaces the field mask of this iterator and any friends with mask

lgdo.lh5.store module

This module implements routines from reading and writing LEGEND Data Objects in HDF5 files.

class lgdo.lh5.store.**LH5Store**(*base_path=''*, *keep_open=False*)

Bases: object

Class to represent a store of LEGEND HDF5 files. The two main methods implemented by the class are *read()* and *write()*.

Examples

```
>>> from lgdo import LH5Store
>>> store = LH5Store()
>>> obj, _ = store.read("/geds/waveform", "file.lh5")
>>> type(obj)
lgdo.waveformtable.WaveformTable
```

Parameters

- **base_path** (*str*) – directory path to prepend to LH5 files.
- **keep_open** (*bool*) – whether to keep files open by storing the h5py objects as class attributes.

get_buffer(*name*, *lh5_file*, *size=None*, *field_mask=None*)

Returns an LH5 object appropriate for use as a pre-allocated buffer in a read loop. Sets size to *size* if object has a size.

Return type
LGDO

gimme_file(*lh5_file*, *mode='r'*)

Returns a h5py file object from the store or creates a new one.

Parameters

- **lh5_file** (*str | File*) – LH5 file name.
- **mode** (*str*) – mode in which to open file. See h5py.File documentation.

Return type
File

gimme_group(*group*, *base_group*, *grp_attrs=None*, *overwrite=False*)

Returns an existing h5py group from a base group or creates a new one.

See also:

lh5.utils.get_h5_group

Return type
Group

read(*name*, *lh5_file*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *use_h5idx=False*, *field_mask=None*, *obj_buf=None*, *obj_buf_start=0*, *decompress=True*)

Read LH5 object data from a file in the store.

See also:

lh5.core.read

Return type
tuple[LGDO, int]

read_n_rows(*name*, *lh5_file*)

Look up the number of rows in an Array-like object called *name* in *lh5_file*.

Return `None` if it is a *Scalar* or a *Struct*.

Return type
int | None

write(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

Write an LGDO into an LH5 file.

See also:

lh5.core.write

lgdo.lh5.tools module

lgdo.lh5.tools.**load_dfs**(*f_list*, *par_list*, *lh5_group=''*, *idx_list=None*)

Build a `pandas.DataFrame` from LH5 data.

Given a list of files (can use wildcards), a list of LH5 columns, and optionally the group path, return a `pandas.DataFrame` with all values for each parameter.

See also:

load_nda()

Returns
dataframe – contains columns for each parameter in *par_list*, and rows containing all data for the associated parameters concatenated over all files in *f_list*.

Return type
DataFrame

lgdo.lh5.tools.**load_nda**(*f_list*, *par_list*, *lh5_group=''*, *idx_list=None*)

Build a dictionary of `numpy.ndarray`s from LH5 data.

Given a list of files, a list of LH5 table parameters, and an optional group path, return a NumPy array with all values for each parameter.

Parameters

- **f_list** (*str | list[str]*) – A list of files. Can contain wildcards.
- **par_list** (*list[str]*) – A list of parameters to read from each file.

- **lh5_group** (*str*) – group path within which to find the specified parameters.
- **idx_list** (*list[ndarray[Any, dtype[_ScalarType_co]] | list | tuple] | None*) – for fancy-indexed reads. Must be one index array for each file in *f_list*.

Returns
par_data – A dictionary of the parameter data keyed by the elements of *par_list*. Each entry contains the data for the specified parameter concatenated over all files in *f_list*.

Return type
dict[str, *ndarray*[*Any*, *dtype*[*_ScalarType_co*]]]

lgdo.lh5.tools.**ls**(*lh5_file*, *lh5_group=''*, *recursive=False*)

Return a list of LH5 groups in the input file and group, similar to `ls` or `h5ls`. Supports wildcards in group names.

Parameters

- **lh5_file** (*str | Group*) – name of file.
- **lh5_group** (*str*) – group to search. add a `/` to the end of the group name if you want to list all objects inside that group.
- **recursive** (*bool*) – if `True`, recurse into subgroups.

Return type
list[str]

lgdo.lh5.tools.**show**(*lh5_file*, *lh5_group='/'*, *attrs=False*, *indent=''*, *header=True*, *depth=None*, *detail=False*)

Print a tree of LH5 file contents with LGDO datatype.

Parameters

- **lh5_file** (*str | Group*) – the LH5 file.
- **lh5_group** (*str*) – print only contents of this HDF5 group.
- **attrs** (*bool*) – print the HDF5 attributes too.
- **indent** (*str*) – indent the diagram with this string.
- **header** (*bool*) – print *lh5_group* at the top of the diagram.
- **depth** (*int | None*) – maximum tree depth of groups to print
- **detail** (*bool*) – whether to print additional information about how the data is stored

Examples

```
>>> from lgdo import show
>>> show("file.lh5", "/geds/raw")
/geds/raw
├── channel · array<1>{real}
├── energy · array<1>{real}
├── timestamp · array<1>{real}
├── waveform · table{t0,dt,values}
│   ├── dt · array<1>{real}
│   ├── t0 · array<1>{real}
│   └── values · array_of_equalsized_arrays<1,1>{real}
└── wf_std · array<1>{real}
```

lgdo.lh5.utils module

Implements utilities for LEGEND Data Objects.

lgdo.lh5.utils.**expand_path**(*path*, *substitute=None*, *list=False*, *base_path=None*)

Expand (environment) variables and wildcards to return absolute paths.

Parameters

- **path** (*str*) – name of path, which may include environment variables and wildcards.
- **list** (*bool*) – if `True`, return a list. If `False`, return a string; if `False` and a unique file is not found, raise an exception.
- **substitute** (*dict[str, str] | None*) – use this dictionary to substitute variables. Environment variables take precedence.
- **base_path** (*str | None*) – name of base path. Returned paths will be relative to base.

Returns

path or list of paths – Unique absolute path, or list of all absolute paths

Return type

str | list

lgdo.lh5.utils.**expand_vars**(*expr*, *substitute=None*)

Expand (environment) variables.

Note: Malformed variable names and references to non-existing variables are left unchanged.

Parameters

- **expr** (*str*) – string expression, which may include (environment) variables prefixed by `$`.
- **substitute** (*dict[str, str] | None*) – use this dictionary to substitute variables. Takes precedence over environment variables.

Return type

str

lgdo.lh5.utils.**fmtbytes**(*num*, *suffix='B'*)

Returns formatted f-string for printing human-readable number of bytes.

lgdo.lh5.utils.**get_buffer**(*name*, *lh5_file*, *size=None*, *field_mask=None*)

Returns an LGDO appropriate for use as a pre-allocated buffer.

Sets size to *size* if object has a size.

Return type

LGDO

lgdo.lh5.utils.**get_h5_group**(*group*, *base_group*, *grp_attrs=None*, *overwrite=False*)

Returns an existing `h5py` group from a base group or creates a new one. Can also set (or replace) group attributes.

Parameters

- **group** (*str | Group*) – name of the HDF5 group.
- **base_group** (*Group*) – HDF5 group to be used as a base.
- **grp_attrs** (*Mapping[str, Any] | None*) – HDF5 group attributes.

- **overwrite** (*bool*) – whether overwrite group attributes, ignored if *grp_attrs* is None.

Return type
Group

lgdo.lh5.utils.**read_n_rows**(*name*, *h5f*)

Look up the number of rows in an Array-like LGDO object on disk.

Return None if *name* is a *Scalar* or a *Struct*.

Return type
int | None

lgdo.types package

LEGEND Data Objects (LGDO) types.

Submodules

lgdo.types.array module

Implements a LEGEND Data Object representing an n-dimensional array and corresponding utilities.

class lgdo.types.array.**Array**(*nda=None*, *shape=()*, *dtype=None*, *fill_val=None*, *attrs=None*)

Bases: *LGDO*

Holds an numpy.ndarray and attributes.

Array (and the other various array types) holds an *nda* instead of deriving from numpy.ndarray for the following reasons:

- It keeps management of the *nda* totally under the control of the user. The user can point it to another object's buffer, grab the *nda* and toss the *Array*, etc.
- It allows the management code to send just the *nda*'s the central routines for data manpulation. Keeping LGDO's out of that code allows for more standard, reusable, and (we expect) performant Python.
- It allows the first axis of the *nda* to be treated as "special" for storage in *Table*s.

Parameters

- **nda** (*np.ndarray*) – An numpy.ndarray to be used for this object's internal array. Note: the array is used directly, not copied. If not supplied, internal memory is newly allocated based on the shape and dtype arguments.
- **shape** (*tuple[int, ...]*) – A numpy-format shape specification for shape of the internal ndarray. Required if *nda* is None, otherwise unused.
- **dtype** (*np.dtype*) – Specifies the type of the data in the array. Required if *nda* is None, otherwise unused.
- **fill_val** (*float | int | None*) – If None, memory is allocated without initialization. Otherwise, the array is allocated with all elements set to the corresponding fill value. If *nda* is not None, this parameter is ignored.
- **attrs** (*dict[str, Any] | None*) – A set of user attributes to be carried along with this LGDO.

_abc_impl = <_abc._abc_data object>

append(*value*)

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

insert(*i*, *value*)

resize(*new_size*)

view_as(*library*, *with_units=False*)

View the Array data as a third-party format data structure.

This is a zero-copy operation. Supported third-party formats are:

- pd: returns a `pandas.Series`
- np: returns the internal *nda* attribute (`numpy.ndarray`)
- ak: returns an `ak.Array` initialized with *self.nda*

Parameters

- **library** (*str*) – format of the returned data view.
- **with_units** (*bool*) – forward physical units to the output data.

Return type
pd.DataFrame | np.NDArray | ak.Array

See also:

LGDO.view_as

lgdo.types.arrayofequalsizedarrays module

Implements a LEGEND Data Object representing an array of equal-sized arrays and corresponding utilities.

class lgdo.types.arrayofequalsizedarrays.**ArrayOfEqualSizedArrays**(*dims=None*, *nda=None*, *shape=()*, *dtype=None*, *fill_val=None*, *attrs=None*)

Bases: *Array*

An array of equal-sized arrays.

Arrays of equal size within a file but could be different from application to application. Canonical example: array of same-length waveforms.

Parameters

- **dims** (*tuple[int, ...] | None*) – specifies the dimensions required for building the *ArrayOfEqualSizedArrays*' *datatype* attribute.
- **nda** (*np.ndarray*) – An numpy.ndarray to be used for this object's internal array. Note: the array is used directly, not copied. If not supplied, internal memory is newly allocated based on the *shape* and *dtype* arguments.
- **shape** (*tuple[int, ...]*) – A NumPy-format shape specification for shape of the internal array. Required if *nda* is None, otherwise unused.
- **dtype** (*np.dtype*) – Specifies the type of the data in the array. Required if *nda* is None, otherwise unused.
- **fill_val** (*int | float | None*) – If None, memory is allocated without initialization. Otherwise, the array is allocated with all elements set to the corresponding fill value. If *nda* is not None, this parameter is ignored.
- **attrs** (*dict[str, Any] | None*) – A set of user attributes to be carried along with this LGDO.

Notes

If shape is not "1D array of arrays of shape given by axes 1-N" (of *nda*) then specify the dimensionality split in the constructor.

See also:

Array

_abc_impl = <_abc._abc_data object>

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

to_vov(*cumulative_length=None*)

Convert (and eventually resize) to *vectorofvectors.VectorOfVectors*.

Parameters
cumulative_length (*ndarray | None*) – cumulative length array of the output vector of vectors. Each vector in the output is filled with values found in the *ArrayOfEqualSizedArrays*, starting from the first index. if None, use all of the original 2D array and make vectors of equal size.

Return type
VectorOfVectors

view_as(*library*, *with_units=False*)

View the array as a third-party format data structure.

See also:

LGDO.view_as

Return type
pd.DataFrame | np.NDArray | ak.Array

lgdo.types.encoded module

class lgdo.types.encoded.**ArrayOfEncodedEqualSizedArrays**(*encoded_data=None*, *decoded_size=None*, *attrs=None*)

Bases: *LGDO*

An array of encoded arrays with equal decoded size.

Used to represent an encoded *ArrayOfEqualSizedArrays*. In addition to an internal *VectorOfVectors* *self.encoded_data* storing the encoded data, the size of the decoded arrays is stored in a *Scalar* *self.encoded_size*.

See also:

ArrayOfEqualSizedArrays

Parameters

- **encoded_data** (*VectorOfVectors*) – the vector of vectors holding the encoded data.
- **decoded_size** (*Scalar | int*) – the length of the decoded arrays.
- **attrs** (*dict[str, Any] | None*) – A set of user attributes to be carried along with this LGDO. Should include information about the codec used to encode the data.

_abc_impl = <_abc._abc_data object>

append(*value*)

Append a 1D encoded array at the end.

See also:

VectorOfVectors.append

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

insert(*i*, *value*)

Insert an encoded array at index *i*.

See also:

VectorOfVectors.insert

replace(*i*, *value*)

Replace the encoded array at index *i* with a new one.

See also:

VectorOfVectors.replace

resize(*new_size*)

Resize array along the first axis.

See also:

VectorOfVectors.resize

view_as(*library*, *with_units=False*)

View the encoded data as a third-party format data structure.

This is nearly a zero-copy operation.

Supported third-party formats are:

- pd: returns a `pandas.DataFrame`
- ak: returns an `ak.Array` (record type)

Note: In the view, *decoded_size* is expanded into an array.

Parameters

- **library** (*str*) – format of the returned data view.
- **with_units** (*bool*) – forward physical units to the output data.

Return type
pd.DataFrame | np.NDArray | ak.Array

See also:

LGDO.view_as

class lgdo.types.encoded.**VectorOfEncodedVectors**(*encoded_data=None*, *decoded_size=None*, *attrs=None*)

Bases: *LGDO*

An array of variable-length encoded arrays.

Used to represent an encoded *VectorOfVectors*. In addition to an internal *VectorOfVectors* *self.encoded_data* storing the encoded data, a 1D *Array* in *self.encoded_size* holds the original sizes of the encoded vectors.

See also:

VectorOfVectors

Parameters

- **encoded_data** (VectorOfVectors) – the vector of encoded vectors.
- **decoded_size** (Array) – an array holding the original length of each encoded vector in *encoded_data*.
- **attrs** (*dict[str, Any] | None*) – A set of user attributes to be carried along with this LGDO. Should include information about the codec used to encode the data.

_abc_impl = <_abc._abc_data object>

append(*value*)

Append a 1D encoded vector at the end.

Parameters

value (*tuple[ndarray[Any, dtype[_ScalarType_co]], int]*) – a tuple holding the encoded array and its decoded size.

See also:

VectorOfVectors.append

datatype_name()

The name for this LGDO's datatype attribute.

Return type

str

form_datatype()

Return this LGDO's datatype attribute string.

Return type

str

insert(*i*, *value*)

Insert an encoded vector at index *i*.

Parameters

- **i** (*int*) – the new vector will be inserted before this index.
- **value** (*tuple[ndarray[Any, dtype[_ScalarType_co]], int]*) – a tuple holding the encoded array and its decoded size.

See also:

VectorOfVectors.insert

replace(*i*, *value*)

Replace the encoded vector (and decoded size) at index *i* with a new one.

Parameters

- **i** (*int*) – index of the vector to be replaced.

- **value** (*tuple[ndarray[Any, dtype[_ScalarType_co]], int]*) – a tuple holding the encoded array and its decoded size.

See also:

VectorOfVectors.replace

resize(*new_size*)

Resize vector along the first axis.

See also:

VectorOfVectors.resize

view_as(*library*, *with_units=False*)

View the encoded data as a third-party format data structure.

This is a zero-copy or nearly zero-copy operation.

Supported third-party formats are:

- pd: returns a pandas.DataFrame
- ak: returns an ak.Array (record type)

Parameters

- **library** (*str*) – format of the returned data view.
- **with_units** (*bool*) – forward physical units to the output data.

Return type
pd.DataFrame | np.NDArray | ak.Array

See also:

LGDO.view_as

lgdo.types.fixedsizearray module

Implements a LEGEND Data Object representing an n-dimensional array of fixed size and corresponding utilities.

class lgdo.types.fixedsizearray.**FixedSizeArray**(*nda=None*, *shape=()*, *dtype=None*, *fill_val=None*, *attrs=None*)

Bases: *Array*

An array of fixed-size arrays.

Arrays with guaranteed shape along axes > 0: for example, an array of vectors will always length 3 on axis 1, and it will never change from application to application. This data type is used for optimized memory handling on some platforms. We are not that sophisticated so we are just storing this identification for LGDO validity, i.e. for now this class is just an alias for *Array*, but keeps track of the datatype name.

See also:

Array

_abc_impl = <_abc._abc_data object>

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

view_as(*library*, *with_units=False*)

View the array as a third-party format data structure.

See also:

LGDO.view_as

lgdo.types.lgdo module

class lgdo.types.lgdo.LGDO(*attrs=None*)

Bases: ABC

Abstract base class representing a LEGEND Data Object (LGDO).

_abc_impl = <_abc._abc_data object>

abstract datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

abstract form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

getattrs(*datatype=False*)

Return a copy of the LGDO attributes dictionary.

Parameters
datatype (*bool*) – if `False`, remove `datatype` attribute from the output dictionary.

Return type
dict

abstract view_as(*library*, *with_units=False*)

View the LGDO data object as a third-party format data structure.

This is typically a zero-copy or nearly zero-copy operation unless explicitly stated in the concrete LGDO documentation. The view can be turned into a copy explicitly by the user with the appropriate methods. If requested by the user, the output format supports it and the LGDO carries a `units` attribute, physical units are attached to the view through the `pint` package.

Typical supported third-party libraries are:

- pd: pandas

- np: numpy
- ak: awkward

Note: Awkward does not support attaching units through Pint, at the moment.

but the actual supported formats may vary depending on the concrete LGDO class.

Parameters

- **library** (*str*) – format of the returned data view.
- **with_units** (*bool*) – forward physical units to the output data.

Return type
pd.DataFrame | np.NDArray | ak.Array

lgdo.types.scalar module

Implements a LEGEND Data Object representing a scalar and corresponding utilities.

class lgdo.types.scalar.**Scalar**(*value*, *attrs=None*)

Bases: *LGDO*

Holds just a scalar value and some attributes (datatype, units, ...).

Parameters

- **value** (*int | float | str*) – the value for this scalar.
- **attrs** (*dict[str, Any] | None*) – a set of user attributes to be carried along with this LGDO.

_abc_impl = <_abc._abc_data object>

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

view_as(*with_units=False*)

Dummy function, returns the scalar value itself.

See also:

LGDO.view_as

lgdo.types.struct module

Implements a LEGEND Data Object representing a struct and corresponding utilities.

class lgdo.types.struct.**Struct**(*obj_dict=None*, *attrs=None*)

Bases: *LGDO*, dict

A dictionary of LGDO's with an optional set of attributes.

After instantiation, add fields using *add_field()* to keep the datatype updated, or call *update_datatype()* after adding.

Parameters

- **obj_dict** (*Mapping[str, LGDO] | None*) – instantiate this Struct using the supplied named LGDO's. Note: no copy is performed, the objects are used directly.
- **attrs** (*Mapping[str, Any] | None*) – a set of user attributes to be carried along with this LGDO.

_abc_impl = <_abc._abc_data object>

add_field(*name*, *obj*)

Add a field to the table.

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

remove_field(*name*, *delete=False*)

Remove a field from the table.

Parameters

- **name** (*str | int*) – name of the field to be removed.
- **delete** (*bool*) – if True, delete the field object by calling The del statement.

update_datatype()

view_as()

View the Struct data as a third-party format data structure.

> **Error:** Not implemented. Since Struct's fields can have different lengths, converting to a NumPy, Pandas or Awkward is generally not possible. Call *LGDO.view_as()* on the fields instead.

See also:

LGDO.view_as

lgdo.types.table module

Implements a LEGEND Data Object representing a special struct of arrays of equal length and corresponding utilities.

class lgdo.types.table.**Table**(*col_dict=None*, *size=None*, *attrs=None*)

Bases: *Struct*

A special struct of arrays or subtable columns of equal length.

Holds onto an internal read/write location `loc` that is useful in managing table I/O using functions like *push_row()*, *is_full()*, and *clear()*.

Note: If you write to a table and don't fill it up to its total size, be sure to resize it before passing to data processing functions, as they will call `__len__()` to access valid data, which returns the `size` attribute.

Parameters

- **size** (*int | None*) – sets the number of rows in the table. *Array*s in *col_dict will be resized to match size if both are not ``None``*. If *size* is left as `None`, the number of table rows is determined from the length of the first array in *col_dict*. If neither is provided, a default length of 1024 is used.
- **col_dict** (*Mapping[str, LGDO] | pd.DataFrame | ak.Array | None*) – instantiate this table using the supplied mapping of column names and array-like objects. Supported input types are: mapping of strings to LGDOs, `pd.DataFrame` and `ak.Array`. Note 1: no copy is performed, the objects are used directly (unless `ak.Array` is provided). Note 2: if *size* is not `None`, all arrays will be resized to match it. Note 3: if the arrays have different lengths, all will be resized to match the length of the first array.
- **attrs** (*Mapping[str, Any] | None*) – A set of user attributes to be carried along with this LGDO.

Notes

the `loc` attribute is initialized to 0.

_abc_impl = <_abc._abc_data object>

add_column(*name*, *obj*, *use_obj_size=False*)

Alias for *add_field()* using table terminology 'column'.

add_field(*name*, *obj*, *use_obj_size=False*)

Add a field (column) to the table.

Use the name "field" here to match the terminology used in *Struct*.

Parameters

- **name** (*str*) – the name for the field in the table.
- **obj** (LGDO) – the object to be added to the table.

- **use_obj_size** (*bool*) – if `True`, resize the table to match the length of *obj*.

clear() → None. Remove all items from D.

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

eval(*expr*, *parameters=None*)

Apply column operations to the table and return a new LGDO.

Internally uses `numexpr.evaluate()` if dealing with columns representable as NumPy arrays or *eval()* if *VectorOfVectors* are involved. In the latter case, the VoV columns are viewed as `ak.Array` and the respective routines are therefore available.

To columns nested in subtables can be accessed by scoping with two underscores (__). For example:

```
tbl.eval("a + tbl2__b")
```

computes the sum of column *a* and column *b* in the subtable *tbl2*.

Parameters

- **expr** (*str*) – if the expression only involves non-*VectorOfVectors* columns, the syntax is the one supported by `numexpr.evaluate()` (see here for documentation). Note: because of internal limitations, reduction operations must appear the last in the stack. If at least one considered column is a *VectorOfVectors*, plain *eval()* is used and `ak.Array` transforms can be used through the `ak.` prefix. (NumPy functions are analogously accessible through `np.`). See also examples below.
- **parameters** (*Mapping[str, str] | None*) – a dictionary of function parameters. Passed to `numexpr.evaluate`() as *local_dict* argument or to *eval()* as *locals* argument.

Return type
LGDO

Examples

```
>>> import lgdo
>>> tbl = lgdo.Table(
...   col_dict={
...     "a": lgdo.Array([1, 2, 3]),
...     "b": lgdo.VectorOfVectors([[5], [6, 7], [8, 9, 0]]),
...   }
... )
>>> print(tbl.eval("a + b"))
[[6],
 [8 9],
 [11 12  3],
]
>>> print(tbl.eval("np.sum(a) + ak.sum(b)"))
41
```

flatten(*_prefix=''*)

Flatten the table, if nested.

Returns a new *Table* (that references, not copies, the existing columns) with columns in nested tables being moved to the first level (and renamed appropriately).

Examples

```
>>> repr(tbl)
"Table(dict={'a': Array([1 2 3], attrs={'datatype': 'array<1>{real}'}), 'tbl':␣
↪Table(dict={'b': Array([4 5 6], attrs={'datatype': 'array<1>{real}'}), 'tbl1':
↪ Table(dict={'z': Array([9 9 9], attrs={'datatype': 'array<1>{real}'})},␣
↪attrs={'datatype': 'table{z}'})}, attrs={'datatype': 'table{b,tbl1}'})},␣
↪attrs={'datatype': 'table{a,tbl}'})"
>>> tbl.flatten().keys()
dict_keys(['a', 'tbl__b', 'tbl__tbl1__z'])
```

Return type
Table

get_dataframe(*cols=None*, *copy=False*, *prefix=''*)

Get a `pandas.DataFrame` from the data in the table.

Warning: This method is deprecated. Use *view_as()* to view the table as a Pandas dataframe.

Notes

The requested data must be array-like, with the `nda` attribute.

Parameters

- **cols** (*list[str] | None*) – a list of column names specifying the subset of the table's columns to be added to the dataframe.
- **copy** (*bool*) – When `True`, the dataframe allocates new memory and copies data into it. Otherwise, the raw `nda`'s from the table are used directly.
- **prefix** (*str*) – The prefix to be added to the column names. Used when recursively getting the dataframe of a Table inside this Table

Return type
DataFrame

is_full()

Return type
bool

join(*other_table*, *cols=None*, *do_warn=True*)

Add the columns of another table to this table.

Notes

Following the join, both tables have access to *other_table*'s fields (but *other_table* doesn't have access to this table's fields). No memory is allocated in this process. *other_table* can go out of scope and this table will retain access to the joined data.

Parameters

- **other_table** (Table) – the table whose columns are to be joined into this table.
- **cols** (*list[str] | None*) – a list of names of columns from *other_table* to be joined into this table.
- **do_warn** (*bool*) – set to `False` to turn off warnings associated with mismatched *loc* parameter or *add_column()* warnings.

push_row()

remove_column(*name*, *delete=False*)

Alias for *remove_field()* using table terminology 'column'.

resize(*new_size=None*, *do_warn=False*)

view_as(*library*, *with_units=False*, *cols=None*, *prefix=''*)

View the Table data as a third-party format data structure.

This is typically a zero-copy or nearly zero-copy operation.

Supported third-party formats are:

- pd: returns a pandas.DataFrame
- ak: returns an ak.Array (record type)

Notes

Conversion to Awkward array only works when the key is a string.

Parameters

- **library** (*str*) – format of the returned data view.
- **with_units** (*bool*) – forward physical units to the output data.
- **cols** (*list[str] | None*) – a list of column names specifying the subset of the table's columns to be added to the data view structure.
- **prefix** (*str*) – The prefix to be added to the column names. Used when recursively getting the dataframe of a *Table* inside this *Table*.

Return type
pd.DataFrame | np.NDArray | ak.Array

See also:

LGDO.view_as

lgdo.types.table.**_ak_to_lgdo_or_col_dict**(*array*)

lgdo.types.vectorofvectors module

Implements a LEGEND Data Object representing a variable-length array of variable-length arrays and corresponding utilities.

class lgdo.types.vectorofvectors.**VectorOfVectors**(*data=None*, *flattened_data=None*, *cumulative_length=None*, *shape_guess=None*, *dtype=None*, *fill_val=None*, *attrs=None*)

Bases: *LGDO*

A n-dimensional variable-length 1D array of variable-length 1D arrays.

If the vector is 2-dimensional, the internal representation is as two NumPy arrays, one to store the flattened data contiguosly (`flattened_data`) and one to store the cumulative sum of lengths of each vector (`cumulative_length`). When the dimension is more than 2, `flattened_data` is a *VectorOfVectors* itself.

Examples

```
>>> from lgdo import VectorOfVectors
>>> data = VectorOfVectors(
...   [[[1, 2], [3, 4, 5]], [[2], [4, 8, 9, 7]], [[5, 3, 1]]],
...   attrs={"units": "m"}
... )
>>> print(data)
[[[1, 2], [3, 4, 5]],
 [[2], [4, 8, 9, 7]],
 [[5, 3, 1]]
] with attrs={'units': 'm'}
>>> data.view_as("ak")
<Array [[[1, 2], [3, 4, 5]], ..., [[5, ..., 1]]] type='3 * var * var * int64'>
```

Note: Many class methods are currently implemented only for 2D vectors and will raise an exception on higher dimensional data.

Parameters

- **data** (*ArrayLike | None*) – Any array-like structure accepted by the `ak.Array` constructor, with the exception that elements cannot be of type `OptionType`, `UnionType` or `RecordType`. Takes priority over *flattened_data* and *cumulative_length*. The serialization of the `ak.Array` is performed through `ak.to_buffers()`. Since the latter returns non-data-owning NumPy arrays, which would prevent later modifications like resizing, a copy is performed.
- **flattened_data** (*ArrayLike | None*) – if not `None`, used as the internal array for *self.flattened_data*. Otherwise, an internal *flattened_data* is allocated based on *cumulative_length* (or *shape_guess*) and *dtype*.

- **cumulative_length** (*ArrayLike | VectorOfVectors | None*) – if not None, used as the internal array for *self.cumulative_length*. Should be *dtype* numpy.uint32. If *cumulative_length* is None, an internal *cumulative_length* is allocated based on the first element of *shape_guess*.
- **shape_guess** (*Sequence[int, ...] | None*) – a NumPy-format shape specification, required if either of *flattened_data* or *cumulative_length* are not supplied. The first element should not be a guess and sets the number of vectors to be stored. The second element is a guess or approximation of the typical length of a stored vector, used to set the initial length of *flattened_data* if it was not supplied.
- **dtype** (*DTypeLike | None*) – sets the type of data stored in *flattened_data*. Required if *flattened_data* and *array* are None.
- **fill_val** (*int | float | None*) – fill all of *self.flattened_data* with this value.
- **attrs** (*Mapping[str, Any] | None*) – a set of user attributes to be carried along with this LGDO.

_abc_impl = <_abc._abc_data object>

_set_vector_unsafe(*i*, *vec*, *lens=None*)

Insert vector *vec* at position *i*.

Assumes that `j = self.cumulative_length[i-1]` is the index (in *self.flattened_data*) of the end of the *(i-1)*th vector and copies *vec* in `self.flattened_data[j:sum(lens)]`. Finally updates `self.cumulative_length[i]` with the new flattened data array length.

Vectors stored after index *i* can be overridden, producing unintended behavior. This method is typically used for fast sequential fill of a pre-allocated vector of vectors.

If i`vec` is 1D array and *lens* is None, set using full array. If *vec* is 2D, require *lens* to be included, and fill each array only up to lengths in *lens*.

> **Danger:** This method can lead to undefined behavior or vector invalidation if used improperly. Use it only if you know what you are doing.

See also:

append, *replace*, *insert*

append(*new*)

Append a 1D vector *new* at the end.

Examples

```
>>> vov = VectorOfVectors([[1, 2, 3], [4, 5]])
>>> vov.append([8, 9])
>>> print(vov)
[[1 2 3],
 [4 5],
 [8 9],
]
```

datatype_name()

The name for this LGDO's datatype attribute.

Return type
str

form_datatype()

Return this LGDO's datatype attribute string.

Return type
str

insert(*i*, *new*)

Insert a vector at index *i*.

self.flattened_data (and therefore *self.cumulative_length*) is resized in order to accommodate the new element.

Examples

```
>>> vov = VectorOfVectors([[1, 2, 3], [4, 5]])
>>> vov.insert(1, [8, 9])
>>> print(vov)
[[1 2 3],
 [8 9],
 [4 5],
]
```

Warning: This method involves a significant amount of memory re-allocation and is expected to perform poorly on large vectors.

replace(*i*, *new*)

Replace the vector at index *i* with *new*.

self.flattened_data (and therefore *self.cumulative_length*) is resized, if the length of *new* is different from the vector currently at index *i*.

Examples

```
>>> vov = VectorOfVectors([[1, 2, 3], [4, 5]])
>>> vov.replace(0, [8, 9])
>>> print(vov)
[[8 9],
 [4 5],
]
```

Warning: This method involves a significant amount of memory re-allocation and is expected to perform poorly on large vectors.

resize(*new_size*)

Resize vector along the first axis.

self.flattened_data is resized only if *new_size* is smaller than the current vector length.

If *new_size* is larger than the current vector length, *self.cumulative_length* is padded with its last element. This corresponds to appending empty vectors.

Examples

```
>>> vov = VectorOfVectors([[1, 2, 3], [4, 5]])
>>> vov.resize(3)
>>> print(vov)
[[1 2 3],
 [4 5],
 [],
]
```

```
>>> vov = VectorOfVectors([[1, 2], [3], [4, 5]])
>>> vov.resize(2)
>>> print(vov)
[[1 2],
 [3],
]
```

to_aoesa(*max_len=None*, *fill_val=nan*, *preserve_dtype=False*)

Convert to `ArrayOfEqualSizedArrays`.

Note: The dtype of the original vector is typically not strictly preserved. The output dtype will be either `np.float64` or `np.int64`. If you want to use the same exact dtype, set *preserve_dtype* to `True`.

Parameters

- **max_len** (*int | None*) – the length of the returned array along its second dimension. Longer vectors will be truncated, shorter will be padded with *fill_val*. If None, the length will be equal to the length of the longest vector.
- **fill_val** (*bool | int | float*) – value used to pad shorter vectors up to *max_len*. The dtype of the output array will be such that both *fill_val* and the vector values can be represented in the same data structure.
- **preserve_dtype** (*bool*) – whether the output array should have exactly the same dtype as the original vector of vectors. The type *fill_val* must be a compatible one.

Return type
ArrayOfEqualSizedArrays

view_as(*library*, *with_units=False*, *fill_val=nan*, *preserve_dtype=False*)

View the vector data as a third-party format data structure.

This is typically a zero-copy or nearly zero-copy operation.

Supported third-party formats are:

- pd: returns a pandas.Series (supported through the awkward-pandas package)
- np: returns a numpy.ndarray, padded with zeros to make it rectangular. This implies memory re-allocation.
- ak: returns an ak.Array. self.cumulative_length is currently re-allocated for technical reasons.

Notes

Awkward array views partially involve memory re-allocation (the *cumulative_length*s), while NumPy "exploded" views clearly imply a full copy.

Parameters

- **library** (*str*) – format of the returned data view.
- **with_units** (*bool*) – forward physical units to the output data.
- **fill_val** (*bool | int | float*) – forwarded to *to_aoesa()*, if *library* is np.
- **preserve_dtype** (*bool*) – forwarded to *to_aoesa()*, if *library* is np.

Return type
pd.DataFrame | np.NDArray | ak.Array

See also:

LGDO.view_as

lgdo.types.vovutils module

VectorOfVectors utilities.

lgdo.types.vovutils.**_ak_is_jagged**(*type_*)

Returns True if ak.Array is jagged at all axes.

This assures that ak.to_buffers() returns the expected data structures.

Return type
bool

lgdo.types.vovutils.**_ak_is_valid**(*type_*)

Returns `True` if `ak.Array` contains only elements we can serialize to LH5.

Return type
bool

@numba.jit lgdo.types.vovutils.**_nb_build_cl**(*sorted_array_in*, *cumulative_length_out*)

numbified inner loop for build_cl

Return type
ndarray[*Any*, *dtype*[*_ScalarType_co*]]

@numba.jit lgdo.types.vovutils.**_nb_explode**(*cumulative_length*, *array_in*, *array_out*)

Numbified inner loop for *explode()*.

Return type
ndarray[*Any*, *dtype*[*_ScalarType_co*]]

@numba.jit lgdo.types.vovutils.**_nb_explode_cl**(*cumulative_length*, *array_out*)

numbified inner loop for explode_cl

Return type
ndarray[*Any*, *dtype*[*_ScalarType_co*]]

@numba.guvectorize lgdo.types.vovutils.**_nb_fill**(*aoa_in*, *len_in*, *flattened_array_out*)

- *Options:* `boundscheck=False`, `cache=True`
- *Precompiled signatures:* `?i?->`, `?l?->`, `?I?->`, `?L?->`, `bib->`, `blb->`, `bIb->`, `bLb->`, `hih->`, `hlh->`, `hIh->`, `hLh->`, `iii->`, `ili->`, `iIi->`, `iLi->`, `lil->`, `lll->`, `lIl->`, `lLl->`, `BiB->`, `BlB->`, `BIB->`, `BLB->`, `HiH->`, `HlH->`, `HIH->`, `HLH->`, `IiI->`, `IlI->`, `III->`, `ILI->`, `LiL->`, `LlL->`, `LIL->`, `LLL->`, `fif->`, `flf->`, `fIf->`, `fLf->`, `did->`, `dld->`, `dId->`, `dLd->`, `FiF->`, `FlF->`, `FIF->`, `FLF->`, `DiD->`, `DlD->`, `DID->`, `DLD->`

Vectorized function to fill flattened array from array of arrays and lengths. Values in aoa_in past lengths will not be copied.

Parameters

- **aoa_in** (*ndarray[Any, dtype[_ScalarType_co]]*) – array of arrays containing values to be copied
- **len_in** (*ndarray[Any, dtype[_ScalarType_co]]*) – array of vector lengths for each row of aoa_in
- **flattened_array_out** (*ndarray[Any, dtype[_ScalarType_co]]*) – flattened array to copy values into. Must be longer than sum of lengths in len_in

lgdo.types.vovutils.**build_cl**(*sorted_array_in*, *cumulative_length_out=None*)

Build a cumulative length array from an array of sorted data.

Examples

```
>>> build_cl(np.array([3, 3, 3, 4])
array([3., 4.])
```

For a *sorted_array_in* of indices, this is the inverse of *explode_cl()*, in the sense that doing `build_cl(explode_cl(cumulative_length))` would recover the original *cumulative_length*.

Parameters

- **sorted_array_in** (*ndarray[Any, dtype[_ScalarType_co]]*) – array of data already sorted; each N matching contiguous entries will be converted into a new row of *cumulative_length_out*.
- **cumulative_length_out** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – a pre-allocated array for the output *cumulative_length*. It will always have length <= *sorted_array_in*, so giving them the same length is safe if there is not a better guess.

Returns

cumulative_length_out – the output cumulative length array. If the user provides a *cumulative_length_out* that is too long, this return value is sliced to contain only the used portion of the allocated memory.

Return type

ndarray[*Any*, *dtype*[*_ScalarType_co*]]

lgdo.types.vovutils.**explode**(*cumulative_length*, *array_in*, *array_out=None*)

Explode a data array using a *cumulative_length* array.

This is identical to *explode_cl()*, except *array_in* gets exploded instead of *cumulative_length*.

Examples

```
>>> explode(np.array([2, 3]), np.array([3, 4]))
array([3., 3., 4.])
```

Parameters

- **cumulative_length** (*ndarray[Any, dtype[_ScalarType_co]]*) – the cumulative length array to use for exploding.
- **array_in** (*ndarray[Any, dtype[_ScalarType_co]]*) – the data to be exploded. Must have same length as *cumulative_length*.
- **array_out** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – a pre-allocated array to hold the exploded data. The length should be equal to `cumulative_length[-1]`.

Returns

array_out – the exploded cumulative length array.

Return type

ndarray[*Any*, *dtype*[*_ScalarType_co*]]

lgdo.types.vovutils.**explode_arrays**(*cumulative_length*, *arrays*, *arrays_out=None*)

Explode a set of arrays using a *cumulative_length* array.

Parameters

- **cumulative_length** (*Array*) – the cumulative length array to use for exploding.

- **arrays** (*Sequence[ndarray[Any, dtype[_ScalarType_co]]]*) – the data arrays to be exploded. Each array must have same length as *cumulative_length*.
- **arrays_out** (*Sequence[ndarray[Any, dtype[_ScalarType_co]]] | None*) – a list of pre-allocated arrays to hold the exploded data. The length of the list should be equal to the length of *arrays*, and each entry in arrays_out should have length `cumulative_length[-1]`. If not provided, output arrays are allocated for the user.

Returns
arrays_out – the list of exploded cumulative length arrays.

Return type
list

`lgdo.types.vovutils.`**`explode_cl`**(*cumulative_length*, *array_out=None*)

Explode a *cumulative_length* array.

Examples

```
>>> explode_cl(np.array([2, 3]))
array([0., 0., 1.])
```

This is the inverse of *build_cl()*, in the sense that doing `build_cl(explode_cl(cumulative_length))` would recover the original *cumulative_length*.

Parameters

- **cumulative_length** (*ndarray[Any, dtype[_ScalarType_co]]*) – the cumulative length array to be exploded.
- **array_out** (*ndarray[Any, dtype[_ScalarType_co]] | None*) – a pre-allocated array to hold the exploded cumulative length array. The length should be equal to `cumulative_length[-1]`.

Returns
array_out – the exploded cumulative length array.

Return type
ndarray[*Any*, *dtype*[*_ScalarType_co*]]

lgdo.types.waveformtable module

Implements a LEGEND Data Object representing a special `Table` to store blocks of one-dimensional time-series data.

class `lgdo.types.waveformtable.`**`WaveformTable`**(*size=None*, *t0=0*, *t0_units=None*, *dt=1*, *dt_units=None*, *values=None*, *values_units=None*, *wf_len=None*, *dtype=None*, *attrs=None*)

Bases: *Table*

An LGDO for storing blocks of (1D) time-series data.

A *WaveformTable* is an LGDO *Table* with the 3 columns `t0`, `dt`, and `values`:

- `t0[i]` is a time offset (relative to a user-defined global reference) for the sample in `values[i][0]`. Implemented as an LGDO *Array* with optional attribute `units`.

- `dt[i]` is the sampling period for the waveform at `values[i]`. Implemented as an LGDO *Array* with optional attribute `units`.
- `values[i]` is the `i`'th waveform in the table. Internally, the waveforms values may be either an LGDO *ArrayOfEqualSizedArrays*<1,1>, an LGDO *VectorOfVectors* or *VectorOfEncodedVectors* that supports waveforms of unequal length. Can optionally be given a `units` attribute.

Note: On-disk and in-memory versions could be different e.g. if a compression routine is used.

Parameters

- **size** (*int | None*) – sets the number of rows in the table. If `None`, the size will be determined from the first among *t0*, *dt*, or *values* to return a valid length. If not `None`, *t0*, *dt*, and values will be resized as necessary to match *size*. If *size* is `None` and *t0*, *dt*, and *values* are all non-array-like, a default size of 1024 is used.
- **t0** (*float | Array | np.ndarray*) – t_0 values to be used (or broadcast) to the *t0* column.
- **t0_units** (*str | None*) – units for the t_0 values. If not `None` and *t0* is an LGDO *Array*, overrides what's in *t0*.
- **dt** (*float | Array | np.ndarray*) – δt values (sampling period) to be used (or broadcasted) to the *t0* column.
- **dt_units** (*str | None*) – units for the *dt* values. If not `None` and *dt* is an LGDO *Array*, overrides what's in *dt*.
- **values** (*ArrayOfEqualSizedArrays | VectorOfVectors | np.ndarray*) – The waveform data to be stored in the table. If `None` a block of data is prepared based on the *wf_len* and *dtype* arguments.
- **values_units** (*str | None*) – units for the waveform values. If not `None` and *values* is an LGDO *Array*, overrides what's in *values*.
- **wf_len** (*int | None*) – The length of the waveforms in each entry of a table. If `None` (the default), unequal lengths are assumed and *VectorOfVectors* is used for the *values* column. Ignored if *values* is a 2D ndarray, in which case `values.shape[1]` is used.
- **dtype** (*np.dtype*) – The NumPy `numpy.dtype` of the waveform data. If *values* is not `None`, this argument is ignored. If both *values* and *dtype* are `None`, `numpy.float64` is used.
- **attrs** (*dict[str, Any] | None*) – A set of user attributes to be carried along with this LGDO.

_abc_impl = <_abc._abc_data object>

property dt: *Array*

property dt_units: str

resize_wf_len(*new_len*)

Alias for *wf_len.setter*, for when we want to make it clear in the code that memory is being reallocated.

property t0: *Array*

property t0_units: str

property values: *ArrayOfEqualSizedArrays* | *VectorOfVectors*

property values_units: str

view_as(*library*, *with_units=False*, *cols=None*, *prefix=''*)

View the waveform data as a third-party format data structure.

See also:

LGDO.view_as

Return type
pd.DataFrame | np.NDArray | ak.Array

property wf_len: int

Submodules

lgdo.cli module

legend-pydataobj's command line interface utilities.

lgdo.cli.**lh5concat**(*args=None*)

Command line interface for concatenating array-like LGDOs in LH5 files.

lgdo.cli.**lh5ls**(*args=None*)

`lh5.show()` command line interface.

lgdo.lgdo_utils module

lgdo.lgdo_utils.**copy**(*obj*, *dtype=None*)

lgdo.lgdo_utils.**expand_path**(*path*, *substitute=None*, *list=False*, *base_path=None*)

Return type
str | list

lgdo.lgdo_utils.**expand_vars**(*expr*, *substitute=None*)

Return type
str

lgdo.lgdo_utils.**get_element_type**(*obj*)

Return type
str

lgdo.lgdo_utils.**parse_datatype**(*datatype*)

Return type
tuple[str, tuple[int, ...], str | list[str]]

lgdo.lh5_store module

Warning: This subpackage is deprecated, use *lgdo.lh5*.

class lgdo.lh5_store.**LH5Iterator**(*lh5_files*, *groups*, *base_path=''*, *entry_list=None*, *entry_mask=None*, *field_mask=None*, *buffer_len=3200*, *friend=None*)

Bases: *LH5Iterator*

Warning: This class is deprecated, use *lgdo.lh5.iterator.LH5Iterator*.

Parameters

- **lh5_files** (*str | list[str]*) – file or files to read from. May include wildcards and environment variables.
- **groups** (*str | list[str]*) – HDF5 group(s) to read. If a list is provided for both lh5_files and group, they must be the same size. If a file is wild-carded, the same group will be assigned to each file found
- **entry_list** (*list[int] | list[list[int]] | None*) – list of entry numbers to read. If a nested list is provided, expect one top-level list for each file, containing a list of local entries. If a list of ints is provided, use global entries.
- **entry_mask** (*list[bool] | list[list[bool]] | None*) – mask of entries to read. If a list of arrays is provided, expect one for each file. Ignore if a selection list is provided.
- **field_mask** (*dict[str, bool] | list[str] | tuple[str] | None*) – mask of which fields to read. See `LH5Store.read()` for more details.
- **buffer_len** (*int*) – number of entries to read at a time while iterating through files.
- **friend** (*Iterator | None*) – a "friend" LH5Iterator that will be read in parallel with this. The friend should have the same length and entry list. A single LH5 table containing columns from both iterators will be returned.

_abc_impl = <_abc._abc_data object>

read_object(*name*, *lh5_file*, *start_row=0*, *n_rows=9223372036854775807*, *idx=None*, *field_mask=None*, *obj_buf=None*, *obj_buf_start=0*, *decompress=True*)

Warning: This method is deprecated, use *lgdo.lh5.iterator.LH5Iterator.read()*.

Return type
tuple[Array | Scalar | Struct | VectorOfVectors, int]

write_object(*obj*, *name*, *lh5_file*, *group='/'*, *start_row=0*, *n_rows=None*, *wo_mode='append'*, *write_start=0*, ***h5py_kwargs*)

Warning: This method is deprecated, use `lgdo.lh5.iterator.LH5Iterator.write()`.

class lgdo.lh5_store.**LH5Store**(*base_path=''*, *keep_open=False*)

Bases: *LH5Store*

> **Warning:** This class is deprecated, use `lgdo.lh5.iterator.LH5Store`.

Parameters

- **base_path** (*str*) – directory path to prepend to LH5 files.
- **keep_open** (*bool*) – whether to keep files open by storing the `h5py` objects as class attributes.

read_object(*name*, *lh5_file*, ***kwargs*)

> **Warning:** This method is deprecated, use *lgdo.lh5.store.LH5Store.read()*.

Return type
tuple[Array | Scalar | Struct | VectorOfVectors, int]

write_object(*obj*, *name*, *lh5_file*, ***kwargs*)

> **Warning:** This method is deprecated, use *lgdo.lh5.store.LH5Store.write()*.

Return type
tuple[Array | Scalar | Struct | VectorOfVectors, int]

lgdo.lh5_store.**load_dfs**(*f_list*, *par_list*, *lh5_group=''*, *idx_list=None*)

> **Warning:** This function is deprecated, use *lgdo.types.lgdo.LGDO.view_as()* to view LGDO data as a Pandas data structure.

Return type
DataFrame

lgdo.lh5_store.**load_nda**(*f_list*, *par_list*, *lh5_group=''*, *idx_list=None*)

> **Warning:** This function is deprecated, use *lgdo.types.lgdo.LGDO.view_as()* to view LGDO data as a NumPy data structure.

Return type
dict[str, *ndarray*]

lgdo.lh5_store.**ls**(*lh5_file*, *lh5_group=''*)

> **Warning:** This function is deprecated, import *lgdo.lh5.tools.ls()*.

Return type
list[str]

lgdo.lh5_store.**show**(*lh5_file*, *lh5_group='/'*, *attrs=False*, *indent=''*, *header=True*)

> **Warning:** This function is deprecated, import *lgdo.lh5.tools.show()*.

lgdo.logging module

This module implements some helpers for setting up logging.

lgdo.logging.**setup**(*level=20*, *logger=None*)

Setup a colorful logging output.

If *logger* is None, sets up only the lgdo logger.

Parameters

- **level** (*int*) – logging level (see logging module).
- **logger** (*Logger | None*) – if not *None*, setup this logger.

Examples

```
>>> from lgdo import logging
>>> logging.setup(level=logging.DEBUG)
```

lgdo.units module

lgdo.utils module

Implements utilities for LEGEND Data Objects.

class lgdo.utils.**NumbaDefaults**

Bases: MutableMapping

Bare-bones class to store some Numba default options. Defaults values are set from environment variables

Examples

Set all default option values for a processor at once by expanding the provided dictionary:

```
>>> from numba import guvectorize
>>> from lgdo.utils import numba_defaults_kwargs as nb_kwargs
>>> @guvectorize([], "", **nb_kwargs, nopython=True) # def proc(...): ...
```

Customize one argument but still set defaults for the others:

```
>>> from lgdo.utils import numba_defaults as nb_defaults
>>> @guvectorize([], "", **nb_defaults(cache=False) # def proc(...): ...
```

Override global options at runtime:

```
>>> from lgdo.utils import numba_defaults
>>> # must set options before explicitly importing lgdo modules!
>>> numba_defaults.cache = False
>>> numba_defaults.boundscheck = True
>>> from lgdo import compression # imports of numbified functions happen here
>>> compression.encode(...)
```

_abc_impl = <_abc._abc_data object>

lgdo.utils.**get_element_type**(*obj*)

Get the LGDO element type of a scalar or array.

For use in LGDO datatype attributes.

Parameters

obj (*object*) – if a `str`, will automatically return `string` if the object has a `numpy.dtype`, that will be used for determining the element type otherwise will attempt to case the type of the object to a `numpy.dtype`.

Returns

element_type – A string stating the determined element type of the object.

Return type

str

lgdo.utils.**getenv_bool**(*name*, *default=False*)

Get environment value as a boolean, returning True for 1, t and true (caps-insensitive), and False for any other value and default if undefined.

Return type

bool

2.4 Developer's guide

Note: The https://learn.scientific-python.org webpages are an extremely valuable learning resource for Python software developer. The reader is referred to that for any detail not covered in the following guide.

The following rules and conventions have been established for the package development and are enforced throughout the entire code base. Merge requests that do not comply to the following directives will be rejected.

To start developing *legend-pydataobj*, fork the remote repository to your personal GitHub account (see About Forks). If you have not set up your ssh keys on the computer you will be working on, please follow GitHub's instructions. Once you have your own fork, you can clone it via (replace "yourusername" with your GitHub username):

```
$ git clone git@github.com:yourusername/legend-pydataobj.git
```

All extra tools needed to develop *legend-pydataobj* are listed as optional dependencies and can be installed via pip by running:

```
$ cd legend-pydataobj
$ pip install -e '.[all]'  # single quotes are not needed on bash
```

Important: Pip's `--editable | -e` flag let's you install the package in "developer mode", meaning that any change to the source code will be directly propagated to the installed package and importable in scripts.

Tip: It is strongly recommended to work inside a virtual environment, which guarantees reproductibility and isolation. For more details, see learn.scientific-python.org.

2.4.1 Code style

- All functions and methods (arguments and return types) must be type-annotated. Type annotations for variables like class attributes are also highly appreciated.
- Messaging to the user is managed through the `logging` module. Do not add `print()` statements. To make a logging object available in a module, add this:

  ```
  import logging

  log = logging.getLogger(__name__)
  ```

  at the top. In general, try to keep the number of `logging.debug()` calls low and use informative messages. `logging.info()` calls should be reserved for messages from high-level routines and very sporadic. Good code is never too verbose.
- If an error condition leading to undefined behavior occurs, raise an exception. try to find the most suitable between the built-in exceptions, otherwise `raise RuntimeError("message")`. Do not raise `Warnings`, use `logging.warning()` for that and don't abort the execution.
- Warning messages (emitted when a problem is encountered that does not lead to undefined behavior) must be emitted through `logging.warning()` calls.

A set of pre-commit hooks is configured to make sure that *legend-pydataobj* coherently follows standard coding style conventions. The pre-commit tool is able to identify common style problems and automatically fix them, wherever

possible. Configured hooks are listed in the `.pre-commit-config.yaml` file at the project root folder. They are run remotely on the GitHub repository through the pre-commit bot, but should also be run locally before submitting a pull request:

```
$ cd legend-pydataobj
$ pip install '.[test]'
$ pre-commit run --all-files  # analyse the source code and fix it wherever possible
$ pre-commit install          # install a Git pre-commit hook (strongly recommended)
```

For a more comprehensive guide, check out the learn.scientific-python.org documentation about code style.

2.4.2 Testing

- The *legend-pydataobj* test suite is available below `tests/`. We use pytest to run tests and analyze their output. As a starting point to learn how to write good tests, reading of the relevant learn.scientific-python.org webpage is recommended.
- Unit tests are automatically run for every push event and pull request to the remote Git repository on a remote server (currently handled by GitHub actions). Every pull request must pass all tests before being approved for merging. Running the test suite is simple:

  ```
  $ cd legend-pydataobj
  $ pip install '.[test]'
  $ pytest
  ```

- Additionally, pull request authors are required to provide tests with sufficient code coverage for every proposed change or addition. If necessary, high-level functional tests should be updated. We currently rely on codecov.io to keep track of test coverage. A local report, which must be inspected before submitting pull requests, can be generated by running:

  ```
  $ pytest --cov=lgdo
  ```

2.4.3 Documentation

We adopt best practices in writing and maintaining *legend-pydataobj*'s documentation. When contributing to the project, make sure to implement the following:

- Documentation should be exclusively available on the Project website legend-pydataobj.readthedocs.io. No READMEs, GitHub/LEGEND wiki pages should be written.
- Pull request authors are required to provide sufficient documentation for every proposed change or addition.
- Documentation for functions, classes, modules and packages should be provided as Docstrings along with the respective source code. Docstrings are automatically converted to HTML as part of the *legend-pydataobj* package API documentation.
- General guides, comprehensive tutorials or other high-level documentation (e.g. referring to how separate parts of the code interact between each other) must be provided as separate pages in `docs/source/` and linked in the table of contents.
- Jupyter notebooks should be added to the main Git repository below `docs/source/notebooks`.
- Before submitting a pull request, contributors are required to build the documentation locally and resolve and warnings or errors.

Writing documentation

We adopt the following guidelines for writing documentation:

- Documentation source files must formatted in reStructuredText (reST). A reference format specification is available on the Sphinx reST usage guide. Usage of Cross-referencing syntax in general and for Python objects in particular is recommended. We also support cross-referencing external documentation via sphinx.ext.intersphinx, when referring for example to `pandas.DataFrame`.
- To document Python objects, we also adopt the NumPy Docstring style. Examples are available here.
- We support also the Markdown format through the MyST-Parser.
- Jupyter notebooks placed below `docs/source/notebooks` are automatically rendered to HTML pages by the nbsphinx extension.

Building documentation

Scripts and tools to build documentation are located below `docs/`. To build documentation, `sphinx` and a couple of additional Python packages are required. You can get all the needed dependencies by running:

```
$ cd legend-pydataobj
$ pip install '.[docs]'
```

Pandoc is also required to render Jupyter notebooks. To build documentation, run the following commands:

```
$ cd docs
$ make clean
$ make
```

Documentation can be then displayed by opening `build/html/index.html` with a web browser. Documentation for the `legend-pydataobj` website is built and deployed by Read the Docs.

2.4.4 Versioning

Collaborators with push access to the GitHub repository that wish to release a new project version must implement the following procedures:

- Semantic versioning is adopted. The version string uses the `MAJOR.MINOR.PATCH` format.
- To release a new **minor** or **major version**, the following procedure should be followed:
  1. A new branch with name `releases/vMAJOR.MINOR` (note the `v`) containing the code at the intended stage is created
  2. The commit is tagged with a descriptive message: `git tag vMAJOR.MINOR.0 -m 'short descriptive message here'` (note the `v`)
  3. Changes are pushed to the remote:

     ```
     $ git push origin releases/vMAJOR.MINOR
     $ git push origin refs/tags/vMAJOR.MINOR.0
     ```

- To release a new **patch version**, the following procedure should be followed:
  1. A commit with the patch is created on the relevant release branch `releases/vMAJOR.MINOR`
  2. The commit is tagged: `git tag vMAJOR.MINOR.PATCH` (note the `v`)

3. Changes are pushed to the remote:

```
$ git push origin releases/vMAJOR.MINOR
$ git push origin refs/tags/vMAJOR.MINOR.PATCH
```

- To upload the release to the Python Package Index, a new release must be created through the GitHub interface, associated to the just created tag. Usage of the "Generate release notes" option is recommended.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

P

R

S

T

U

V

W

Z