



# Hierarchical Data Modeling Framework

**HDMF**

*Release 3.13.1.dev9+g3146c97*

unknown

Apr 26, 2024



# GETTING STARTED

<b>1</b>	<b>Installing HDMF</b>	<b>3</b>
1.1	Option 1: Using pip . . . . .	3
1.2	Option 2: Using conda . . . . .	3
<b>2</b>	<b>Tutorials</b>	<b>5</b>
2.1	GenericDataChunkIterator Tutorial . . . . .	5
2.2	DynamicTable Tutorial . . . . .	8
2.3	MultiContainerInterface . . . . .	12
2.4	TermSet . . . . .	17
2.5	AlignedDynamicTable . . . . .	21
2.6	HERD: HDMF External Resources Data Structure . . . . .	27
2.7	DynamicTable How-To Guide . . . . .	34
<b>3</b>	<b>Introduction</b>	<b>49</b>
<b>4</b>	<b>Software Architecture</b>	<b>51</b>
4.1	Main Concepts . . . . .	53
4.2	Additional Concepts . . . . .	55
<b>5</b>	<b>Citing HDMF</b>	<b>59</b>
5.1	BibTeX entry . . . . .	59
5.2	Using RRID . . . . .	59
5.3	Using duecredit . . . . .	59
<b>6</b>	<b>API Documentation</b>	<b>61</b>
6.1	hdmf.common package . . . . .	61
6.2	hdmf.container module . . . . .	88
6.3	hdmf.build package . . . . .	93
6.4	hdmf.spec package . . . . .	109
6.5	hdmf.backends package . . . . .	129
6.6	hdmf.data_utils module . . . . .	144
6.7	hdmf.utils module . . . . .	150
6.8	hdmf.validate package . . . . .	156
6.9	hdmf.testing package . . . . .	161
6.10	hdmf package . . . . .	163
<b>7</b>	<b>Extending Standards</b>	<b>169</b>
7.1	Creating new Extensions . . . . .	169
7.2	Saving Extensions . . . . .	171
7.3	Incorporating extensions . . . . .	172
7.4	Documenting Extensions . . . . .	174

7.5	Further Reading . . . . .	174
<b>8</b>	<b>Building API Classes</b>	<b>175</b>
8.1	The register_class function/decorator . . . . .	175
<b>9</b>	<b>Export</b>	<b>177</b>
9.1	FAQ . . . . .	177
<b>10</b>	<b>Validating HDMF Data</b>	<b>181</b>
<b>11</b>	<b>Support for the HDMF Specification Language</b>	<b>183</b>
<b>12</b>	<b>Installing HDMF for Developers</b>	<b>185</b>
12.1	Set up a virtual environment . . . . .	185
12.2	Install from GitHub . . . . .	186
12.3	Run tests . . . . .	186
12.4	Install latest pre-release . . . . .	187
<b>13</b>	<b>Contributing Guide</b>	<b>189</b>
13.1	Code of Conduct . . . . .	189
13.2	Types of Contributions . . . . .	189
13.3	Contributing Patches and Changes . . . . .	190
13.4	Style Guides . . . . .	191
13.5	Endorsement . . . . .	191
13.6	License and Copyright . . . . .	192
<b>14</b>	<b>How to Make a Roundtrip Test</b>	<b>193</b>
14.1	H5RoundTripMixin . . . . .	193
<b>15</b>	<b>Software Process</b>	<b>195</b>
15.1	Continuous Integration . . . . .	195
15.2	Coverage . . . . .	195
15.3	Installation Requirements . . . . .	195
15.4	Testing Requirements . . . . .	195
15.5	Documentation Requirements . . . . .	196
15.6	Versioning and Releasing . . . . .	196
<b>16</b>	<b>How to Make a Release</b>	<b>197</b>
16.1	Prerequisites . . . . .	197
16.2	Documentation conventions . . . . .	198
16.3	Publish release on PyPI: Step-by-step . . . . .	198
16.4	Publish release on conda-forge: Step-by-step . . . . .	199
<b>17</b>	<b>How to Update Requirements Files</b>	<b>201</b>
17.1	requirements.txt . . . . .	201
17.2	requirements-(dev doc opt).txt . . . . .	201
17.3	requirements-min.txt . . . . .	202
<b>18</b>	<b>Copyright</b>	<b>203</b>
<b>19</b>	<b>License</b>	<b>205</b>
	<b>Python Module Index</b>	<b>207</b>
	<b>Index</b>	<b>209</b>

HDMF is a Python package for working with standardizing, reading, and writing hierarchical object data.

HDMF is a by-product of the [Neurodata Without Borders \(NWB\)](#) project. The goal of NWB was to enable collaborative science within the neurophysiology and systems neuroscience communities through data standardization. The team of neuroscientists and software developers involved with NWB recognize that adoption of a unified data format is an important step toward breaking down the barriers to data sharing in neuroscience. HDMF was central to the NWB development efforts, and has since been split off with the intention of providing it as an open-source tool for other scientific communities.

If you use HDMF in your research, please use the following citation:

A. J. Tritt et al., “HDMF: Hierarchical Data Modeling Framework for Modern Science Data Standards,” 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 2019, pp. 165-179, doi: 10.1109/BigData47090.2019.9005648.



## INSTALLING HDMF

HDMF requires having Python 3.8, 3.9, 3.10, 3.11, or 3.12 installed. If you don't have Python installed and want the simplest way to get started, we recommend you install and use the [Anaconda Distribution](#). It includes Python, NumPy, and many other commonly used packages for scientific computing and data science.

HDMF can be installed with `pip`, `conda`, or from source. HDMF works on Windows, macOS, and Linux.

### 1.1 Option 1: Using pip

If you are a beginner to programming in Python and using Python tools, we recommend that you install HDMF by running the following command in a terminal or command prompt:

```
pip install hdmf
```

### 1.2 Option 2: Using conda

You can also install HDMF using `conda` by running the following command in a terminal or command prompt:

```
conda install -c conda-forge hdmf
```





## 2.1 GenericDataChunkIterator Tutorial

This is a tutorial for interacting with *GenericDataChunkIterator* objects. This tutorial is written for beginners and does not describe the full capabilities and nuances of the functionality. This tutorial is designed to give you basic familiarity with how *GenericDataChunkIterator* works and help you get started with creating a specific instance for your data format or API access pattern.

### 2.1.1 Introduction

The *GenericDataChunkIterator* class represents a semi-abstract version of a *AbstractDataChunkIterator* that automatically handles the selection of buffer regions and resolves communication of compatible chunk regions within a *H5DataIO* wrapper. It does not, however, know how data (values) or metadata (data type, full shape) ought to be directly accessed. This is by intention to be fully agnostic to a range of indexing methods and format-independent APIs, rather than make strong assumptions about how data ranges are to be sliced.

### 2.1.2 Constructing a simple child class

We will begin with a simple example case of data access to a standard Numpy array. To create a *GenericDataChunkIterator* that accomplishes this, we begin by defining our child class.

```
import numpy as np

from hdmf.data_utils import GenericDataChunkIterator

class NumpyArrayDataChunkIterator(GenericDataChunkIterator):
    def __init__(self, array: np.ndarray, **kwargs):
        self.array = array
        super().__init__(**kwargs)

    def _get_data(self, selection):
        return self.array[selection]

    def _get_maxshape(self):
        return self.array.shape

    def _get_dtype(self):
        return self.array.dtype
```

(continues on next page)

(continued from previous page)

```
# To instantiate this class on an array to allow iteration over buffer_shapes,
my_array = np.random.randint(low=0, high=10, size=(12, 6), dtype="int16")
my_custom_iterator = NumpyArrayDataChunkIterator(array=my_array)

# and this iterator now behaves as a standard Python generator (i.e., it can only be
↪ exhausted once)
# that returns DataChunk objects for each buffer.
for buffer in my_custom_iterator:
    print(buffer.data)
```

```
[[3 9 4 0 2 4]
 [0 4 9 6 3 5]
 [0 3 8 5 4 8]
 [8 3 7 8 5 2]
 [8 0 2 5 3 9]
 [9 3 2 2 9 8]
 [2 5 1 9 3 9]
 [1 8 6 1 3 6]
 [2 5 0 4 3 0]
 [3 6 6 3 7 5]
 [2 1 1 3 2 6]
 [1 4 5 0 4 8]]
```

### 2.1.3 Intended use for advanced data I/O

Of course, the real use case for this class is intended for when the amount of data stored on a hard drive is larger than what can be read into RAM. Hence the goal is to read only an amount of data with a size in gigabytes (GB) at or below the *buffer\_gb* argument (defaults to 1 GB).

```
# This design can be seen if we increase the amount of data in our example code
my_array = np.random.randint(low=0, high=10, size=(20000, 5000), dtype="int32")
my_custom_iterator = NumpyArrayDataChunkIterator(array=my_array, buffer_gb=0.2)

for j, buffer in enumerate(my_custom_iterator, start=1):
    print(f"Buffer number {j} returns data from selection: {buffer.selection}")
```

```
Buffer number 1 returns data from selection: (slice(0, 12640, None), slice(0, 3160,
↪ None))
Buffer number 2 returns data from selection: (slice(0, 12640, None), slice(3160, 5000,
↪ None))
Buffer number 3 returns data from selection: (slice(12640, 20000, None), slice(0, 3160,
↪ None))
Buffer number 4 returns data from selection: (slice(12640, 20000, None), slice(3160,
↪ 5000, None))
```

**Note:** Technically, in this example the total data is still fully loaded into RAM from the initial Numpy array. A more accurate use case would be achieved from writing the *test\_array* to a temporary file on your system and loading it back

with `np.memmap`, which is a subtype of Numpy arrays that do not immediately load the data.

## 2.1.4 Writing to an HDF5 file with full control of shape arguments

The true intention of returning data selections of this form, and within a `DataChunk` object, is to write these piecewise to an HDF5 dataset.

```
# This is where the importance of the underlying `chunk_shape` comes in, and why it is
↳critical to performance
# that it perfectly subsets the `buffer_shape`.
import h5py

maxshape = (20000, 5000)
buffer_shape = (10000, 2500)
chunk_shape = (1000, 250)

my_array = np.random.randint(low=0, high=10, size=maxshape, dtype="int32")
my_custom_iterator = NumpyArrayDataChunkIterator(array=my_array, buffer_shape=buffer_
↳shape, chunk_shape=chunk_shape)
out_file = "my_temporary_test_file.hdf5"
with h5py.File(name=out_file, mode="w") as f:
    dset = f.create_dataset(name="test", shape=maxshape, dtype="int16", chunks=my_custom_
↳iterator.chunk_shape)
    for buffer in my_custom_iterator:
        dset[buffer.selection] = buffer.data
# Remember to remove the temporary file after running this and exploring the contents!
```

**Note:** Here we explicitly set the *chunks* value in the HDF5 dataset object; however, a nice part of the design of this iterator is that when wrapped in a `hdmf.backends.hdf5.h5_utils.H5DataIO` that is called within a `hdmf.backends.hdf5.h5tools.HDF5IO` context with a corresponding `hdmf.container.Container`, these details will be automatically parsed.

**Note:** There is some overlap here in nomenclature between HDMF and HDF5. The term *chunk* in both HDMF and HDF5 refer to a subset of dataset, however, in HDF5 a chunk is a piece of dataset on disk, whereas in the context of the *DataChunk* iteration is a block of data in memory. As such, the requirements on the shape and size of chunks are different. In HDF5 these chunks are pieces of a dataset that get compressed and cached together, and they should usually be small in size for optimal performance (typically 1 MB or less). In contrast, a *DataChunk* in HDMF acts as a block of data for writing data to dataset, and spans multiple HDF5 chunks to improve performance. This is achieved by avoiding repeat updates to the same *Chunk* in the HDF5 file, *DataChunk* objects for write should align with *Chunks* in the HDF5 file, i.e., the `DataChunk.selection` should fully cover one or more *Chunks* in the HDF5 file to avoid repeat updates to the same *Chunks* in the HDF5 file. This is what the *buffer* of the `:py:class`~hdmf.data_utils.GenericDataChunkIterator`` does, which upon each iteration returns a single *DataChunk* object (by default > 1 GB) that perfectly spans many HDF5 chunks (by default < 1 MB) to help reduce the number of small I/O operations and help improve performance. In practice, the *buffer* should usually be even larger than the default, i.e., as much free RAM as can be safely used.

Remove the test file

```
import os
if os.path.exists(out_file):
```

(continues on next page)

```
os.remove(out_file)
```

## 2.2 DynamicTable Tutorial

This is a tutorial for interacting with *DynamicTable* objects. This tutorial is written for beginners and does not describe the full capabilities and nuances of *DynamicTable* functionality. Please see the *DynamicTable How-To Guide* for more complete documentation. This tutorial is designed to give you basic familiarity with how *DynamicTable* works and help you get started with creating a *DynamicTable*, adding columns and rows to a *DynamicTable*, and accessing data in a *DynamicTable*.

### 2.2.1 Introduction

The *DynamicTable* class represents a column-based table to which you can add custom columns. It consists of a name, a description, a list of row IDs, and a list of columns.

### 2.2.2 Constructing a table

To create a *DynamicTable*, call the constructor for *DynamicTable* with a string name and string description.

```
from hdmf.common import DynamicTable

users_table = DynamicTable(
    name='users',
    description='a table containing data/metadata about users, one user per row',
)
```

### 2.2.3 Adding columns

You can add columns to a *DynamicTable* using *DynamicTable.add\_column*.

```
users_table.add_column(
    name='first_name',
    description='the first name of the user',
)

users_table.add_column(
    name='last_name',
    description='the last name of the user',
)
```

## 2.2.4 Adding ragged array columns

You may want to add columns to your table that have a different number of entries per row. This is called a “ragged array column”. To do this, pass `index=True` to `DynamicTable.add_column`.

```
users_table.add_column(
    name='phone_number',
    description='the phone number of the user',
    index=True,
)
```

## 2.2.5 Adding rows

You can add rows to a `DynamicTable` using `DynamicTable.add_row`. You must pass in a keyword argument for every column in the table. Ragged array column arguments should be passed in as lists or numpy arrays. The ID of the row will automatically be set and incremented for every row, starting at 0.

```
# id will be set to 0 automatically
users_table.add_row(
    first_name='Grace',
    last_name='Hopper',
    phone_number=['123-456-7890'],
)

# id will be set to 1 automatically
users_table.add_row(
    first_name='Alan',
    last_name='Turing',
    phone_number=['555-666-7777', '888-111-2222'],
)
```

## 2.2.6 Displaying the table contents as a pandas DataFrame

`pandas` is a popular data analysis tool for working with tabular data. Convert your `DynamicTable` to a pandas `DataFrame` using `DynamicTable.to_dataframe`.

```
users_df = users_table.to_dataframe()
users_df
```

Accessing the table as a `DataFrame` provides you with powerful methods for indexing, selecting, and querying tabular data from `pandas`.

Get the “last\_name” column as a pandas `Series`:

```
users_df['last_name']
```

```
id
0    Hopper
1    Turing
Name: last_name, dtype: object
```

The index of the `DataFrame` is automatically set to the table IDs. Get the row with ID = 0 as a pandas `Series`:

```
users_df.loc[0]
```

```
first_name      Grace
last_name       Hopper
phone_number    [123-456-7890]
Name: 0, dtype: object
```

Get single cells of the table by indexing with both ID and column name:

```
print('My first user:', users_df.loc[0, 'first_name'], users_df.loc[0, 'last_name'])
```

```
My first user: Grace Hopper
```

## 2.2.7 Adding columns that reference rows of other `DynamicTable` objects

You can create a column that references rows of another *DynamicTable*. This is analogous to a foreign key in a relational database. To do this, use the table keyword argument for *DynamicTable.add\_column* and set it to the other table.

```
# create a new table of users
users_table = DynamicTable(
    name='users',
    description='a table containing data/metadata about users, one user per row',
)

# add simple columns to this table
users_table.add_column(
    name='first_name',
    description='the first name of the user',
)
users_table.add_column(
    name='last_name',
    description='the last name of the user',
)

# create a new table of addresses to reference
addresses_table = DynamicTable(
    name='addresses',
    description='a table containing data/metadata about addresses, one address per row',
)
addresses_table.add_column(
    name='street_address',
    description='the street number and address',
)
addresses_table.add_column(
    name='city',
    description='the city of the address',
)

# add rows to the addresses table
addresses_table.add_row(
```

(continues on next page)

(continued from previous page)

```

    street_address='123 Main St',
    city='Springfield'
)
addresses_table.add_row(
    street_address='45 British Way',
    city='London'
)

# add a column to the users table that references rows of the addresses table
users_table.add_column(
    name='address',
    description='the address of the user',
    table=addresses_table
)

# add rows to the users table
users_table.add_row(
    first_name='Grace',
    last_name='Hopper',
    address=0 # <-- row index of the address table
)

users_table.add_row(
    first_name='Alan',
    last_name='Turing',
    address=1 # <-- row index of the address table
)

```

## 2.2.8 Displaying the contents of a table with references to another table

Earlier, we converted a *DynamicTable* to a *DataFrame* using *DynamicTable.to\_dataframe* and printed the *DataFrame* to see its contents. This also works when the *DynamicTable* contains a column that references another table. However, the entries for this column for each row will be printed as a nested *DataFrame*. This can be difficult to read, so to view only the row indices of the referenced table, pass `index=True` to *DynamicTable.to\_dataframe*.

```

users_df = users_table.to_dataframe(index=True)
users_df

```

You can then access the referenced table using the `table` attribute of the column object. This is useful when reading a table from a file where you may not have a variable to access the referenced table.

First, use *DynamicTable.\_\_getitem\_\_* (square brackets notation) to get the *DynamicTableRegion* object representing the column. Then access its `table` attribute to get the addresses table and convert the table to a *DataFrame*.

```

address_column = users_table['address']
read_addresses_table = address_column.table
addresses_df = read_addresses_table.to_dataframe()

```

Get the addresses corresponding to the rows of the users table:

```

address_indices = users_df['address'] # pandas Series of row indices into the addresses_
↪ table

```

(continues on next page)

(continued from previous page)

```
addresses_df.iloc[address_indices] # use .iloc because these are row indices not ID_
↪ values
```

---

**Note:** The indices returned by `users_df['address']` are row indices and not the ID values of the table. However, if you are using default IDs, these values will be the same.

---

You now know the basics of creating *DynamicTable* objects and reading data from them, including tables that have ragged array columns and references to other tables. Learn more about working with *DynamicTable* in the *DynamicTable How-To Guide*, including:

- ragged array columns with references to other tables
- nested ragged array columns
- columns with multidimensional array data
- columns with enumerated (categorical) data
- accessing data and properties from the column objects directly
- writing and reading tables to a file
- writing expandable tables
- defining subclasses of *DynamicTable*

## 2.3 MultiContainerInterface

This is a guide to creating custom API classes with the `MultiContainerInterface` class.

### 2.3.1 Introduction

The *MultiContainerInterface* class provides an easy and convenient way to create standard methods for a container class that contains a collection of containers of a specified type. For example, let's say you want to define a class `MyContainerHolder` that contains a collection of `MyContainer` objects. By having `MyContainerHolder` extend *MultiContainerInterface* and specifying certain configuration settings in the class, your `MyContainerHolder` class would be generated with:

1. an attribute for a labelled dictionary that holds `MyContainer` objects
2. an `__init__` method to initialize `MyContainerHolder` with a collection of `MyContainer` objects
3. a method to add `MyContainer` objects to the dictionary
4. access of items from the dictionary using `__getitem__` (square bracket notation)
5. a method to get `MyContainer` objects from the dictionary (optional)
6. a method to create `MyContainer` objects and add them to the dictionary (optional)



### 2.3.2 Specifying the class configuration

To specify the class configuration for a *MultiContainerInterface* subclass, define the variable `__clsconf__` in the new class. `__clsconf__` should be set to a dictionary with three required keys, 'attr', 'type', and 'add'.

The 'attr' key should map to a string value that is the name of the attribute that will be created to hold the collection of container objects.

The 'type' key should map to a type or a tuple of types that says what objects are allowed in this collection.

The 'add' key should map to a string value that is the name of the method to be generated that allows users to add a container to the collection.

```
from hdmf.container import Container, MultiContainerInterface

class ContainerHolder(MultiContainerInterface):

    __clsconf__ = {
        'attr': 'containers',
        'type': Container,
        'add': 'add_container',
    }
```

The above code will generate:

1. the attribute `containers` as a *LabelledDict* that holds *Container* objects
2. the `__init__` method which accepts a collection of *Container* objects
3. the `add_container` method that allows users to add *Container* objects to the `containers` dictionary.

Here is an example of instantiating the new *ContainerHolder* class and using the generated add method.

```
obj1 = Container('obj1')
obj2 = Container('obj2')
holder1 = ContainerHolder()
holder1.add_container(obj1)
holder1.add_container(obj2)
holder1.containers  # this is a LabelledDict where the keys are the name of the container
# i.e., {'obj1': obj1, 'obj2': obj2}
```

### 2.3.3 Constructor options

The constructor accepts a dict/list/tuple of *Container* objects, a single *Container* object, or None. If a dict is passed, only the dict values are used. You can specify the argument as a keyword argument with the attribute name as the keyword argument key.

```
holder2 = ContainerHolder(obj1)
holder3 = ContainerHolder([obj1, obj2])
holder4 = ContainerHolder({'unused_key1': obj1, 'unused_key2': obj2})
holder5 = ContainerHolder(containers=obj1)
```

By default, the new class has the 'name' attribute set to the name of the class, but a user-specified name can be provided in the constructor.

```
named_holder = ContainerHolder(name='My Holder')
```

### 2.3.4 Adding containers to the collection

Similar to the constructor, the generated add method accepts a dict/list/tuple of `Container` objects or a single `Container` object. If a dict is passed, only the dict values are used.

```
holder6 = ContainerHolder()
holder6.add_container(obj1)

holder7 = ContainerHolder()
holder7.add_container([obj1, obj2])

holder8 = ContainerHolder()
holder8.add_container({'unused_key1': obj1, 'unused_key2': obj2})

holder9 = ContainerHolder()
holder9.add_container(containers=obj1)
```

### 2.3.5 Getting items from the collection

You can access a container in the collection by using the name of the container within square brackets. As a convenience, if there is only one item in the collection, you can use `None` within square brackets.

```
holder10 = ContainerHolder(obj1)
holder10['obj1']
holder10[None]
```

### 2.3.6 Getting items from the collection using a custom getter

You can use the 'get' key in `__clsconf__` to generate a getter method as an alternative to using the square bracket notation for accessing items from the collection. Like the square bracket notation, if there is only one item in the collection, you can omit the name or pass `None` to the getter method.

The 'get' key should map to a string value that is the name of the getter method to be generated. The 'get' key in `__clsconf__` is optional.

```
class ContainerHolderWithGet(MultiContainerInterface):

    __clsconf__ = {
        'attr': 'containers',
        'type': Container,
        'add': 'add_container',
        'get': 'get_container',
    }

holder11 = ContainerHolderWithGet(obj1)
holder11.get_container('obj1')
holder11.get_container()
```

### 2.3.7 Creating and adding items to the collection using a custom create method

You can use the 'create' key in `__clsconf__` to generate a create method as a convenience method so that users do not need to initialize the Container object and then add it to the collection. Those two steps are combined into one line. The arguments to the custom create method are the same as the arguments to the Container's `__init__` method, but the `__init__` method *must* be defined using [docval](#). The created object will be returned by the create method.

The 'create' key should map to a string value that is the name of the create method to be generated. The 'create' key in `__clsconf__` is optional.

```
class ContainerHolderWithCreate(MultiContainerInterface):
```

```
    __clsconf__ = {
        'attr': 'containers',
        'type': Container,
        'add': 'add_container',
        'create': 'create_container',
    }
```

```
holder12 = ContainerHolderWithCreate()
holder12.create_container('obj1')
```

### 2.3.8 Specifying multiple types allowed in the collection

The 'type' key in `__clsconf__` allows specifying a single type or a list/tuple of types.

You cannot specify the 'create' key in `__clsconf__` when multiple types are allowed in the collection because it cannot be determined which type to initialize.

```
from hdmf.container import Data
```

```
class ContainerHolderWithMultipleTypes(MultiContainerInterface):
```

```
    __clsconf__ = {
        'attr': 'things',
        'type': (Container, Data),
        'add': 'add_thing',
    }
```

### 2.3.9 Specifying multiple collections

You can support multiple collections in your *MultiContainerInterface* subclass by setting the `__clsconf__` variable to a list of dicts instead of a single dict.

When specifying multiple collections, square bracket notation access of items (i.e., calling `__getitem__`) is not supported, because it is not clear which collection to get the desired item from.

```
from hdmf.container import Data
```

(continues on next page)

(continued from previous page)

```
class MultiCollectionHolder(MultiContainerInterface):

    __clsconf__ = [
        {
            'attr': 'containers',
            'type': Container,
            'add': 'add_container',
        },
        {
            'attr': 'data',
            'type': Data,
            'add': 'add_data',
        },
    ]
```

### 2.3.10 Managing container parents

If the parent of the container being added is not already set, then the parent will be set to the containing object.

```
obj3 = Container('obj3')
holder13 = ContainerHolder(obj3)
obj3.parent # this is holder13
```

*LabelledDict* objects support removal of an item using the `del` operator or the *pop* method. If the parent of the container being removed is the containing object, then its parent will be reset to `None`.

```
del holder13.containers['obj3']
obj3.parent # this is back to None
```

### 2.3.11 Using a custom constructor

You can override the automatically generated constructor for your *MultiContainerInterface* subclass.

```
class ContainerHolderWithCustomInit(MultiContainerInterface):

    __clsconf__ = {
        'attr': 'containers',
        'type': Container,
        'add': 'add_container',
    }

    def __init__(self, name, my_containers):
        super().__init__(name=name)
        self.containers = my_containers
        self.add_container(Container('extra_container'))

holder14 = ContainerHolderWithCustomInit('my_name', [obj1, obj2])
holder14.containers # contains the 'extra_container' container
```

## 2.4 TermSet

This is a user guide for interacting with the *TermSet* and *TermSetWrapper* classes. The *TermSet* and *TermSetWrapper* types are experimental and are subject to change in future releases. If you use these types, please provide feedback to the HDMF team so that we can improve the structure and overall capabilities.

### 2.4.1 Introduction

The *TermSet* class provides a way for users to create their own set of terms from brain atlases, species taxonomies, and anatomical, cell, and gene function ontologies.

Users will be able to validate their data and attributes to their own set of terms, ensuring clean data to be used inline with the FAIR principles later on. The *TermSet* class allows for a reusable and sharable pool of metadata to serve as references for any dataset or attribute. The *TermSet* class is used closely with *HERD* to more efficiently map terms to data.

In order to actually use a *TermSet*, users will use the *TermSetWrapper* to wrap data and attributes. The *TermSetWrapper* uses a user-provided *TermSet* to perform validation.

*TermSet* is built upon the resources from LinkML, a modeling language that uses YAML-based schema, giving *TermSet* a standardized structure and a variety of tools to help the user manage their references.

### 2.4.2 How to make a TermSet Schema

Before the user can take advantage of all the wonders within the *TermSet* class, the user needs to create a LinkML schema (YAML) that provides all the permissible term values. Please refer to <https://linkml.io/linkml/intro/tutorial06.html> to learn more about how LinkML structures their schema.

1. The name of the schema is up to the user, e.g., the name could be “Species” if the term set will contain species terms.
2. The prefixes will be the standardized prefix of your source, followed by the URI to the terms. For example, the NCBI Taxonomy is abbreviated as NCBI\_TAXON, and Ensemble is simply Ensemble. As mentioned prior, the URI needs to be to the terms; this is to allow the URI to later be coupled with the source id for the term to create a valid link to the term source page.
3. The schema uses LinkML enumerations to list all the possible terms. To define the all the permissible values, the user can define them manually in the schema, transfer them from a Google spreadsheet, or pull them into the schema dynamically from a LinkML supported source.

For a clear example, please view the [example\\_term\\_set.yaml](#) for this tutorial, which provides a concise example of how a term set schema looks.

---

**Note:** For more information regarding LinkML Enumerations, please refer to <https://linkml.io/linkml/intro/tutorial06.html>.

---

---

**Note:** For more information on how to properly format the Google spreadsheet to be compatible with LinkML, please refer to <https://linkml.io/schemasheets/#examples>.

---

---

**Note:** For more information how to properly format the schema to support LinkML Dynamic Enumerations, please refer to <https://linkml.io/linkml/schemas/enums.html#dynamic-enums>.

---

```

from hdmf.common import DynamicTable, VectorData
import os
import numpy as np

try:
    import linkml_runtime # noqa: F401
except ImportError as e:
    raise ImportError("Please install linkml-runtime to run this example: pip install_
↳linkml-runtime") from e
from hdmf.term_set import TermSet, TermSetWrapper

try:
    dir_path = os.path.dirname(os.path.abspath(__file__))
    yaml_file = os.path.join(dir_path, 'example_term_set.yaml')
    schemasheets_folder = os.path.join(dir_path, 'schemasheets')
    dynamic_schema_path = os.path.join(dir_path, 'example_dynamic_term_set.yaml')
except NameError:
    dir_path = os.path.dirname(os.path.abspath('.'))
    yaml_file = os.path.join(dir_path, 'gallery/example_term_set.yaml')
    schemasheets_folder = os.path.join(dir_path, 'gallery/schemasheets')
    dynamic_schema_path = os.path.join(dir_path, 'gallery/example_dynamic_term_set.yaml')

# Use Schemasheets to create TermSet schema
# -----
# The :py:class:`~hdmf.term_set.TermSet` class builds off of LinkML Schemasheets,
↳allowing users to convert between
# a Google spreadsheet to a complete LinkML schema. Once the user has defined the
↳necessary LinkML metadata within the
# spreadsheet, the spreadsheet needs to be saved as individual tsv files, i.e., one tsv
↳file per spreadsheet tab. Please
# refer to the Schemasheets tutorial link above for more details on the required syntax
↳structure within the sheets.
# Once the tsv files are in a folder, the user simply provides the path to the folder
↳with ``schemasheets_folder``.
termset = TermSet(schemasheets_folder=schemasheets_folder)

# Use Dynamic Enumerations to populate TermSet
# -----
# The :py:class:`~hdmf.term_set.TermSet` class allows user to skip manually defining
↳permissible values, by pulling from
# a LinkML supported source. These sources contain multiple ontologies. A user can
↳select a node from an ontology,
# in which all the elements on the branch, starting from the chosen node, will be used
↳as permissible values.
# Please refer to the LinkML Dynamic Enumeration tutorial for more information on these
↳sources and how to setup Dynamic
# Enumerations within the schema. Once the schema is ready, the user provides a path to
↳the schema and set
# ``dynamic=True``. A new schema, with the populated permissible values, will be created
↳in the same directory.
termset = TermSet(term_schema_path=dynamic_schema_path, dynamic=True)

```

```
ERROR:root:Prefix TEMP not declared: using default
```

```
Downloading cl.db.gz: 0.00B [00:00, ?B/s]
Downloading cl.db.gz: 0%|          | 8.00k/82.9M [00:00<26:57, 53.7kB/s]
Downloading cl.db.gz: 9%|          | 7.31M/82.9M [00:00<00:02, 37.1MB/s]
Downloading cl.db.gz: 27%|         | 22.8M/82.9M [00:00<00:00, 88.1MB/s]
Downloading cl.db.gz: 39%|         | 32.0M/82.9M [00:00<00:00, 60.3MB/s]
Downloading cl.db.gz: 48%|         | 40.0M/82.9M [00:00<00:00, 64.4MB/s]
Downloading cl.db.gz: 66%|        | 54.4M/82.9M [00:00<00:00, 87.3MB/s]
Downloading cl.db.gz: 74%|        | 61.1M/82.9M [00:00<00:00, 82.7MB/s]
Downloading cl.db.gz: 77%|        | 64.1M/82.9M [00:01<00:00, 65.7MB/s]
Downloading cl.db.gz: 87%|        | 72.0M/82.9M [00:01<00:00, 65.6MB/s]
Downloading cl.db.gz: 87%|        | 72.1M/82.9M [00:01<00:00, 41.5MB/s]
Downloading cl.db.gz: 94%|        | 78.3M/82.9M [00:01<00:00, 42.9MB/s]
```

### 2.4.3 Viewing TermSet values

*TermSet* has methods to retrieve terms. The *view\_set* method will return a dictionary of all the terms and the corresponding information for each term. Users can index specific terms from the *TermSet*. LinkML runtime will need to be installed. You can do so by first running `pip install linkml-runtime`.

```
terms = TermSet(term_schema_path=yaml_file)
print(terms.view_set)
```

```
# Retrieve a specific term
terms['Homo sapiens']
```

```
{'Homo sapiens': Term_Info(id='NCBI_TAXON:9606', description='the species is human',
→ meaning='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?mode=Info&id=9606'),
→ 'Mus musculus': Term_Info(id='NCBI_TAXON:10090', description='the species is a house
→ mouse', meaning='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?mode=Info&
→ id=10090'), 'Ursus arctos horribilis': Term_Info(id='NCBI_TAXON:116960', description=
→ 'the species is a grizzly bear', meaning='https://www.ncbi.nlm.nih.gov/Taxonomy/
→ Browser/wwwtax.cgi?mode=Info&id=116960'), 'Myrmecophaga tridactyla': Term_Info(id=
→ 'NCBI_TAXON:71006', description='the species is an anteater', meaning='https://www.
→ ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?mode=Info&id=71006')}
```

```
Term_Info(id='NCBI_TAXON:9606', description='the species is human', meaning='https://www.
→ ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?mode=Info&id=9606')
```

### 2.4.4 Validate Data with TermSetWrapper

*TermSetWrapper* can be wrapped around data. To validate data, the user will set the data to the wrapped data, in which validation must pass for the data object to be created.

```
data = VectorData(
    name='species',
    description='...',
    data=TermSetWrapper(value=['Homo sapiens'], termset=terms)
)
```

### 2.4.5 Validate Compound Data with TermSetWrapper

*TermSetWrapper* can be wrapped around compound data. The user will set the field within the compound data type that is to be validated with the termset.

```
c_data = np.array([('Homo sapiens', 24)], dtype=[('species', 'U50'), ('age', 'i4')])
data = VectorData(
    name='species',
    description='...',
    data=TermSetWrapper(value=c_data, termset=terms, field='species')
)
```

### 2.4.6 Validate Attributes with TermSetWrapper

Similar to wrapping datasets, *TermSetWrapper* can be wrapped around any attribute. To validate attributes, the user will set the attribute to the wrapped value, in which validation must pass for the object to be created.

```
data = VectorData(
    name='species',
    description=TermSetWrapper(value='Homo sapiens', termset=terms),
    data=['Human']
)
```

```
/home/docs/checkouts/readthedocs.org/user_builds/hdmf/checkouts/dev/src/hdmf/container.
↳py:118: UserWarning: Field value already wrapped with TermSetWrapper.
    warn(msg)
```

### 2.4.7 Validate on append with TermSetWrapper

As mentioned prior, when using a *TermSetWrapper*, all new data is validated. This is true for adding new data with append and extend.

```
data = VectorData(
    name='species',
    description='...',
    data=TermSetWrapper(value=['Homo sapiens'], termset=terms)
)

data.append('Ursus arctos horribilis')
data.extend(['Mus musculus', 'Myrmecophaga tridactyla'])
```



## 2.4.8 Validate Data in a DynamicTable

Validating data for *DynamicTable* is determined by which columns were initialized with a *TermSetWrapper*. The data is validated when the columns are created and modified using `DynamicTable.add_row`.

```
col1 = VectorData(
    name='Species_1',
    description='...',
    data=TermSetWrapper(value=['Homo sapiens'], termset=terms),
)
col2 = VectorData(
    name='Species_2',
    description='...',
    data=TermSetWrapper(value=['Mus musculus'], termset=terms),
)
species = DynamicTable(name='species', description='My species', columns=[col1,col2])
```

## 2.4.9 Validate new rows in a DynamicTable with TermSetWrapper

Validating new rows to *DynamicTable* is simple. The `add_row` method will automatically check each column for a *TermSetWrapper*. If a wrapper is being used, then the data will be validated for that column using that column's *TermSet* from the *TermSetWrapper*. If there is invalid data, the row will not be added and the user will be prompted to fix the new data in order to populate the table.

```
species.add_row(Species_1='Mus musculus', Species_2='Mus musculus')
```

## 2.4.10 Validate new columns in a DynamicTable with TermSetWrapper

To add a column that is validated using *TermSetWrapper*, wrap the data in the `add_column` method as if you were making a new instance of *VectorData*.

```
species.add_column(name='Species_3',
                  description='...',
                  data=TermSetWrapper(value=['Ursus arctos horribilis', 'Mus musculus'],
                  ↪ termset=terms),)
```

**Total running time of the script:** (0 minutes 8.087 seconds)

## 2.5 AlignedDynamicTable

This is a user guide to interacting with *AlignedDynamicTable* objects.

## 2.5.1 Introduction

The class *AlignedDynamicTable* represents a column-based table with support for grouping columns by category. *AlignedDynamicTable* inherits from *DynamicTable* and may contain additional *DynamicTable* objects, one per sub-category. All tables must align, i.e., they are required to have the same number of rows. Some key features of *AlignedDynamicTable* are:

- support custom categories, each of which is a *DynamicTable* stored as part of the *AlignedDynamicTable*,
- support interaction with category tables individually as well as treating the *AlignedDynamicTable* as a single large table, and
- because *AlignedDynamicTable* is itself a *DynamicTable* users can:
  - Use *DynamicTableRegion* to reference rows in *AlignedDynamicTable*
  - Add custom columns to the *AlignedDynamicTable*, and
  - Interact with *AlignedDynamicTable* as well as the category (sub-tables) it contains in the same fashion as with *DynamicTable*

### When to use (and not use) AlignedDynamicTable?

*AlignedDynamicTable* is a useful data structure but it is also fairly complex, consisting of multiple *DynamicTable* objects, each of which is itself a complex type composed of many datasets and attributes. In general, if a simpler data structure is sufficient, then consider using those instead. For example, consider using instead:

- *DynamicTable* if a regular table is sufficient.
- A compound dataset via *Table* if all columns of a table are fixed and fast, column-based access is not critical but fast row-based access is.
- Multiple, separate tables if using *AlignedDynamicTable* would lead to duplication of data (i.e., de-normalize data), e.g., by having to replicate values across rows of the table.

Use *AlignedDynamicTable* when:

- When you need to group columns in a *DynamicTable* by category
- Need to avoid name collisions between columns in a *DynamicTable* and creating compound columns is not an option

## 2.5.2 Constructing a table

To create an *AlignedDynamicTable*, call the constructor with:

- name string with the name of the table, and
- description string to describe the table.

```
from hdmf.common import AlignedDynamicTable

customer_table = AlignedDynamicTable(
    name='customers',
    description='an example aligned table',
)
```

## Initializing columns of the primary table

The basic behavior of adding data and initializing *AlignedDynamicTable* is the same as in *DynamicTable*. See the *DynamicTable How-To Guide* for details. E.g., using the `columns` and `colnames` parameters (which are inherited from *DynamicTable*) we can define the columns of the primary table. All columns must have the same length.

```
from hdmf.common import VectorData

col1 = VectorData(
    name='firstname',
    description='Customer first name',
    data=['Peter', 'Emma']
)
col2 = VectorData(
    name='lastname',
    description='Customer last name',
    data=['Williams', 'Brown']
)

customer_table = AlignedDynamicTable(
    name='customer',
    description='an example aligned table',
    columns=[col1, col2]
)
```

## Initializing categories

By specifying the `category_tables` as a list of *DynamicTable* objects we can then directly specify the sub-category tables. Optionally, we can also set the `categories` names of the sub-tables as an array of strings to define the ordering of categories.

```
from hdmf.common import DynamicTable

# create the home_address category table
subcol1 = VectorData(
    name='city',
    description='city',
    data=['Rivercity', 'Mountaincity']
)
subcol2 = VectorData(
    name='street',
    description='street data',
    data=['Amazonstreet', 'Alpinestreet']
)
homeaddress_table = DynamicTable(
    name='home_address',
    description='home address of the customer',
    columns=[subcol1, subcol2]
)

# create the table
customer_table = AlignedDynamicTable(
```

(continues on next page)

(continued from previous page)

```
name='customer',
description='an example aligned table',
columns=[col1, col2],
category_tables=[homeaddress_table, ]
)

# render the table in the online docs
customer_table.to_dataframe()
```

## 2.5.3 Adding more data to the table

We can add rows, columns, and new categories to the table.

### Adding a row

To add a row via `add_row` we can either: 1) provide the row data as a single dict to the `data` parameter or 2) specify a dict for each category and column as keyword arguments. Additional optional arguments include `id` and `enforce_unique_id`.

```
customer_table.add_row(
    firstname='Paul',
    lastname='Smith',
    home_address={'city': 'Bugcity',
                  'street': 'Beestree'}
)

# render the table in the online docs
customer_table.to_dataframe()
```

### Adding a column

To add a columns we use `add_column`.

```
customer_table.add_column(
    name='zipcode',
    description='zip code of the city',
    data=[11111, 22222, 33333], # specify data for the 3 rows in the table
    category='home_address' # use None (or omit) to add columns to the primary table
)

# render the table in the online docs
customer_table.to_dataframe()
```

## Adding a category

To add a new *DynamicTable* as a category, we use *add\_category*.

**Note:** Only regular *DynamicTables* are allowed as category tables. Using an *AlignedDynamicTable* as a category for another *AlignedDynamicTable* is currently not supported.

```
# create a new category DynamicTable for the work address
subcol1 = VectorData(
    name='city',
    description='city',
    data=['Busycity', 'Worktown', 'Labortown']
)
subcol2 = VectorData(
    name='street',
    description='street data',
    data=['Cannery Row', 'Woodwork Avenue', 'Steel Street']
)
subcol3 = VectorData(
    name='zipcode',
    description='zip code of the city',
    data=[33333, 44444, 55555])
workaddress_table = DynamicTable(
    name='work_address',
    description='home address of the customer',
    columns=[subcol1, subcol2, subcol3]
)

# add the category to our AlignedDynamicTable
customer_table.add_category(category=workaddress_table)

# render the table in the online docs
customer_table.to_dataframe()
```

**Note:** Because each category is stored as a separate *DynamicTable* there are no name collisions between the columns of the *home\_address* and *work\_address* tables, so that both can contain matching *city*, *street*, and *zipcode* columns. However, since a category table is a sub-part of the primary table, categories must not have the same name as other columns or other categories in the primary table.

## 2.5.4 Accessing categories, columns, rows, and cells

### Convert to a pandas DataFrame

If we need to access the whole table for analysis, then converting the table to pandas *DataFrame* is a convenient option. To ignore the *id* columns of all category tables we can simply set the *ignore\_category\_ids* parameter.

```
# render the table in the online docs while ignoring the id column of category tables
customer_table.to_dataframe(ignore_category_ids=True)
```

## Accessing categories

```
# Get the list of all categories
_ = customer_table.categories

# Get the DynamicTable object of a particular category
_ = customer_table.get_category(name='home_address')

# Alternatively, we can use normal array slicing to get the category as a pandas
↳ DataFrame.
# NOTE: In contrast to the previous call, the table is here converted to a DataFrame.
_ = customer_table['home_address']
```

## Accessing columns

We can use the standard Python `in` operator to check if a column exists

```
# To check if a column exists in the primary table we only need to specify the column.
↳ name
# or alternatively specify the category as None
_ = 'firstname' in customer_table
_ = (None, 'firstname') in customer_table
# To check if a column exists in a category table we need to specify the category
# and column name as a tuple
_ = ('home_address', 'zipcode') in customer_table
```

We can use standard array slicing to get the *VectorData* object of a column.

```
# To get a column from the primary table we just provide the name.
_ = customer_table['firstname']
# To get a column from a category table we provide both the category name and column name
_ = customer_table['home_address', 'city']
```

## Accessing rows

Accessing rows works much like in *DynamicTable How-To Guide*

```
# Get a single row by index as a DataFrame
customer_table[1]
```

```
# Get a range of rows as a DataFrame
customer_table[0:2]
```

```
# Get a list of rows as a DataFrame
customer_table[[0, 2]]
```

## Accessing cells

To get a set of cells we need to specify the: 1) category, 2) column, and 3) row index when slicing into the table.

When selecting from the primary table we need to specify `None` for the category, followed by the column name and the selection.

```
# Select rows 0:2 from the 'firstname' column in the primary table
customer_table[None, 'firstname', 0:2]
```

```
['Peter', 'Emma']
```

```
# Select rows 1 from the 'firstname' column in the primary table
customer_table[None, 'firstname', 1]
```

```
'Emma'
```

```
# Select rows 0 and 2 from the 'firstname' column in the primary table
customer_table[None, 'firstname', [0, 2]]
```

```
['Peter', 'Paul']
```

```
# Select rows 0:2 from the 'city' column of the 'home_address' category table
customer_table['home_address', 'city', 0:2]
```

```
['Rivercity', 'Mountaincity']
```

## 2.6 HERD: HDMF External Resources Data Structure

This is a user guide to interacting with the [HERD](#) class. The HERD type is experimental and is subject to change in future releases. If you use this type, please provide feedback to the HDMF team so that we can improve the structure and access of data stored with this type for your use cases.

### 2.6.1 Introduction

The [HERD](#) class provides a way to organize and map user terms from their data (keys) to multiple entities from the external resources. A typical use case for external resources is to link data stored in datasets or attributes to ontologies. For example, you may have a dataset `country` storing locations. Using [HERD](#) allows us to link the country names stored in the dataset to an ontology of all countries, enabling more rigid standardization of the data and facilitating data query and introspection.

From a user's perspective, one can think of the [HERD](#) as a simple table, in which each row associates a particular key stored in a particular object (i.e., Attribute or Dataset in a file) with a particular entity (i.e., a term of an online resource). That is, (object, key) refer to parts inside a file and entity refers to an external resource outside the file, and [HERD](#) allows us to link the two. To reduce data redundancy and improve data integrity, [HERD](#) stores this data internally in a collection of interlinked tables.

- [KeyTable](#) where each row describes a [Key](#)
- [FileTable](#) where each row describes a [File](#)
- [EntityTable](#) where each row describes an [Entity](#)

- *EntityKeyTable* where each row describes an *EntityKey*
- *ObjectTable* where each row describes an *Object*
- *ObjectKeyTable* where each row describes an *ObjectKey* pair identifying which keys are used by which objects.

The *HERD* class then provides convenience functions to simplify interaction with these tables, allowing users to treat *HERD* as a single large table as much as possible.

## 2.6.2 Rules to HERD

When using the *HERD* class, there are rules to how users store information in the interlinked tables.

1. Multiple *Key* objects can have the same name. They are disambiguated by the *Object* associated with each, meaning we may have keys with the same name in different objects, but for a particular object all keys must be unique.
2. In order to query specific records, the *HERD* class uses '(file, object\_id, relative\_path, field, key)' as the unique identifier.
3. *Object* can have multiple *Key* objects.
4. Multiple *Object* objects can use the same *Key*.
5. Do not use the private methods to add into the *KeyTable*, *FileTable*, *EntityTable*, *ObjectTable*, *ObjectKeyTable*, *EntityKeyTable* individually.
6. URIs are optional, but highly recommended. If not known, an empty string may be used.
7. An entity ID should be the unique string identifying the entity in the given resource. This may or may not include a string representing the resource and a colon. Use the format provided by the resource. For example, Identifiers.org uses the ID ncbigene:22353 but the NCBI Gene uses the ID 22353 for the same term.
8. In a majority of cases, *Object* objects will have an empty string for 'field'. The *HERD* class supports compound data\_types. In that case, 'field' would be the field of the compound data\_type that has an external reference.
9. In some cases, the attribute that needs an external reference is not a object with a 'data\_type'. The user must then use the nearest object that has a data type to be used as the parent object. When adding an external resource for an object with a data type, users should not provide an attribute. When adding an external resource for an attribute of an object, users need to provide the name of the attribute.
10. The user must provide a *File* or an *Object* that has *File* along the parent hierarchy.

## 2.6.3 Creating an instance of the HERD class

```
from hdmf.common import HERD
from hdmf.common import DynamicTable, VectorData
from hdmf.term_set import TermSet
from hdmf import Container, HERDManager
from hdmf import Data
import numpy as np
import os
# Ignore experimental feature warnings in the tutorial to improve rendering
import warnings
warnings.filterwarnings("ignore", category=UserWarning, message="HERD is experimental")

try:
```

(continues on next page)



(continued from previous page)

```

    dir_path = os.path.dirname(os.path.abspath(__file__))
    yaml_file = os.path.join(dir_path, 'example_term_set.yaml')
except NameError:
    dir_path = os.path.dirname(os.path.abspath('.'))
    yaml_file = os.path.join(dir_path, 'gallery/example_term_set.yaml')

# Class to represent a file
class HERDManagerContainer(Container, HERDManager):
    def __init__(self, **kwargs):
        kwargs['name'] = 'HERDManagerContainer'
        super().__init__(**kwargs)

herd = HERD()
file = HERDManagerContainer(name='file')

```

## 2.6.4 Using the `add_ref` method

`add_ref` is a wrapper function provided by the `HERD` class that simplifies adding data. Using `add_ref` allows us to treat new entries similar to adding a new row to a flat table, with `add_ref` taking care of populating the underlying data structures accordingly.

```

data = Data(name="species", data=['Homo sapiens', 'Mus musculus'])
data.parent = file
herd.add_ref(
    file=file,
    container=data,
    key='Homo sapiens',
    entity_id='NCBI_TAXON:9606',
    entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?id=9606'
)

herd.add_ref(
    file=file,
    container=data,
    key='Mus musculus',
    entity_id='NCBI_TAXON:10090',
    entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?id=10090'
)

```

## 2.6.5 Using the `add_ref` method with an attribute

It is important to keep in mind that when adding and *Object* to the `:py:class:~hdmf.common.resources.ObjectTable`, the parent object identified by `Object.object_id` must be the closest parent to the target object (i.e., `Object.relative_path` must be the shortest possible path and as such cannot contain any objects with a `data_type` and associated `object_id`).

A common example would be with the *DynamicTable* class, which holds *VectorData* objects as columns. If we wanted to add an external reference on a column from a *DynamicTable*, then we would use the column as the object and not the *DynamicTable* (Refer to rule 9).

```
genotypes = DynamicTable(name='genotypes', description='My genotypes')
genotypes.add_column(name='genotype_name', description="Name of genotypes")
genotypes.add_row(id=0, genotype_name='Rorb')
genotypes.parent = file
herd.add_ref(
    file=file,
    container=genotypes,
    attribute='genotype_name',
    key='Rorb',
    entity_id='MGI:1346434',
    entity_uri='http://www.informatics.jax.org/marker/MGI:1343464'
)

# Note: :py:func:`~hdmf.common.resources.HERD.add_ref` internally resolves the object
# to the closest parent, so that ``herd.add_ref(container=genotypes, attribute='genotype_
# ↪name')`` and
# ``herd.add_ref(container=genotypes.genotype_name, attribute=None)`` will ultimately
# ↪both use the ``object_id``
# of the ``genotypes.genotype_name`` :py:class:`~hdmf.common.table.VectorData` column and
# not the object_id of the genotypes table.
```

## 2.6.6 Using the `add_ref` method without the file parameter.

Even though *File* is required to create/add a new reference, the user can omit the file parameter if the *Object* has a file in its parent hierarchy.

```
col1 = VectorData(
    name='Species_Data',
    description='species from NCBI and Ensemble',
    data=['Homo sapiens', 'Ursus arctos horribilis'],
)

# Create a DynamicTable with this column and set the table parent to the file object
# ↪created earlier
species = DynamicTable(name='species', description='My species', columns=[col1])
species.parent = file

herd.add_ref(
    container=species,
    attribute='Species_Data',
    key='Ursus arctos horribilis',
    entity_id='NCBI_TAXON:116960',
```

(continues on next page)

(continued from previous page)

```
entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?mode=Info&id'
)
```

## 2.6.7 Visualize HERD

Users can visualize `~hdmf.common.resources.HERD` as a flattened table or as separate tables.

```
# ~hdmf.common.resources.HERD` as a flattened table
herd.to_dataframe()

# The individual interlinked tables:
herd.files.to_dataframe()
herd.objects.to_dataframe()
herd.entities.to_dataframe()
herd.keys.to_dataframe()
herd.object_keys.to_dataframe()
herd.entity_keys.to_dataframe()
```

## 2.6.8 Using the get\_key method

The `get_key` method will return a `Key` object. In the current version of `HERD`, duplicate keys are allowed; however, each key needs a unique linking Object. In other words, each combination of (file, container, relative\_path, field, key) can exist only once in `HERD`.

```
# The :py:func:`~hdmf.common.resources.HERD.get_key` method will be able to return the
# :py:class:`~hdmf.common.resources.Key` object if the :py:class:`~hdmf.common.resources.
↳Key` object is unique.
genotype_key_object = herd.get_key(key_name='Rorb')

# If the :py:class:`~hdmf.common.resources.Key` object has a duplicate name, then the
↳user will need
# to provide the unique (file, container, relative_path, field, key) combination.
species_key_object = herd.get_key(file=file,
                                   container=species['Species_Data'],
                                   key_name='Ursus arctos horribilis')

# The :py:func:`~hdmf.common.resources.HERD.get_key` also will check the
# :py:class:`~hdmf.common.resources.Object` for a :py:class:`~hdmf.common.resources.File`
↳along the parent hierarchy
# if the file is not provided as in :py:func:`~hdmf.common.resources.HERD.add_ref`
```

### 2.6.9 Using the `add_ref` method with a `key_object`

Multiple *Object* objects can use the same *Key*. To use an existing key when adding new entries into *HERD*, pass the *Key* object instead of the 'key\_name' to the `add_ref` method. If a 'key\_name' is used, a new *Key* will be created.

```
herd.add_ref(
    file=file,
    container=genotypes,
    attribute='genotype_name',
    key=genotype_key_object,
    entity_id='ENSEMBL:ENSG000000198963',
    entity_uri='https://uswest.ensembl.org/Homo_sapiens/Gene/Summary?db=core;
    ↪g=ENSG000000198963'
)
```

### 2.6.10 Using the `get_object_entities`

The `get_object_entities` method allows the user to retrieve all entities and key information associated with an *Object* in the form of a pandas DataFrame.

```
herd.get_object_entities(file=file,
                        container=genotypes['genotype_name'],
                        relative_path='')
```

### 2.6.11 Using the `get_object_type`

The `get_object_entities` method allows the user to retrieve all entities and key information associated with an *Object* in the form of a pandas DataFrame.

```
herd.get_object_type(object_type='Data')
```

### 2.6.12 Special Case: Using `add_ref` with compound data

In most cases, the field is left as an empty string, but if the dataset or attribute is a compound data\_type, then we can use the 'field' value to differentiate the different columns of the dataset. For example, if a dataset has a compound data\_type with columns/fields 'x', 'y', and 'z', and each column/field is associated with different ontologies, then use field='x' to denote that 'x' is using the external reference.

```
# Let's create a new instance of :py:class:`~hdmf.common.resources.HERD`.
herd = HERD()

data = Data(
    name='data_name',
    data=np.array(
        [('Mus musculus', 9, 81.0), ('Homo sapiens', 3, 27.0)],
        dtype=[('species', 'U14'), ('age', 'i4'), ('weight', 'f4')]
    )
)
data.parent = file
```

(continues on next page)

(continued from previous page)

```

herd.add_ref(
    file=file,
    container=data,
    field='species',
    key='Mus musculus',
    entity_id='NCBI_TAXON:txid10090',
    entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?id=10090'
)

```

### 2.6.13 Using add\_ref\_termset

The `add_ref_termset` method allows users to not only validate terms, i.e., keys, but also add references for an entire datasets, rather than single entries as we saw prior with `add_ref`.

```

# :py:func:`~hdmf.common.resources.HERD.add_ref_termset` has many optional fields,
# giving the user a range of control when adding references. Let's see an example.
herd = HERD()
terms = TermSet(term_schema_path=yaml_file)

herd.add_ref_termset(file=file,
                    container=species,
                    attribute='Species_Data',
                    key='Ursus arctos horribilis',
                    termset=terms)

```

### 2.6.14 Using add\_ref\_termset for an entire dataset

As mentioned above, `add_ref_termset` supports iteratively validating and populating `HERD`.

```

# When populating :py:class:`~hdmf.common.resources.HERD`, users may have some terms
# that are not in the :py:class:`~hdmf.term_set.TermSet`. As a result,
# :py:func:`~hdmf.common.resources.HERD.add_ref_termset` will return all of the missing
# terms in a dictionary. It is up to the user to either add these terms to the
# :py:class:`~hdmf.term_set.TermSet` or remove them from the dataset.

herd = HERD()
terms = TermSet(term_schema_path=yaml_file)

herd.add_ref_termset(file=file,
                    container=species,
                    attribute='Species_Data',
                    termset=terms)

```

### 2.6.15 Write HERD

*HERD* is written as a zip file of the individual tables written to tsv. The user provides the path, which contains the name of the file.

```
herd.to_zip(path='./HERD.zip')
```

### 2.6.16 Read HERD

Users can read *HERD* from the zip file by providing the path to the file itself.

```
er_read = HERD.from_zip(path='./HERD.zip')
os.remove('./HERD.zip')
```

## 2.7 DynamicTable How-To Guide

This is a user guide to interacting with `DynamicTable` objects.

### 2.7.1 Introduction

The *DynamicTable* class represents a column-based table to which you can add custom columns. It consists of a name, a description, a list of row IDs, and a list of columns. Columns are represented by objects of the class *VectorData*, including subclasses of *VectorData*, such as *VectorIndex*, and *DynamicTableRegion*.

### 2.7.2 Constructing a table

To create a *DynamicTable*, call the constructor for *DynamicTable* with a string name and string description. Specifying the arguments with keywords is recommended.

```
from hdmf.common import DynamicTable

table = DynamicTable(
    name='my_table',
    description='an example table',
)
```

### 2.7.3 Initializing columns

You can create a *DynamicTable* with particular columns by passing a list or tuple of *VectorData* objects for the columns argument in the constructor.

If the *VectorData* objects contain data values, then each *VectorData* object must contain the same number of rows as each other. A list of row IDs may be passed into the *DynamicTable* constructor using the `id` argument. If IDs are passed in, there should be the same number of rows as the column data. If IDs are not passed in, then the IDs will be set to `range(len(column_data))` by default.

```

from hdmf.common import VectorData, VectorIndex

col1 = VectorData(
    name='col1',
    description='column #1',
    data=[1, 2],
)
col2 = VectorData(
    name='col2',
    description='column #2',
    data=['a', 'b'],
)

# this table will have two rows with ids 0 and 1
table = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col1, col2],
)

# this table will have two rows with ids 0 and 1
table_set_ids = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col1, col2],
    id=[0, 1],
)

```

If a list of integers is passed to `id`, *DynamicTable* automatically creates an *ElementIdentifiers* object, which is the data type that stores row IDs. The above command is equivalent to:

```

from hdmf.common.table import ElementIdentifiers

table_set_ids = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col1, col2],
    id=ElementIdentifiers(name='id', data=[0, 1]),
)

```

## 2.7.4 Adding rows

You can also add rows to a *DynamicTable* using *DynamicTable.add\_row*. A keyword argument for every column in the table must be supplied. You may also supply an optional row ID.

```

table.add_row(
    col1=3,
    col2='c',
    id=2,
)

```

**Note:** If no ID is supplied, the row ID is automatically set to the number of rows of the table prior to adding the

new row. This can result in duplicate IDs. In general, IDs should be unique, but this is not enforced by default. Pass `enforce_unique_id=True` to `DynamicTable.add_row` to raise an error if the ID is set to an existing ID value.

---

```
# this row will have ID 3 by default
table.add_row(
    col1=4,
    col2='d',
)
```

## 2.7.5 Adding columns

You can add columns to a `DynamicTable` using `DynamicTable.add_column`. If the table already has rows, then the `data` argument must be supplied as a list of values, one for each row already in the table.

```
table.add_column(
    name='col3',
    description='column #3',
    data=[True, True, False, True], # specify data for the 4 rows in the table
)
```

## 2.7.6 Enumerated (categorical) data

`EnumData` is a special type of column for storing an enumerated data type. This way each unique value is stored once, and the data references those values by index. Using this method is more efficient than storing a single value many times, and has the advantage of communicating to downstream tools that the data is categorical in nature.

**Warning:** `EnumData` is currently an experimental feature and as such should not be used for production use.

```
from hdmf.common.table import EnumData
import warnings
warnings.filterwarnings(action="ignore", message="EnumData is experimental")

# this column has a length of 5, not 3. the first row has value "aa"
enum_col = EnumData(
    name='cell_type',
    description='this column holds categorical variables',
    data=[0, 1, 2, 1, 0],
    elements=['aa', 'bb', 'cc']
)

my_table = DynamicTable(
    name='my_table',
    description='an example table',
    columns=[enum_col],
)
```



### 2.7.7 Ragged array columns

A table column with a different number of elements for each row is called a “ragged array column”. To initialize a *DynamicTable* with a ragged array column, pass both the *VectorIndex* and its target *VectorData* in for the `columns` argument in the constructor. For instance, the following code creates a column called `col1` where the first cell is `['1a', '1b', '1c']` and the second cell is `['2a']`.

```
col1 = VectorData(
    name='col1',
    description='column #1',
    data=['1a', '1b', '1c', '2a'],
)
# the 3 signifies that elements 0 to 3 (exclusive) of the target column belong to the
↪first row
# the 4 signifies that elements 3 to 4 (exclusive) of the target column belong to the
↪second row
col1_ind = VectorIndex(
    name='col1_index',
    target=col1,
    data=[3, 4],
)

table_ragged_col = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col1, col1_ind],
)
```

**Note:** By convention, the name of the *VectorIndex* should be the name of the target column with the added suffix “\_index”.

`VectorIndex.data` provides the indices for how to break `VectorData.data` into cells

You can add an empty ragged array column to an existing *DynamicTable* by specifying `index=True` to *DynamicTable.add\_column*. This method only works if run before any rows have been added to the table.

```
new_table = DynamicTable(
    name='my_table',
    description='an example table',
)

new_table.add_column(
    name='col4',
    description='column #4',
    index=True,
)
```

If the table already contains data, you must specify the new column values for the existing rows using the `data` argument and you must specify the end indices of the `data` argument that correspond to each row as a list/tuple/array of values for the `index` argument.

```
table.add_column( # <-- this table already has 4 rows
    name='col4',
```

(continues on next page)

(continued from previous page)

```

description='column #4',
data=[1, 0, -1, 0, -1, 1, 1, -1],
index=[3, 4, 6, 8], # specify the end indices (exclusive) of data for each row
)

```

Alternatively we may also define the ragged array data as a nested list and use the `index` argument to indicate the number of levels. In this case, the `add_column` function will automatically flatten the data array and compute the corresponding index vectors.

```

table.add_column( # <-- this table already has 4 rows
    name='col5',
    description='column #5',
    data=[[1, ], [2, 2]],      # row 1
          [[3, 3], ],        # row 2
          [[4, ], [5, 5]],    # row 3
          [[6, 6], [7, 7, 7]] # row 4
    index=2 # number of levels in the ragged array
)
# Show that the ragged array was converted to flat VectorData with a double VectorIndex
print("Flattened data: %s" % str(table.col5.data))
print("Level 1 index: %s" % str(table.col5_index.data))
print("Level 2 index: %s" % str(table.col5_index_index.data))

```

```

Flattened data: [1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 7, 7, 7]
Level 1 index: [1, 3, 5, 6, 8, 10, 13]
Level 2 index: [2, 3, 5, 7]

```

## 2.7.8 Referencing rows of other tables

You can create a column that references rows of another table by adding a `DynamicTableRegion` object as a column of your `DynamicTable`. This is analogous to a foreign key in a relational database.

```

from hdmf.common.table import DynamicTableRegion

dtr_col = DynamicTableRegion(
    name='table1_ref',
    description='references rows of earlier table',
    data=[0, 1, 0, 0], # refers to row indices of the 'table' variable
    table=table
)

data_col = VectorData(
    name='col2',
    description='column #2',
    data=['a', 'a', 'a', 'b'],
)

table2 = DynamicTable(
    name='my_table',
    description='an example table',

```

(continues on next page)

(continued from previous page)

```
columns=[dtr_col, data_col],
)
```

Here, the data of `dtr_col` maps to rows of `table` (0-indexed).

**Note:** The data values of *DynamicTableRegion* map to the row index, not the row ID, though if you are using default IDs, these values will be the same.

Reference more than one row of another table with a *DynamicTableRegion* indexed by a *VectorIndex*.

```
indexed_dtr_col = DynamicTableRegion(
    name='table1_ref2',
    description='references multiple rows of earlier table',
    data=[0, 0, 1, 1, 0, 0, 1],
    table=table
)

# row 0 refers to rows [0, 0], row 1 refers to rows [1], row 2 refers to rows [1, 0],
# row 3 refers to rows [0, 1] of
# the "table" variable
dtr_idx = VectorIndex(
    name='table1_ref2_index',
    target=indexed_dtr_col,
    data=[2, 3, 5, 7],
)

table3 = DynamicTable(
    name='my_table',
    description='an example table',
    columns=[dtr_idx, indexed_dtr_col],
)
```

## 2.7.9 Setting the target table of a DynamicTableRegion column of a DynamicTable

A subclass of *DynamicTable* might have a pre-defined *DynamicTableRegion* column. To write this column correctly, the “table” attribute of the column must be set so that users know to what table the row index values reference. Because the target table could be any table, the “table” attribute must be set explicitly. There are three ways to do so. First, you can use the `target_tables` argument of the *DynamicTable* constructor as shown below. This argument is a dictionary mapping the name of the *DynamicTableRegion* column to the target table. Secondly, the target table can be set after the *DynamicTable* has been initialized using `my_table.my_column.table = other_table`. Finally, you can create the *DynamicTableRegion* column and pass the `table` attribute to *DynamicTableRegion.\_\_init\_\_* and then pass the column to *DynamicTable.\_\_init\_\_* using the `columns` argument. However, this approach is not recommended for columns defined in the schema, because it is up to the user to ensure that the column is created in accordance with the schema.

```
class SubTable(DynamicTable):
    __columns__ = (
        {'name': 'dtr', 'description': 'required region', 'required': True, 'table':
        True},
    )
```

(continues on next page)

(continued from previous page)

```

referenced_table = DynamicTable(
    name='referenced_table',
    description='an example table',
)

sub_table = SubTable(
    name='sub_table',
    description='an example table',
    target_tables={'dtr': referenced_table},
)
# now the target table of the DynamicTableRegion column 'dtr' is set to `referenced_table`

```

### 2.7.10 Creating an expandable table

When using the default HDF5 backend, each column of these tables is an HDF5 Dataset, which by default are set in size. This means that once a file is written, it is not possible to add a new row. If you want to be able to save this file, load it, and add more rows to the table, you will need to set this up when you create the *DynamicTable*. You do this by wrapping the data with *H5DataIO* and the argument `maxshape=(None,)`.

```

from hdmf.backends.hdf5.h5_utils import H5DataIO

col1 = VectorData(
    name='expandable_col1',
    description='column #1',
    data=H5DataIO(data=[1, 2], maxshape=(None,)),
)
col2 = VectorData(
    name='expandable_col2',
    description='column #2',
    data=H5DataIO(data=['a', 'b'], maxshape=(None,)),
)

# don't forget to wrap the row IDs too!
ids = ElementIdentifiers(
    name='id',
    data=H5DataIO(data=[0, 1], maxshape=(None,)),
)

expandable_table = DynamicTable(
    name='expandable_table',
    description='an example table that can be expanded after being saved to a file',
    columns=[col1, col2],
    id=ids,
)

```

Now you can write the file, read it back, and run `expandable_table.add_row()`. In this example, we are setting `maxshape` to `(None,)`, which means this is a 1-dimensional matrix that can expand indefinitely along its single dimension. You could also use an integer in place of `None`. For instance, `maxshape=(8,)` would allow the column to grow up to a length of 8. Whichever `maxshape` you choose, it should be the same for all *VectorData* and *ElementIdentifiers* objects in the *DynamicTable*, since they must always be the same length. The default *ElementIdentifiers* automatically generated when you pass a list of integers to the `id` argument of the

`DynamicTable` constructor is not expandable, so do not forget to create a `ElementIdentifiers` object, and wrap that data as well. If any of the columns are indexed, the data argument of `VectorIndex` will also need to be wrapped with `H5DataIO`.

### 2.7.11 Converting the table to a pandas DataFrame

`pandas` is a popular data analysis tool, especially for working with tabular data. You can convert your `DynamicTable` to a `DataFrame` using `DynamicTable.to_dataframe`. Accessing the table as a `DataFrame` provides you with powerful, standard methods for indexing, selecting, and querying tabular data from `pandas`. This is the recommended method of reading data from your table. See also the [pandas indexing documentation](#). Printing a `DynamicTable` as a `DataFrame` or displaying the `DataFrame` in Jupyter shows a more intuitive tabular representation of the data than printing the `DynamicTable` object.

```
df = table.to_dataframe()
```

**Note:** Changes to the `DataFrame` will not be saved in the `DynamicTable`.

### 2.7.12 Converting the table from a pandas DataFrame

If your data is already in a `DataFrame`, you can convert the `DataFrame` to a `DynamicTable` using the class method `DynamicTable.from_dataframe`.

```
table_from_df = DynamicTable.from_dataframe(
    name='my_table',
    df=df,
)
```

### 2.7.13 Accessing elements

To access an element in the *i*-th row in the column with name “col\_name” in a `DynamicTable`, use square brackets notation: `table[i, col_name]`. You can also use a tuple of row index and column name within the square brackets.

```
table[0, 'col1'] # returns 1
table[(0, 'col1')] # returns 1
```

```
1
```

If the column is a ragged array, instead of a single value being returned, a list of values for that element is returned.

```
table[0, 'col4'] # returns [1, 0, -1]
```

```
[1, 0, -1]
```

Standard Python and numpy slicing can be used for the row index.

```
import numpy as np

table[:2, 'col1'] # get a list of elements from the first two rows at column 'col1'
```

(continues on next page)

(continued from previous page)

```

table[0:3:2, 'col1'] # get a list of elements from rows 0 to 3 (exclusive) in steps of 2 at column 'col1'
table[3::-1, 'col1'] # get a list of elements from rows 3 to 0 in reverse order at column 'col1'

# the following are equivalent to table[0:3:2, 'col1']
table[slice(0, 3, 2), 'col1']
table[np.s_[0:3:2], 'col1']
table[[0, 2], 'col1']
table[np.array([0, 2]), 'col1']

```

```
[1, 3]
```

If the column is a ragged array, instead of a list of row values being returned, a list of list elements for the selected rows is returned.

```
table[:, 'col4'] # returns [[1, 0, -1], [0]]
```

```
[[1, 0, -1], [0]]
```

**Note:** You cannot supply a list/tuple for the column name. For this kind of access, first convert the *DynamicTable* to a *DataFrame*.

## 2.7.14 Accessing columns

To access all the values in a column, use square brackets with a colon for the row index: `table[:, col_name]`. If the column is a ragged array, a list of list elements is returned.

```

table[:, 'col1'] # returns [1, 2, 3, 4]
table[:, 'col4'] # returns [[1, 0, -1], [0], [-1, 1], [1, -1]]

```

```
[[1, 0, -1], [0], [-1, 1], [1, -1]]
```

## 2.7.15 Accessing rows

To access the *i*-th row in a *DynamicTable*, returned as a *DataFrame*, use the syntax `table[i]`. Standard Python and numpy slicing can be used for the row index.

```

table[0] # get the 0th row of the table as a DataFrame
table[:2] # get the first two rows
table[0:3:2] # get rows 0 to 3 (exclusive) in steps of 2
table[3::-1] # get rows 3 to 0 in reverse order

# the following are equivalent to table[0:3:2]
table[slice(0, 3, 2)]
table[np.s_[0:3:2]]
table[[0, 2]]
table[np.array([0, 2])]

```

---

**Note:** The syntax `table[i]` returns the *i*-th row, NOT the row with ID of *i*.

---

## 2.7.16 Iterating over rows

To iterate over the rows of a *DynamicTable*, first convert the *DynamicTable* to a *DataFrame* using *DynamicTable.to\_dataframe*. For more information on iterating over a *DataFrame*, see [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/basics.html#iteration](https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#iteration)

```
df = table.to_dataframe()
for row in df.itertuples():
    print(row)
```

```
Pandas(Index=0, col1=1, col2='a', col3=True, col4=[1, 0, -1], col5=[[1], [2, 2]])
Pandas(Index=1, col1=2, col2='b', col3=True, col4=[0], col5=[[3, 3]])
Pandas(Index=2, col1=3, col2='c', col3=False, col4=[-1, 1], col5=[[4], [5, 5]])
Pandas(Index=3, col1=4, col2='d', col3=True, col4=[1, -1], col5=[[6, 6], [7, 7, 7]])
```

## 2.7.17 Accessing the column data types

To access the *VectorData* or *VectorIndex* object representing a column, you can use three different methods. Use the column name in square brackets, e.g., `table[col_name]`, use the *DynamicTable.get* method, or use the column name as an attribute, e.g., `table.col_name`.

```
table['col1']
table.get('col1') # equivalent to table['col1'] except this returns None if 'col1' is not
                  ↳ found
table.get('col1', default=0) # you can change the default return value
table.col1
```

```
<hdmf.common.table.VectorData object at 0x7f8aad29dbe0>
```

---

**Note:** Using the column name as an attribute does NOT work if the column name is the same as a non-column name attribute or method of the *DynamicTable* class, e.g., `name`, `description`, `object_id`, `parent`, `modified`.

---

If the column is a ragged array, then the methods above will return the *VectorIndex* associated with the ragged array.

```
table['col4']
table.get('col4') # equivalent to table['col4'] except this returns None if 'col4' is not
                  ↳ found
table.get('col4', default=0) # you can change the default return value
```

```
<hdmf.common.table.VectorIndex object at 0x7f8aad29d3a0>
```

---

**Note:** The attribute syntax `table.col_name` currently returns the *VectorData* instead of the *VectorIndex* for a ragged array. This is a known issue and will be fixed in a future version of HDMF.

---

## 2.7.18 Accessing elements from column data types

Standard Python and numpy slicing can be used on the *VectorData* or *VectorIndex* objects to access elements from column data. If the column is a ragged array, then instead of a list of row values being returned, a list of list elements for the selected rows is returned.

```
table['col1'][0] # get the 0th element from column 'col1'
table['col1'][:2] # get a list of the 0th and 1st elements
table['col1'][0:3:2] # get a list of the 0th to 3rd (exclusive) elements in steps of 2
table['col1'][3::-1] # get a list of the 3rd to 0th elements in reverse order

# the following are equivalent to table['col1'][0:3:2]
table['col1'][slice(0, 3, 2)]
table['col1'][np.s_[0:3:2]]
table['col1'][[0, 2]]
table['col1'][np.array([0, 2])]

# this slicing and indexing works for ragged array columns as well
table['col4'][:2] # get a list of the 0th and 1st list elements
```

```
[[1, 0, -1], [0]]
```

---

**Note:** The syntax `table[col_name][i]` is equivalent to `table[i, col_name]`.

---

## 2.7.19 Multi-dimensional columns

A column can be represented as a multi-dimensional rectangular array or a list of lists, each containing the same number of elements.

```
col5 = VectorData(
    name='col5',
    description='column #5',
    data=[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']],
)
```

## 2.7.20 Ragged multi-dimensional columns

Each element within a column can be an n-dimensional array or list or lists. This is true for ragged array columns as well.

```
col6 = VectorData(
    name='col6',
    description='column #6',
    data=[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']],
)
col6_ind = VectorIndex(
    name='col6_index',
    target=col6,
    data=[2, 3],
)
```



### 2.7.21 Nested ragged array columns

In the example above, the ragged array column above has two rows. The first row has two elements, where each element has 3 sub-elements. This can be thought of as a 2x3 array. The second row has one element with 3 sub-elements, or a 1x3 array. This works only if the data for col5 is a rectangular array, that is, each row element contains the same number of sub-elements. If each row element does not contain the same number of sub-elements, then a nested ragged array approach must be used instead.

A [VectorIndex](#) object can index another [VectorIndex](#) object. For example, the first row of a table might be a 2x3 array, the second row might be a 3x2 array, and the third row might be a 1x1 array. This cannot be represented by a singly indexed column, but can be represented by a nested ragged array column.

```
col7 = VectorData(
    name='col7',
    description='column #6',
    data=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm'],
)
col7_ind = VectorIndex(
    name='col7_index',
    target=col7,
    data=[3, 6, 8, 10, 12, 13],
)
col7_ind_ind = VectorIndex(
    name='col7_index_index',
    target=col7_ind,
    data=[2, 5, 6],
)

# all indices must be added to the table
table_double_ragged_col = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col7, col7_ind, col7_ind_ind],
)
```

Access the first row using the same syntax as before, except now a list of lists is returned. You can then index the resulting list of lists to access the individual elements.

```
table_double_ragged_col[0, 'col7'] # returns [['a', 'b', 'c'], ['d', 'e', 'f']]
table_double_ragged_col['col7'][0] # same as line above
```

```
[['a', 'b', 'c'], ['d', 'e', 'f']]
```

Accessing the column named 'col7' using square bracket notation will return the top-level [VectorIndex](#) for the column. Accessing the column named 'col7' using dot notation will return the [VectorData](#) object

```
table_double_ragged_col['col7'] # returns col7_ind_ind
table_double_ragged_col.col7 # returns the col7 VectorData object
```

```
<hdmf.common.table.VectorData object at 0x7f8aad29d0d0>
```

## 2.7.22 Accessing data from a DynamicTable that contain references to rows of other DynamicTable objects

By default, when `DynamicTable.__getitem__` and `DynamicTable.get` are supplied with an int, list of ints, numpy array, or a slice representing rows to return, a pandas `DataFrame` is returned. If the `DynamicTable` contains a `DynamicTableRegion` column that references rows of other `DynamicTable` objects, then by default, the `DynamicTable.__getitem__` and `DynamicTable.get` methods will return row indices of the referenced table, and not the contents of the referenced table. To return the contents of the referenced table as a nested `DataFrame` containing only the referenced rows, use `DynamicTable.get` with `index=False`.

```
# create a new table of users
users_table = DynamicTable(
    name='users',
    description='a table containing data/metadata about users, one user per row',
)

# add simple columns to this table
users_table.add_column(
    name='first_name',
    description='the first name of the user',
)
users_table.add_column(
    name='last_name',
    description='the last name of the user',
)

# create a new table of addresses to reference
addresses_table = DynamicTable(
    name='addresses',
    description='a table containing data/metadata about addresses, one address per row',
)
addresses_table.add_column(
    name='street_address',
    description='the street number and address',
)
addresses_table.add_column(
    name='city',
    description='the city of the address',
)

# add rows to the addresses table
addresses_table.add_row(
    street_address='123 Main St',
    city='Springfield'
)
addresses_table.add_row(
    street_address='45 British Way',
    city='London'
)

# add a column to the users table that references rows of the addresses table
users_table.add_column(
    name='address',
```

(continues on next page)

(continued from previous page)

```

        description='the address of the user',
        table=addresses_table
    )

    # add rows to the users table
    users_table.add_row(
        first_name='Grace',
        last_name='Hopper',
        address=0 # <-- row index of the address table
    )

    users_table.add_row(
        first_name='Alan',
        last_name='Turing',
        address=1 # <-- row index of the address table
    )

    # get the first row of the users table
    users_table.get(0)

```

```

# get the first row of the users table with a nested dataframe
users_table.get(0, index=False)

```

```

# get the first two rows of the users table
users_table.get([0, 1])

```

```

# get the first two rows of the users table with nested dataframes
# of the addresses table in the address column
users_table.get([0, 1], index=False)

```

**Note:** You can also get rows from a *DynamicTable* as a list of lists where the i-th nested list contains the values for the i-th row. This method is generally not recommended.

### 2.7.23 Displaying the contents of a table with references to another table

Earlier, we converted a *DynamicTable* to a *DataFrame* using *DynamicTable.to\_dataframe* and printed the *DataFrame* to see its contents. This also works when the *DynamicTable* contains a column that references another table. However, the entries for this column for each row will be printed as a nested *DataFrame*. This can be difficult to read, so to view only the row indices of the referenced table, pass `index=True` to *DynamicTable.to\_dataframe*.

```

users_df = users_table.to_dataframe(index=True)
users_df

```

You can then access the referenced table using the `table` attribute of the column object. This is useful when reading a table from a file where you may not have a variable to access the referenced table.

First, use *DynamicTable.\_\_getitem\_\_* (square brackets notation) to get the *DynamicTableRegion* object representing the column. Then access its `table` attribute to get the addresses table and convert the table to a *DataFrame*.

```
address_column = users_table['address']
read_addresses_table = address_column.table
addresses_df = read_addresses_table.to_dataframe()
```

Get the addresses corresponding to the rows of the users table:

```
address_indices = users_df['address'] # pandas Series of row indices into the addresses_
↳ table
addresses_df.iloc[address_indices] # use .iloc because these are row indices not ID_
↳ values
```

---

**Note:** The indices returned by `users_df['address']` are row indices and not the ID values of the table. However, if you are using default IDs, these values will be the same.

---

## 2.7.24 Creating custom DynamicTable subclasses

TODO

Defining `__columns__`

TODO

## INTRODUCTION

HDMF provides a high-level Python API for specifying, reading, writing and manipulating hierarchical object data. This section provides a broad overview of the software architecture of HDMF (see Section *Software Architecture*) and its functionality.



## SOFTWARE ARCHITECTURE

The main goal of HDMF is to enable users and developers to efficiently interact with the hierarchical object data. The following figures provide an overview of the high-level architecture of HDMF and functionality of the various components.

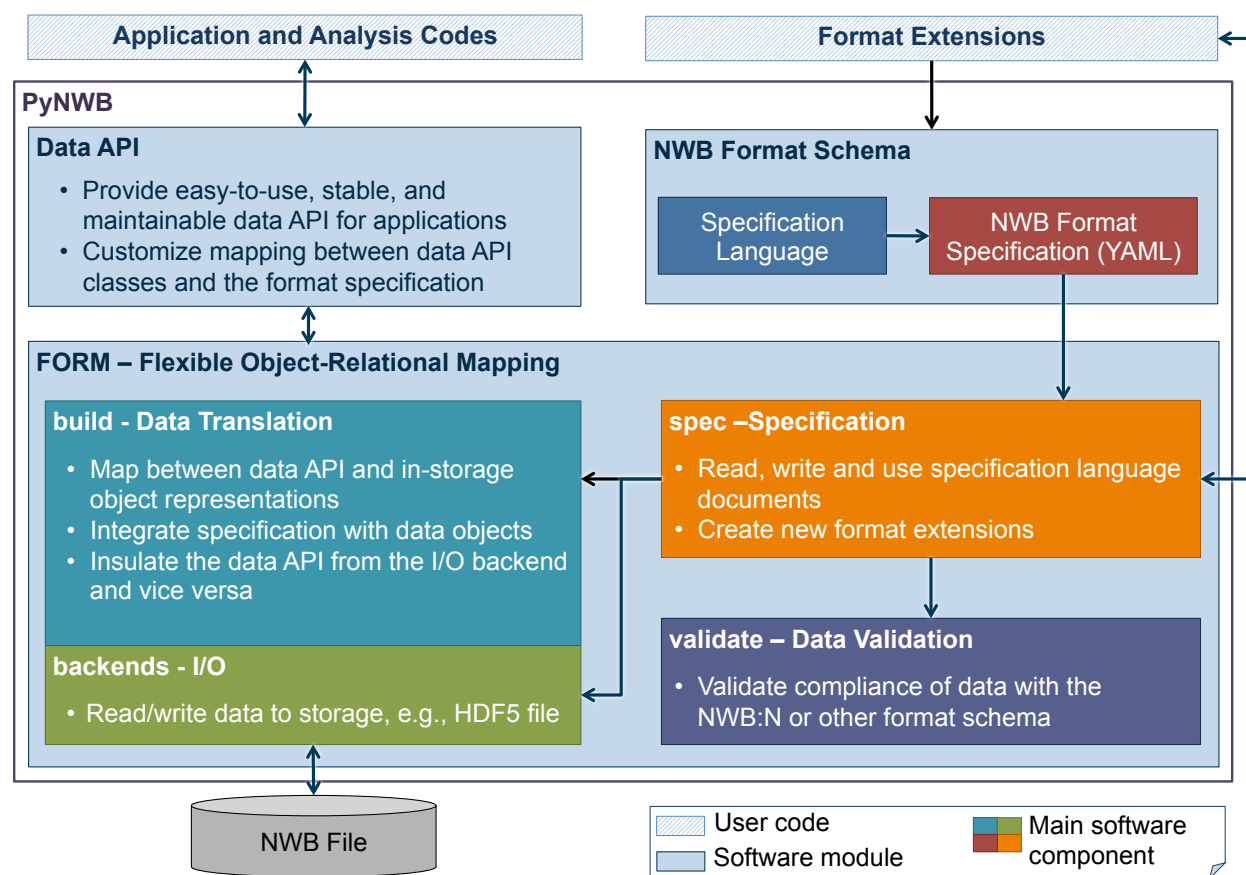


Fig. 1: Overview of the high-level software architecture of HDMF (click to enlarge).

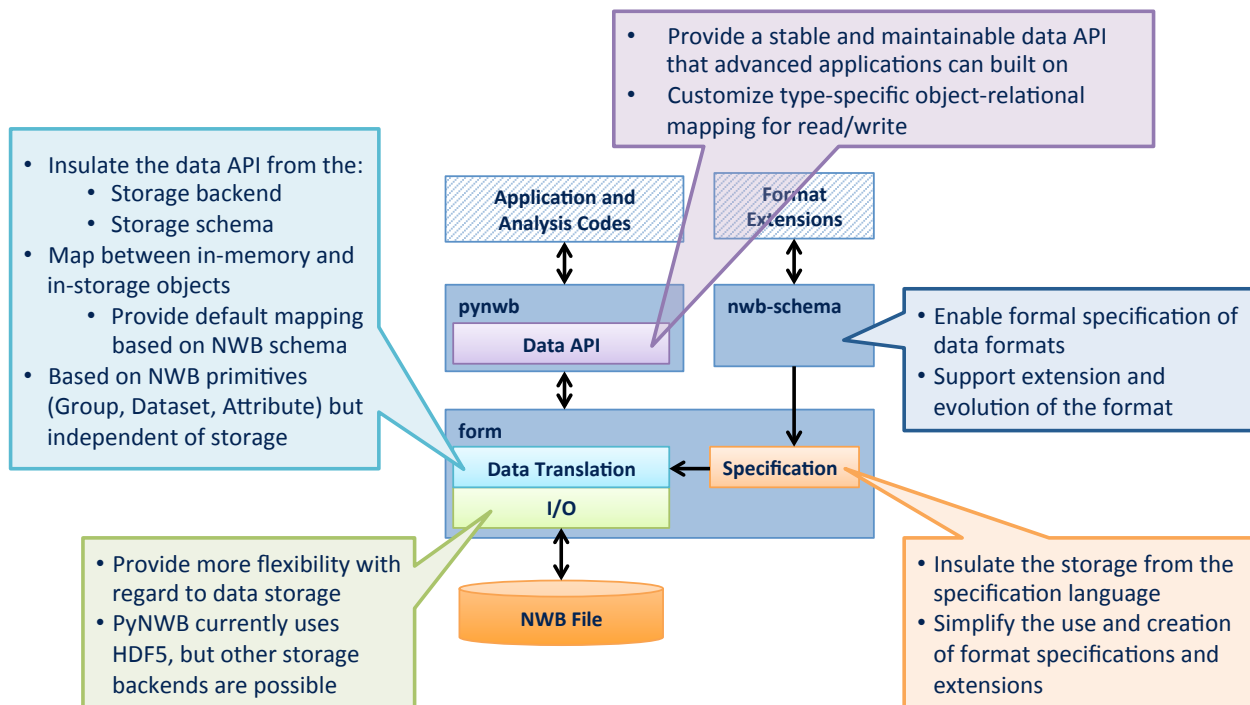


Fig. 2: We choose a modular design for HDMF to enable flexibility and separate the various levels of standardizing hierarchical data (click to enlarge).



## 4.1 Main Concepts

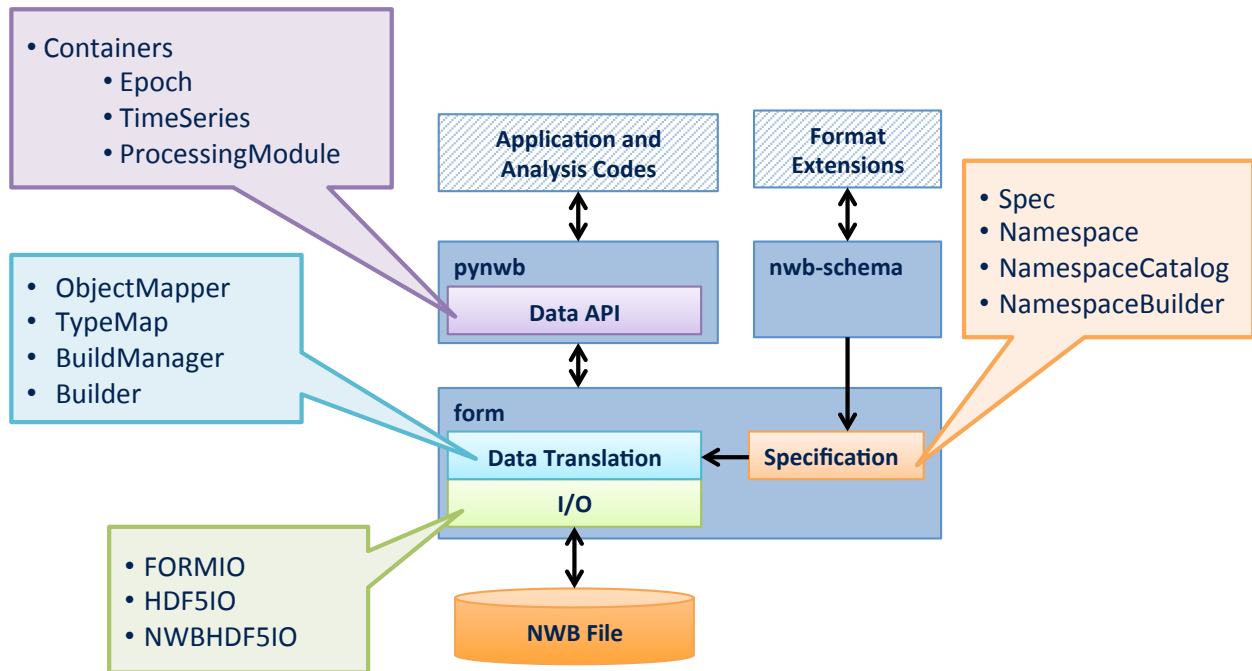


Fig. 3: Overview of the main concepts/classes in HDMF and their location in the overall software architecture (click to enlarge).

### 4.1.1 Container

- In memory objects
- Interface for (most) applications
- Similar to a table row
- HDMF does not provide these. They are left for standards developers to define how users interact with data.
- There are two Container base classes:
  - `Container` - represents a collection of objects
  - `Data` - represents data
- **Main Module:** `hdmf.container`

### 4.1.2 Builder

- Intermediary objects for I/O
- Interface for I/O
- Backend readers and writers must return and accept these
- There are different kinds of builders for different base types:
  - *GroupBuilder* - represents a collection of objects
  - *DatasetBuilder* - represents data
  - *LinkBuilder* - represents soft-links
  - *RegionBuilder* - represents a slice into data (Subclass of *DatasetBuilder*)
- **Main Module:** *hdmf.build.builders*

### 4.1.3 Spec

- Interact with format specifications
- Data structures to specify data types and what said types consist of
- Python representation for YAML specifications
- Interface for writing extensions or custom specification
- There are several main specification classes:
  - *AttributeSpec* - specification for metadata
  - *GroupSpec* - specification for a collection of objects (i.e. subgroups, datasets, link)
  - *DatasetSpec* - specification for dataset (like and n-dimensional array). Specifies data type, dimensions, etc.
  - *LinkSpec* - specification for link (like a POSIX soft link)
  - *RefSpec* - specification for references (References are like links, but stored as data)
  - *DtypeSpec* - specification for compound data types. Used to build complex data type specification, e.g., to define tables (used only in *DatasetSpec* and correspondingly *DatasetSpec*)
- **Main Modules:** *hdmf.spec*

---

**Note:** A *data\_type* defines a reusable type in a format specification that can be referenced and used elsewhere in other specifications. The specification of the standard is basically a collection of *data\_types*,

- *data\_type\_inc* is used to include an existing type and
- *data\_type\_def* is used to define a new type

i.e, if both keys are defined then we create a new type that uses/inherits an existing type as a base.

---

#### 4.1.4 ObjectMapper

- Maintains the mapping between *Container* attributes and *Spec* components
- Provides a way of converting between *Container* and *Builder*, while leaving standards developers with the flexibility of presenting data to users in a user-friendly manner, while storing data in an efficient manner
- ObjectMappers are constructed using a *Spec*
- Ideally, one ObjectMapper for each data type
- Things an ObjectMapper should do:
  - Given a *Builder*, return a Container representation
  - Given a *Container*, return a Builder representation
- **Main Module:** `hdmf.build.objectmapper`

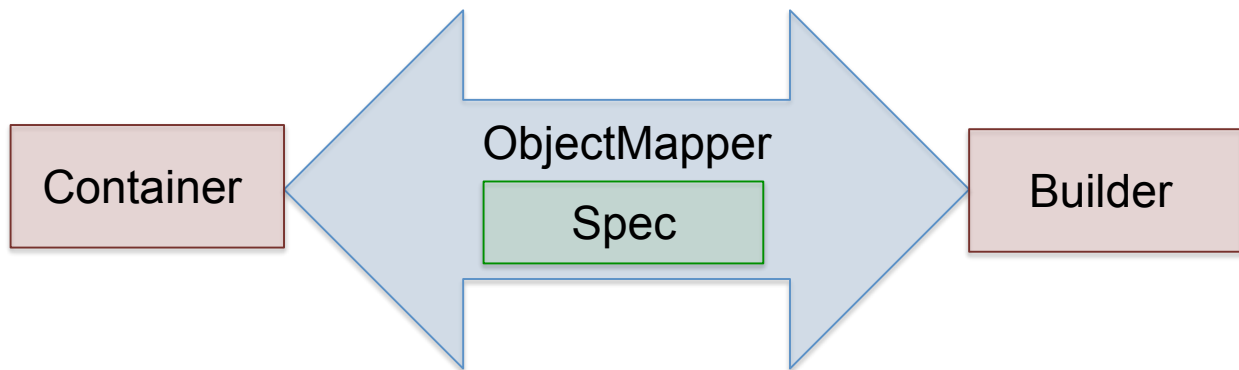


Fig. 4: Relationship between *Container*, *Builder*, *ObjectMapper*, and *Spec*

## 4.2 Additional Concepts

### 4.2.1 Namespace, NamespaceCatalog, NamespaceBuilder

- **Namespace**
  - A namespace for specifications
  - Necessary for making standards extensions and standard core specification
  - Contains basic info about who created extensions
- *NamespaceCatalog* – A class for managing namespaces
- *NamespaceBuilder* – A utility for building extensions

## 4.2.2 TypeMap

- Map between data types, Container classes (i.e. a Python class object) and corresponding ObjectMapper classes
- Constructed from a NamespaceCatalog
- Things a TypeMap does:
  - Given a data\_type, return the associated Container class
  - Given a Container class, return the associated ObjectMapper
- HDMF has one of these classes:
  - the base class (i.e. *TypeMap*)
- TypeMaps can be merged, which is useful when combining extensions

## 4.2.3 BuildManager

- Responsible for memoizing *Builder* and *Container*
- Constructed from a *TypeMap*
- HDMF only has one of these: *hdmf.build.manager.BuildManager*

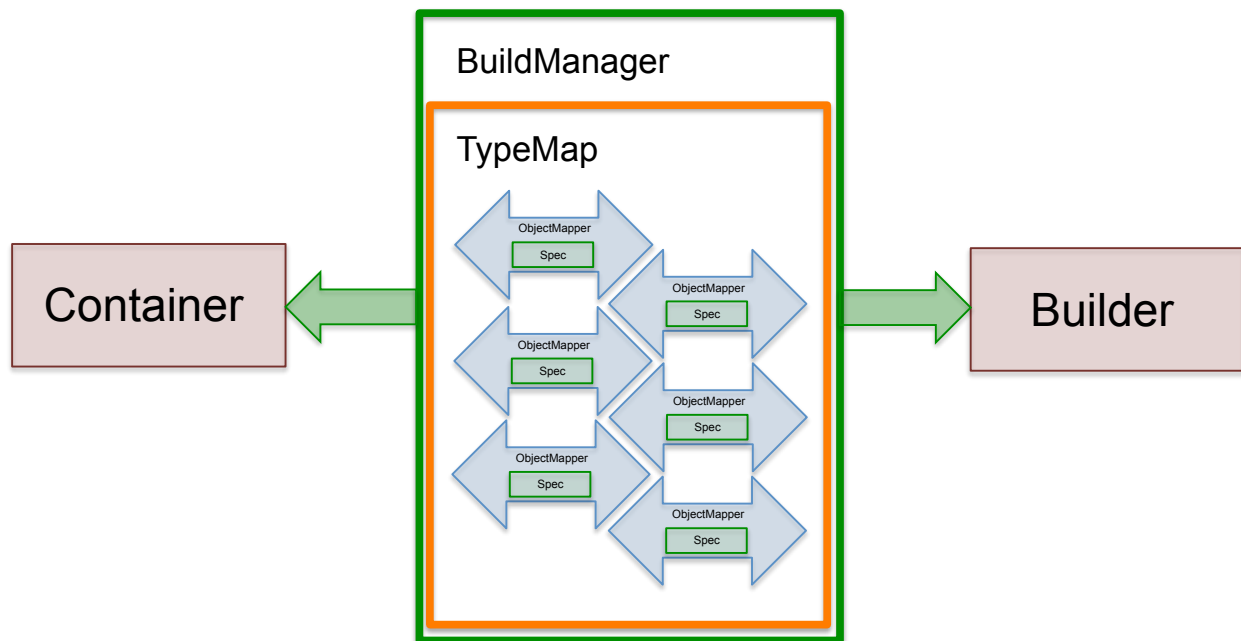


Fig. 5: Overview of *BuildManager* (and *TypeMap*) (click to enlarge).

#### 4.2.4 HDMFIO

- Abstract base class for I/O
- *HDMFIO* has two key abstract methods:
  - *write\_builder* – given a builder, write data to storage format
  - *read\_builder* – given a handle to storage format, return builder representation
  - Others: *open* and *close*
- Constructed with a *BuildManager*
- Extend this for creating a new I/O backend
- HDMF has one concrete form of this:
  - *HDF5IO* - reading and writing HDF5

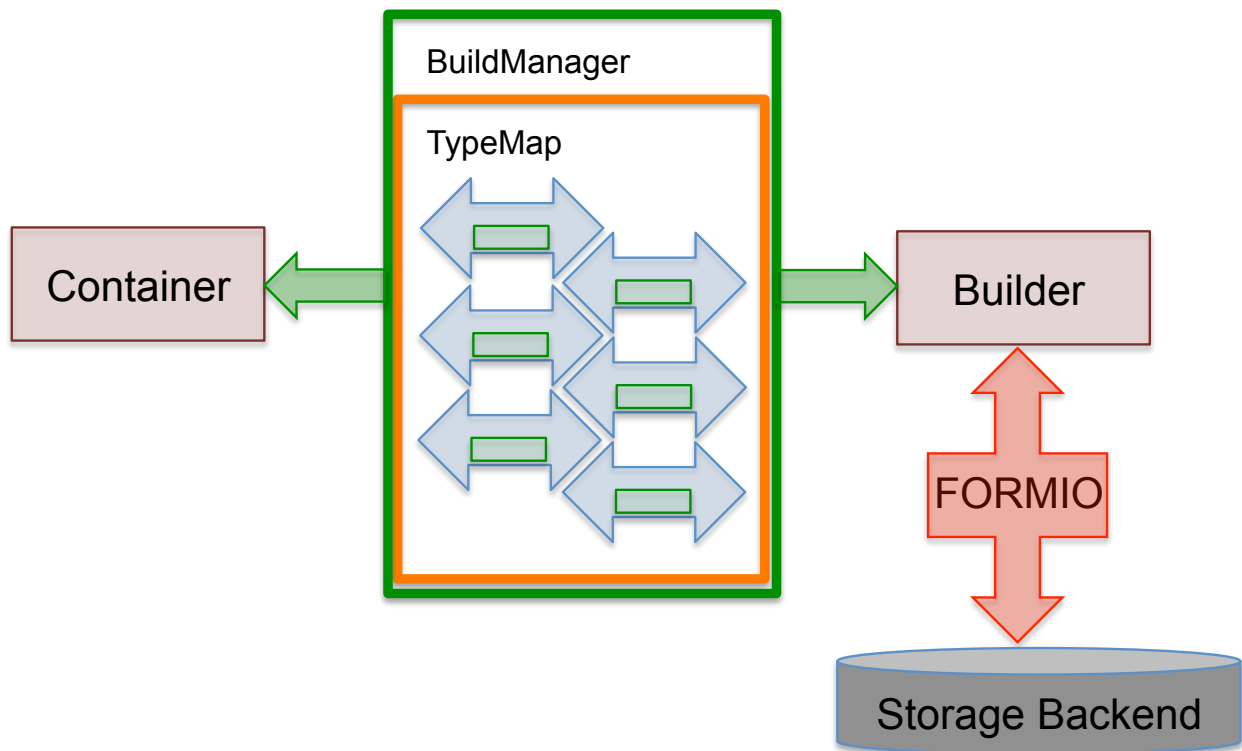


Fig. 6: Overview of *HDMFIO* (click to enlarge).



## CITING HDMF

## 5.1 BibTeX entry

If you use HDMF in your research, please use the following citation:

```
@INPROCEEDINGS{9005648,  
  author={A. J. {Tritt} and O. {Rübel} and B. {Dichter} and R. {Ly} and D. {Kang} and E. {Chang} and L. M. {Frank} and K. {Bouchard}},  
  booktitle={2019 IEEE International Conference on Big Data (Big Data)},  
  title={HDMF: Hierarchical Data Modeling Framework for Modern Science Data Standards},  
  year={2019},  
  volume={},  
  number={},  
  pages={165-179},  
  doi={10.1109/BigData47090.2019.9005648}}
```

## 5.2 Using RRID

- **RRID:** (Hierarchical Data Modeling Framework, RRID:SCR\_021303)

## 5.3 Using duecredit

Citations can be generated using [duecredit](#). To install duecredit, run `pip install duecredit`.

You can obtain a list of citations for your Python script, e.g., `yourscript.py`, using:

```
cd /path/to/your/module  
python -m duecredit yourscrip.py
```

Alternatively, you can set the environment variable `DUECREDIT_ENABLE=yes`

```
DUECREDIT-ENABLE=yes python yourscrip.py
```

Citations will be saved in a hidden file (`.duecredit.p`) in the current directory. You can then use the [duecredit](#) command line tool to export the citations to different formats. For example, you can display your citations in BibTeX format using:

```
duecredit summary --format=bibtex
```

For more information on using duecredit, please consult its [homepage](#).



## API DOCUMENTATION

## 6.1 hdmf.common package

### 6.1.1 Subpackages

**hdmf.common.io package**

**Submodules**

**hdmf.common.io.alignedtable module**

**class** hdmf.common.io.alignedtable.**AlignedDynamicTableMap**(*spec*)

Bases: *DynamicTableMap*

Customize the mapping for AlignedDynamicTable

Create a map from AbstractContainer attributes to specifications

**Parameters**

**spec** (*DatasetSpec* or *GroupSpec*) – The specification for mapping objects to builders

**constructor\_args** = {'name': <function *ObjectMapper.get\_container\_name*>}

**obj\_attrs** = {'colnames': <function *DynamicTableMap.attr\_columns*>}

**hdmf.common.io.multi module**

**class** hdmf.common.io.multi.**SimpleMultiContainerMap**(*spec*)

Bases: *ObjectMapper*

Create a map from AbstractContainer attributes to specifications

**Parameters**

**spec** (*DatasetSpec* or *GroupSpec*) – The specification for mapping objects to builders

**containers\_attr**(*container*, *manager*)

**containers\_carg**(*builder*, *manager*)

**datas\_attr**(*container*, *manager*)

```
constructor_args = {'containers': <function
SimpleMultiContainerMap.containers_carg>, 'name': <function
ObjectMapper.get_container_name>}

obj_attrs = {'containers': <function SimpleMultiContainerMap.containers_attr>,
'datas': <function SimpleMultiContainerMap.datas_attr>}
```

## hdmf.common.io.resources module

**class** hdmf.common.io.resources.HERDMap(*spec*)

Bases: *ObjectMapper*

Create a map from AbstractContainer attributes to specifications

### Parameters

**spec** (*DatasetSpec* or *GroupSpec*) – The specification for mapping objects to builders

**construct\_helper**(*name*, *parent\_builder*, *table\_cls*, *manager*)

Create a new instance of *table\_cls* with data from *parent\_builder*[*name*].

The *DatasetBuilder* for *name* is associated with *data\_type* *Data* and container class *Data*, but users should use the more specific *table\_cls* for these datasets.

**keys**(*builder*, *manager*)

**files**(*builder*, *manager*)

**entities**(*builder*, *manager*)

**objects**(*builder*, *manager*)

**object\_keys**(*builder*, *manager*)

**entity\_keys**(*builder*, *manager*)

```
constructor_args = {'entities': <function HERDMap.entities>, 'entity_keys':
<function HERDMap.entity_keys>, 'files': <function HERDMap.files>, 'keys':
<function HERDMap.keys>, 'name': <function ObjectMapper.get_container_name>,
'object_keys': <function HERDMap.object_keys>, 'objects': <function
HERDMap.objects>}
```

```
obj_attrs = {}
```

## hdmf.common.io.table module

**class** hdmf.common.io.table.DynamicTableMap(*spec*)

Bases: *ObjectMapper*

Create a map from AbstractContainer attributes to specifications

### Parameters

**spec** (*DatasetSpec* or *GroupSpec*) – The specification for mapping objects to builders

**attr\_columns**(*container*, *manager*)

**get\_attr\_value**(*spec, container, manager*)

Get the value of the attribute corresponding to this spec from the given container

**Parameters**

- **spec** (*Spec*) – the spec to get the attribute value for
- **container** (*DynamicTable*) – the container to get the attribute value from
- **manager** (*BuildManager*) – the BuildManager used for managing this build

**constructor\_args** = {'name': <function ObjectMapper.get\_container\_name>}

**obj\_attrs** = {'colnames': <function DynamicTableMap.attr\_columns>}

**class** hdmf.common.io.table.DynamicTableGenerator(\*args, \*\*kwargs)

Bases: *CustomClassGenerator*

**classmethod** apply\_generator\_to\_field(*field\_spec, bases, type\_map*)

Return True if this is a DynamicTable and the field spec is a column.

**classmethod** process\_field\_spec(*classdict, docval\_args, parent\_cls, attr\_name, not\_inherited\_fields, type\_map, spec*)

Add `__columns__` to the classdict and update the docval args for the field spec with the given attribute name. :param classdict: The dict to update with `__columns__`. :param docval\_args: The list of docval arguments. :param parent\_cls: The parent class. :param attr\_name: The attribute name of the field spec for the container class to generate. :param not\_inherited\_fields: Dictionary of fields not inherited from the parent class. :param type\_map: The type map to use. :param spec: The spec for the container class to generate.

**classmethod** post\_process(*classdict, bases, docval\_args, spec*)

Convert classdict['`__columns__`'] to tuple. :param classdict: The class dictionary. :param bases: The list of base classes. :param docval\_args: The dict of docval arguments. :param spec: The spec for the container class to generate.

## Module contents

### 6.1.2 Submodules

#### hdmf.common.alignedtable module

Collection of Container classes for interacting with aligned and hierarchical dynamic tables

**class** hdmf.common.alignedtable.AlignedDynamicTable(*name, description, id=None, columns=None, colnames=None, target\_tables=None, category\_tables=None, categories=None*)

Bases: *DynamicTable*

DynamicTable container that supports storing a collection of subtables. Each sub-table is a DynamicTable itself that is aligned with the main table by row index. I.e., all DynamicTables stored in this group MUST have the same number of rows. This type effectively defines a 2-level table in which the main data is stored in the main table implemented by this type and additional columns of the table are grouped into categories, with each category being represented by a separate DynamicTable stored within the group.

**NOTE: To remain compatible with DynamicTable, the attribute colnames represents only the** columns of the main table (not including the category tables). To get the full list of column names, use the `get_colnames()` function instead.

**Parameters**

- **name** (*str*) – the name of this table
- **description** (*str*) – a description of what is in this table
- **id** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO* or *ElementIdentifiers*) – the identifiers for this table
- **columns** (*tuple* or *list*) – the columns in this table
- **colnames** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator*) – the ordered names of the columns in this table. columns must also be provided.
- **target\_tables** (*dict*) – dict mapping DynamicTableRegion column name to the table that the DTR points to. The column is added to the table if it is not already present (i.e., when it is optional).
- **category\_tables** (*list*) – List of DynamicTables to be added to the container. NOTE - Only regular DynamicTables are allowed. Using AlignedDynamicTable as a category for AlignedDynamicTable is currently not supported.
- **categories** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator*) – List of names with the ordering of category tables

**property category\_tables**

List of DynamicTables to be added to the container. NOTE - Only regular DynamicTables are allowed. Using AlignedDynamicTable as a category for AlignedDynamicTable is currently not supported.

**property categories**

Get the list of names the categories

Short-hand for `list(self.category_tables.keys())`

**Raises**

KeyError if the given name is not in `self.category_tables`

**add\_category(category)**

Add a new DynamicTable to the AlignedDynamicTable to create a new category in the table.

NOTE: The table must align with (i.e, have the same number of rows as) the main data table (and other category tables). I.e., if the AlignedDynamicTable is already populated with data then we have to populate the new category with the corresponding data before adding it.

**raises**

ValueError is raised if the input table does not have the same number of rows as the main table. ValueError is raised if the table is an AlignedDynamicTable instead of regular DynamicTable.

**Parameters**

**category** (*DynamicTable*) – Add a new DynamicTable category

**get\_category(name=None)****Parameters**

**name** (*str*) – Name of the category we want to retrieve

**add\_column**(*name*, *description*, *data*=[], *table*=False, *index*=False, *enum*=False, *col\_cls*=<class 'hdmf.common.table.VectorData'>, *check\_ragged*=True, *category*=None)

Add a column to the table

**raises**

KeyError if the category does not exist

**Parameters**

- **name** (*str*) – the name of this VectorData
- **description** (*str*) – a description for this column
- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – a dataset where the first dimension is a concatenation of multiple vectors
- **table** (*bool* or *DynamicTable*) – whether or not this is a table region or the table the region applies to
- **index** (*bool* or *VectorIndex* or *ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *int*) –
  - False (default): do not generate a VectorIndex
  - True: generate one empty VectorIndex
  - VectorIndex: Use the supplied VectorIndex
  - array-like of ints: Create a VectorIndex and use these values as the data
  - int: Recursively create *n* VectorIndex objects for a multi-ragged array
- **enum** (*bool* or *ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator*) – whether or not this column contains data from a fixed set of elements
- **col\_cls** (*type*) – class to use to represent the column data. If *table*=True, this field is ignored and a *DynamicTableRegion* object is used. If *enum*=True, this field is ignored and a *EnumData* object is used.
- **check\_ragged** (*bool*) – whether or not to check for ragged arrays when adding data to the table. Set to False to avoid checking every element if performance issues occur.
- **category** (*str*) – The category the column should be added to

**add\_row**(*data*=None, *id*=None, *enforce\_unique\_id*=False)

We can either provide the row data as a single dict or by specifying a dict for each category

**Parameters**

- **data** (*dict*) – the data to put in this row
- **id** (*int*) – the ID for the row
- **enforce\_unique\_id** (*bool*) – enforce that the id in the table must be unique

**get\_colnames**(*include\_category\_tables*=False, *ignore\_category\_ids*=False)

Get the full list of names of columns for this table

**returns**

List of tuples (*str*, *str*) where the first string is the name of the *DynamicTable* that contains the column and the second string is the name of the column. If *include\_category\_tables* is False, then a list of column names is returned.

**Parameters**

- **include\_category\_tables** (*bool*) – Ignore sub-category tables and just look at the main table
- **ignore\_category\_ids** (*bool*) – Ignore id columns of sub-category tables

**to\_dataframe**(*ignore\_category\_ids=False*)

Convert the collection of tables to a single pandas DataFrame

**Parameters**

- **ignore\_category\_ids** (*bool*) – Ignore id columns of sub-category tables

**\_\_getitem\_\_**(*item*)

Called to implement standard array slicing syntax.

Same as `self.get(item)`. See [get](#) for details.

**get**(*item, \*\*kwargs*)

Access elements (rows, columns, category tables etc.) from the table. Instead of calling this function directly, the class also implements standard array slicing syntax via `__getitem__` (which calls this function). For example, instead of calling `self.get(item=slice(2,5))` we may use the often more convenient form of `self[2:5]` instead.

**Parameters**

**item** – Selection defining the items of interest. This may be either a:

- **int, list, array, slice**

[Return one or multiple row of the table as a pandas.DataFrame. For example:]

- `self[0]` : Select the first row of the table
- `self[[0, 3]]` : Select the first and fourth row of the table
- `self[1:4]` : Select the rows with index 1,2,3 from the table

- **string**

[Return a column from the main table or a category table. For example:]

- `self['column']` : Return the column from the main table.
- `self['my_category']` : Returns a DataFrame of the `my_category` category table. This is a shorthand for `self.get_category('my_category').to_dataframe()`.

- **tuple**: Get a column, row, or cell from a particular category table. The tuple is expected to consist of the following elements:

- **category**: string with the name of the category. To select from the main table use `self.name` or `None`.
- **column**: string with the name of the column, and
- **row**: integer index of the row.

The tuple itself then may take the following forms:

- **Select a single column from a table via:**

\* `self[category, column]`

- **Select a single full row of a given category table via:**

\* `self[row, category]` (recommended, for consistency with `DynamicTable`)

\* `self[category, row]`

– Select a single cell via:

- \* `self[row, (category, column)]` (recommended, for consistency with `DynamicTable`)
- \* `self[row, category, column]`
- \* `self[category, column, row]`

### Returns

Depending on the type of selection the function returns a:

- **pandas.DataFrame**: when retrieving a row or category table
- **array** : when retrieving a single column
- **single value** : when retrieving a single cell. The data type and shape will depend on the data type and shape of the cell/column.

**has\_foreign\_columns**(*ignore\_category\_tables=False*)

Does the table contain `DynamicTableRegion` columns

### returns

True if the table or any of the category tables contains a `DynamicTableRegion` column, else False

### Parameters

**ignore\_category\_tables** (*bool*) – Ignore the category tables and only check in the main table columns

**get\_foreign\_columns**(*ignore\_category\_tables=False*)

**Determine the names of all columns that link to another `DynamicTable`, i.e.,**

find all `DynamicTableRegion` type columns. Similar to a foreign key in a database, a `DynamicTableRegion` column references elements in another table.

### returns

List of tuples (*str*, *str*) where the first string is the name of the category table (or `None` if the column is in the main table) and the second string is the column name.

### Parameters

**ignore\_category\_tables** (*bool*) – Ignore the category tables and only check in the main table columns

**get\_linked\_tables**(*other\_tables=None*, *ignore\_category\_tables=False*)

**Get a list of the full list of all tables that are being linked to directly or indirectly**

from this table via foreign `DynamicTableColumns` included in this table or in any table that can be reached through `DynamicTableRegion` columns

**Returns: List of dicts with the following keys:**

- ‘source\_table’ : The source table containing the `DynamicTableRegion` column
- ‘source\_column’ : The relevant `DynamicTableRegion` column in the ‘source\_table’
- ‘target\_table’ : The target `DynamicTable`; same as `source_column.table`.

### Parameters

- **other\_tables** (*list* or *tuple* or *set*) – List of additional tables to consider in the search. Usually this parameter is used for internal purposes, e.g., when we need to consider `AlignedDynamicTable`
- **ignore\_category\_tables** (*bool*) – Ignore the category tables and only check in the main table columns

```
data_type = 'AlignedDynamicTable'
```

```
namespace = 'hdmf-common'
```

### `hdmf.common.hierarchicaltable` module

Module providing additional functionality for dealing with hierarchically nested tables, i.e., tables containing `DynamicTableRegion` references.

`hdmf.common.hierarchicaltable.to_hierarchical_dataframe(dynamic_table)`

Create a hierarchical `pandas.DataFrame` that represents all data from a collection of linked `DynamicTables`.

**LIMITATIONS:** Currently this function only supports `DynamicTables` with a single `DynamicTableRegion` column. If a table has more than one `DynamicTableRegion` column then the function will expand only the first `DynamicTableRegion` column found for each table. Any additional `DynamicTableRegion` columns will remain nested.

**NOTE:** Some useful functions for further processing of the generated `DataFrame` include:

- `pandas.DataFrame.reset_index` to turn the data from the `pandas.MultiIndex` into columns
- `drop_id_columns` to remove all ‘id’ columns
- `flatten_column_index` to flatten the column index

#### Parameters

**dynamic\_table** (*DynamicTable*) – `DynamicTable` object to be converted to a hierarchical `pandas.DataFrame`

#### Returns

Hierarchical `pandas.DataFrame` with usually a `pandas.MultiIndex` on both the index and columns.

#### Return type

`DataFrame`

`hdmf.common.hierarchicaltable.drop_id_columns(dataframe, inplace=False)`

Drop all columns named ‘id’ from the table.

In case a column name is a tuple the function will drop any column for which the inner-most name is ‘id’. The ‘id’ columns of `DynamicTable` is in many cases not necessary for analysis or display. This function allow us to easily filter all those columns.

#### raises `TypeError`

In case that `dataframe` parameter is not a `pandas.DataFrame`.

#### Parameters

- **dataframe** (*DataFrame*) – `Pandas dataframe` to update (usually generated by the `to_hierarchical_dataframe` function)
- **inplace** (*bool*) – Update the `dataframe` inplace or return a modified copy



**Returns**

pandas.DataFrame with the id columns removed

**Return type**

DataFrame

`hdmf.common.hierarchicaltable.flatten_column_index(dataframe, max_levels=None, inplace=False)`

Flatten the column index of a pandas DataFrame.

The functions changes the dataframe.columns from a pandas.MultiIndex to a normal Index, with each column usually being identified by a tuple of strings. This function is typically used in conjunction with DataFrames generated by [to\\_hierarchical\\_dataframe](#)

**raises ValueError**

In case the num\_levels is not >0

**raises TypeError**

In case that dataframe parameter is not a pandas.DataFrame.

**Parameters**

- **dataframe** (DataFrame) – Pandas dataframe to update (usually generated by the [to\\_hierarchical\\_dataframe](#) function)
- **max\_levels** (int or integer) – Maximum number of levels to use in the resulting column Index. NOTE: When limiting the number of levels the function simply removes levels from the beginning. As such, removing levels may result in columns with duplicate names. Value must be >0.
- **inplace** (bool) – Update the dataframe inplace or return a modified copy

**Returns**

pandas.DataFrame with a regular pandas.Index columns rather and a pandas.MultiIndex

**Return type**

DataFrame

**hdmf.common.multi module**

`class hdmf.common.multi.SimpleMultiContainer(name, containers=None)`

Bases: [MultiContainerInterface](#)

**Parameters**

- **name** (str) – the name of this container
- **containers** (list or tuple) – the Container or Data objects in this file

**property containers**

a dictionary containing the Container or Data in this SimpleMultiContainer

`__getitem__(name=None)`

Get a Container from this SimpleMultiContainer

**Parameters**

**name** (str) – the name of the Container or Data

**Returns**

the Container or Data with the given name

**Return type***Container* or *Data***add\_container**(*containers*)

Add one or multiple Container or Data objects to this SimpleMultiContainer

**Parameters****containers** (*list* or *tuple* or *dict* or *Container* or *Data*) – one or multiple Container or Data objects to add to this SimpleMultiContainer**data\_type** = 'SimpleMultiContainer'**get\_container**(*name=None*)

Get a Container from this SimpleMultiContainer

**Parameters****name** (*str*) – the name of the Container or Data**Returns**

the Container or Data with the given name

**Return type***Container* or *Data***namespace** = 'hdmf-common'**hdmf.common.resources module****class** hdmf.common.resources.**KeyTable**(*name='keys', data=[]*)Bases: *Table*

A table for storing keys used to reference external resources.

**Parameters**

- **name** (*str*) – the name of this table
- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the data in this table

**add\_row**(*key*)**Parameters****key** (*str*) – The user key that maps to the resource term / registry symbol.**class** hdmf.common.resources.**Key**(*key, table=None, idx=None*)Bases: *Row*

A Row class for representing rows in the KeyTable.

**Parameters**

- **key** (*str*) – The user key that maps to the resource term / registry symbol.
- **table** (*Table*) – None
- **idx** (*int*) – None

**todict**()

```
class hdmf.common.resources.EntityTable(name='entities', data=[])
```

Bases: [Table](#)

A table for storing the external resources a key refers to.

#### Parameters

- **name** ([str](#)) – the name of this table
- **data** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Array](#) or [StrDataset](#) or [HDMFDataset](#) or [AbstractDataChunkIterator](#) or [DataIO](#)) – the data in this table

```
add_row(entity_id, entity_uri)
```

#### Parameters

- **entity\_id** ([str](#)) – The unique ID for the resource term / registry symbol.
- **entity\_uri** ([str](#)) – The URI for the resource term / registry symbol.

```
class hdmf.common.resources.Entity(entity_id, entity_uri, table=None, idx=None)
```

Bases: [Row](#)

A Row class for representing rows in the EntityTable.

#### Parameters

- **entity\_id** ([str](#)) – The unique ID for the resource term / registry symbol.
- **entity\_uri** ([str](#)) – The URI for the resource term / registry symbol.
- **table** ([Table](#)) – None
- **idx** ([int](#)) – None

```
todict()
```

```
class hdmf.common.resources.FileTable(name='files', data=[])
```

Bases: [Table](#)

A table for storing file ids used in external resources.

#### Parameters

- **name** ([str](#)) – the name of this table
- **data** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Array](#) or [StrDataset](#) or [HDMFDataset](#) or [AbstractDataChunkIterator](#) or [DataIO](#)) – the data in this table

```
add_row(file_object_id)
```

#### Parameters

- **file\_object\_id** ([str](#)) – The file id of the file that contains the object

```
class hdmf.common.resources.File(file_object_id, table=None, idx=None)
```

Bases: [Row](#)

A Row class for representing rows in the FileTable.

#### Parameters

- **file\_object\_id** ([str](#)) – The file id of the file that contains the object
- **table** ([Table](#)) – None
- **idx** ([int](#)) – None

`todict()`

`class hdmf.common.resources.ObjectTable(name='objects', data=[])`

Bases: [`Table`](#)

A table for storing objects (i.e. Containers) that contain keys that refer to external resources.

**Parameters**

- **name** (`str`) – the name of this table
- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – the data in this table

`add_row(files_idx, object_id, object_type, relative_path, field)`

**Parameters**

- **files\_idx** (`int`) – The row idx for the file\_object\_id in FileTable containing the object.
- **object\_id** (`str`) – The object ID for the Container/Data.
- **object\_type** (`str`) – The type of the object. This is also the parent in relative\_path.
- **relative\_path** (`str`) – ('The relative\_path of the attribute of the object that uses ', 'an external resource reference key. Use an empty string if not applicable.')
- **field** (`str`) – The field of the compound data type using an external resource. Use an empty string if not applicable.

`class hdmf.common.resources.Object(files_idx, object_id, object_type, relative_path, field, table=None, idx=None)`

Bases: [`Row`](#)

A Row class for representing rows in the ObjectTable.

**Parameters**

- **files\_idx** (`int`) – The row idx for the file\_object\_id in FileTable containing the object.
- **object\_id** (`str`) – The object ID for the Container/Data.
- **object\_type** (`str`) – The type of the object. This is also the parent in relative\_path.
- **relative\_path** (`str`) – ('The relative\_path of the attribute of the object that uses ', 'an external resource reference key. Use an empty string if not applicable.')
- **field** (`str`) – The field of the compound data type using an external resource. Use an empty string if not applicable.
- **table** ([`Table`](#)) – None
- **idx** (`int`) – None

`todict()`

`class hdmf.common.resources.ObjectKeyTable(name='object_keys', data=[])`

Bases: [`Table`](#)

A table for identifying which keys are used by which objects for referring to external resources.

**Parameters**

- **name** (`str`) – the name of this table
- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – the data in this table

**add\_row**(*objects\_idx*, *keys\_idx*)

**Parameters**

- **objects\_idx** (*int* or *Object*) – The index into the objects table for the Object that uses the Key.
- **keys\_idx** (*int* or *Key*) – The index into the keys table that is used to make an external resource reference.

**class** hdmf.common.resources.**EntityKeyTable**(*name='entity\_keys'*, *data=[]*)

Bases: *Table*

A table for identifying which entities are used by which keys for referring to external resources.

**Parameters**

- **name** (*str*) – the name of this table
- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the data in this table

**add\_row**(*entities\_idx*, *keys\_idx*)

**Parameters**

- **entities\_idx** (*int* or *Entity*) – The index into the EntityTable for the Entity that associated with the Key.
- **keys\_idx** (*int* or *Key*) – The index into the KeyTable that is used to make an external resource reference.

**class** hdmf.common.resources.**EntityKey**(*entities\_idx*, *keys\_idx*, *table=None*, *idx=None*)

Bases: *Row*

A Row class for representing rows in the EntityKeyTable.

**Parameters**

- **entities\_idx** (*int* or *Entity*) – The index into the EntityTable for the Entity that associated with the Key.
- **keys\_idx** (*int* or *Key*) – The index into the KeyTable that is used to make an external resource reference.
- **table** (*Table*) – None
- **idx** (*int*) – None

**todict**()

**class** hdmf.common.resources.**ObjectKey**(*objects\_idx*, *keys\_idx*, *table=None*, *idx=None*)

Bases: *Row*

A Row class for representing rows in the ObjectKeyTable.

**Parameters**

- **objects\_idx** (*int* or *Object*) – The index into the objects table for the Object that uses the Key.
- **keys\_idx** (*int* or *Key*) – The index into the keys table that is used to make an external resource reference.
- **table** (*Table*) – None

- **idx** (*int*) – None

**todict()**

```
class hdmf.common.resources.HERD(keys=None, files=None, entities=None, objects=None,  
                                object_keys=None, entity_keys=None, type_map=None)
```

Bases: *Container*

HDMF External Resources Data Structure. A table for mapping user terms (i.e. keys) to resource entities.

#### Parameters

- **keys** (*KeyTable*) – The table storing user keys for referencing resources.
- **files** (*FileTable*) – The table for storing file ids used in external resources.
- **entities** (*EntityTable*) – The table storing entity information.
- **objects** (*ObjectTable*) – The table storing object information.
- **object\_keys** (*ObjectKeyTable*) – The table storing object-key relationships.
- **entity\_keys** (*EntityKeyTable*) – The table storing entity-key relationships.
- **type\_map** (*TypeMap*) – The type map. If None is provided, the HDMF-common type map will be used.

#### property keys

The table storing user keys for referencing resources.

#### property files

The table for storing file ids used in external resources.

#### property entities

The table storing entity information.

#### property objects

The table storing object information.

#### property object\_keys

The table storing object-key relationships.

#### property entity\_keys

The table storing entity-key relationships.

```
static assert_external_resources_equal(left, right, check_dtype=True)
```

Compare that the keys, resources, entities, objects, and object\_keys tables match

#### Parameters

- **left** – HERD object to compare with right
- **right** – HERD object to compare with left
- **check\_dtype** – Enforce strict checking of dtypes. Dtypes may be different for example for ids, where depending on how the data was saved ids may change from int64 to int32. (Default: True)

#### Returns

The function returns True if all values match. If mismatches are found, AssertionError will be raised.

**Raises**

**AssertionError** – Raised if any differences are found. The function collects all differences into a single error so that the assertion will indicate all found differences.

**add\_ref\_container**(*root\_container*)

**Method to search through the root\_container for all instances of TermSet.**

Currently, only datasets are supported. By using a TermSet, the data comes validated and can use the permissible values within the set to populate HERD.

**Parameters**

**root\_container** (*HERDManager*) – The root container or file containing objects with a TermSet.

**add\_ref\_termset**(*termset*, *file=None*, *container=None*, *attribute=None*, *field="*, *key=None*)

**This method allows users to take advantage of using the TermSet class to provide the entity information**

for add\_ref, while also validating the data. This method supports adding a single key or an entire dataset to the HERD tables. For both cases, the term, i.e., key, will be validated against the permissible values in the TermSet. If valid, it will proceed to call add\_ref. Otherwise, the method will return a dict of missing terms (terms not found in the TermSet).

**Parameters**

- **termset** (*TermSet*) – The TermSet to be used if the container/attribute does not have one.
- **file** (*HERDManager*) – The file associated with the container.
- **container** (*str* or *AbstractContainer*) – The Container/Data object that uses the key or the object\_id for the Container/Data object that uses the key.
- **attribute** (*str*) – The attribute of the container for the external reference.
- **field** (*str*) – The field of the compound data type using an external resource.
- **key** (*str* or *Key*) – The name of the key or the Key object from the KeyTable for the key to add a resource for.

**add\_ref**(*entity\_id*, *container=None*, *attribute=None*, *field="*, *key=None*, *entity\_uri=None*, *file=None*)

Add information about an external reference used in this file.

It is possible to use the same name of the key to refer to different resources so long as the name of the key is not used within the same object, relative\_path, and field combination. This method does not support such functionality by default.

**Parameters**

- **entity\_id** (*str*) – The identifier for the entity at the resource.
- **container** (*str* or *AbstractContainer*) – The Container/Data object that uses the key or the object\_id for the Container/Data object that uses the key.
- **attribute** (*str*) – The attribute of the container for the external reference.
- **field** (*str*) – The field of the compound data type using an external resource.
- **key** (*str* or *Key*) – The name of the key or the Key object from the KeyTable for the key to add a resource for.
- **entity\_uri** (*str*) – The URI for the identifier at the resource.

- **file** (*HERDManager*) – The file associated with the container.

**get\_key**(*key\_name*, *file=None*, *container=None*, *relative\_path=""*, *field=""*)

Return a Key.

If *container*, *relative\_path*, and *field* are provided, the Key that corresponds to the given name of the key for the given container, *relative\_path*, and *field* is returned.

If there are multiple matches, a list of all matching keys will be returned.

#### Parameters

- **key\_name** (*str*) – The name of the Key to get.
- **file** (*HERDManager*) – The file associated with the container.
- **container** (*str* or *AbstractContainer*) – The Container/Data object that uses the key or the object id for the Container/Data object that uses the key.
- **relative\_path** (*str*) – ('The *relative\_path* of the attribute of the object that uses ', 'an external resource reference key. Use an empty string if not applicable.')
- **field** (*str*) – The field of the compound data type using an external resource.

**get\_entity**(*entity\_id*)

#### Parameters

**entity\_id** (*str*) – The ID for the identifier at the resource.

**get\_object\_type**(*object\_type*, *relative\_path=""*, *field=""*, *all\_instances=False*)

Get all entities/resources associated with an *object\_type*.

#### Parameters

- **object\_type** (*str*) – The type of the object. This is also the parent in *relative\_path*.
- **relative\_path** (*str*) – ('The *relative\_path* of the attribute of the object that uses ', 'an external resource reference key. Use an empty string if not applicable.')
- **field** (*str*) – The field of the compound data type using an external resource.
- **all\_instances** (*bool*) – ('The bool to return a dataframe with all instances of the *object\_type*.', 'If True, *relative\_path* and *field* inputs will be ignored.')

**get\_object\_entities**(*container*, *file=None*, *attribute=None*, *relative\_path=""*, *field=""*)

Get all entities/resources associated with an object.

#### Parameters

- **container** (*str* or *AbstractContainer*) – The Container/data object that is linked to resources/entities.
- **file** (*HERDManager*) – The file.
- **attribute** (*str*) – The attribute of the container for the external reference.
- **relative\_path** (*str*) – ('The *relative\_path* of the attribute of the object that uses ', 'an external resource reference key. Use an empty string if not applicable.')
- **field** (*str*) – The field of the compound data type using an external resource.



**to\_dataframe**(*use\_categories=False*)

**Convert the data from the keys, resources, entities, objects, and object\_keys tables**

to a single joint dataframe. I.e., here data is being denormalized, e.g., keys that are used across multiple entities or objects will be duplicated across the corresponding rows.

Returns: `DataFrame` with all data merged into a single, flat, denormalized table.

**Parameters**

**use\_categories** (`bool`) – Use a multi-index on the columns to indicate which category each column belongs to.

**Returns**

A `DataFrame` with all data merged into a flat, denormalized table.

**Return type**

`DataFrame`

**to\_zip**(*path*)

Write the tables in HERD to zipped tsv files.

**Parameters**

**path** (`str`) – The path to the zip file.

**classmethod get\_zip\_directory**(*path*)

Return the directory of the file given.

**Parameters**

**path** (`str`) – The path to the zip file.

**classmethod from\_zip**(*path*)

Method to read in zipped tsv files to populate HERD.

**Parameters**

**path** (`str`) – The path to the zip file.

**data\_type** = 'HERD'

**namespace** = 'hdmf-experimental'

## hdmf.common.sparse module

**class** `hdmf.common.sparse.CSRMatrix`(*data, indices=None, indptr=None, shape=None, name='csr\_matrix'*)

Bases: `Container`

**Parameters**

- **data** (`csr_matrix` or `ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – the data to use for this `CSRMatrix` or CSR data array. If passing CSR data array, *indices*, *indptr*, and *shape* must also be provided
- **indices** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – CSR index array
- **indptr** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – CSR index pointer array
- **shape** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – the shape of the matrix

- **name** (*str*) – the name to use for this when storing

```
to_spmat()

data_type = 'CSRMatrix'

namespace = 'hdmf-common'
```

### hdmf.common.table module

Collection of Container classes for interacting with data types related to the storage and use of dynamic data tables as part of the hdmf-common schema

```
class hdmf.common.table.VectorData(name, description, data=[])
```

Bases: *Data*

A n-dimensional dataset representing a column of a DynamicTable. If used without an accompanying VectorIndex, first dimension is along the rows of the DynamicTable and each step along the first dimension is a cell of the larger table. VectorData can also be used to represent a ragged array if paired with a VectorIndex. This allows for storing arrays of varying length in a single cell of the DynamicTable by indexing into this VectorData. The first vector is at VectorData[0:VectorIndex(0)+1]. The second vector is at VectorData[VectorIndex(0)+1:VectorIndex(1)+1], and so on.

#### Parameters

- **name** (*str*) – the name of this VectorData
- **description** (*str*) – a description for this column
- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – a dataset where the first dimension is a concatenation of multiple vectors

#### property description

a description for this column

#### add\_row(val)

Append a data value to this VectorData column

#### Parameters

**val** (*None*) – the value to add to this column

#### get(key, \*\*kwargs)

Retrieve elements from this VectorData

#### Parameters

- **key** – Selection of the elements
- **kwargs** – Ignored

#### extend(ar, \*\*kwargs)

Add all elements of the iterable arg to the end of this VectorData.

Each subclass of VectorData should have its own extend method to ensure functionality and efficiency.

#### Parameters

**arg** – The iterable to add to the end of this VectorData

```
data_type = 'VectorData'
```

```
namespace = 'hdmf-common'
```

```
class hdmf.common.table.VectorIndex(name, data, target)
```

Bases: [VectorData](#)

When paired with a [VectorData](#), this allows for storing arrays of varying length in a single cell of the [DynamicTable](#) by indexing into this [VectorData](#). The first vector is at `VectorData[0:VectorIndex(0)+1]`. The second vector is at `VectorData[VectorIndex(0)+1:VectorIndex(1)+1]`, and so on.

#### Parameters

- **name** ([str](#)) – the name of this [VectorIndex](#)
- **data** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Array](#) or [StrDataset](#) or [HDMFDataset](#) or [AbstractDataChunkIterator](#) or [DataIO](#)) – a 1D dataset containing indexes that apply to [VectorData](#) object
- **target** ([VectorData](#)) – the target dataset that this index applies to

#### property target

the target dataset that this index applies to

```
add_vector(arg, **kwargs)
```

Add the given data value to the target [VectorData](#) and append the corresponding index to this [VectorIndex](#)  
:param arg: The data value to be added to self.target

```
add_row(arg, **kwargs)
```

Convenience function. Same as [add\\_vector](#)

```
__getitem__(arg)
```

Select elements in this [VectorIndex](#) and retrieve the corresponding data from the self.target [VectorData](#)

#### Parameters

**arg** – slice or integer index indicating the elements we want to select in this [VectorIndex](#)

#### Returns

Scalar or list of values retrieved

```
get(arg, **kwargs)
```

Select elements in this [VectorIndex](#) and retrieve the corresponding data from the self.target [VectorData](#)

#### Parameters

- **arg** – slice or integer index indicating the elements we want to select in this [VectorIndex](#)
- **kwargs** – any additional arguments to `get` method of the self.target [VectorData](#)

#### Returns

Scalar or list of values retrieved

```
data_type = 'VectorIndex'
```

```
namespace = 'hdmf-common'
```

```
class hdmf.common.table.ElementIdentifiers(name, data=[])
```

Bases: [Data](#)

Data container with a list of unique identifiers for values within a dataset, e.g. rows of a [DynamicTable](#).

#### Parameters

- **name** ([str](#)) – the name of this [ElementIdentifiers](#)

- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – a 1D dataset containing integer identifiers

**data\_type** = 'ElementIdentifiers'

**namespace** = 'hdmf-common'

**class** `hdmf.common.table.DynamicTable`(*name, description, id=None, columns=None, colnames=None, target\_tables=None*)

Bases: `Container`

A column-based table. Columns are defined by the argument *columns*. This argument must be a list/tuple of `VectorData` and `VectorIndex` objects or a list/tuple of dicts containing the keys *name* and *description* that provide the name and description of each column in the table. Additionally, the keys *index*, *table*, *enum* can be used for specifying additional structure to the table columns. Setting the key *index* to `True` can be used to indicate that the `VectorData` column will store a ragged array (i.e. will be accompanied with a `VectorIndex`). Setting the key *table* to `True` can be used to indicate that the column will store regions to another `DynamicTable`. Setting the key *enum* to `True` can be used to indicate that the column data will come from a fixed set of values.

Columns in `DynamicTable` subclasses can be statically defined by specifying the class attribute `__columns__`, rather than specifying them at runtime at the instance level. This is useful for defining a table structure that will get reused. The requirements for `__columns__` are the same as the requirements described above for specifying table columns with the *columns* argument to the `DynamicTable` constructor.

#### Parameters

- **name** (`str`) – the name of this table
- **description** (`str`) – a description of what is in this table
- **id** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO` or `ElementIdentifiers`) – the identifiers for this table
- **columns** (`tuple` or `list`) – the columns in this table
- **colnames** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – the ordered names of the columns in this table. *columns* must also be provided.
- **target\_tables** (`dict`) – dict mapping `DynamicTableRegion` column name to the table that the DTR points to. The column is added to the table if it is not already present (i.e., when it is optional).

#### property description

a description of what is in this table

#### property id

the identifiers for this table

#### property colnames

the ordered names of the columns in this table. *columns* must also be provided.

#### property columns

the columns in this table

**add\_row**(*data=None, id=None, enforce\_unique\_id=False, check\_ragged=True*)

Add a row to the table. If *id* is not provided, it will auto-increment.

#### Parameters

- **data** (`dict`) – the data to put in this row

- **id** (`int`) – the ID for the row
- **enforce\_unique\_id** (`bool`) – enforce that the id in the table must be unique
- **check\_ragged** (`bool`) – whether or not to check for ragged arrays when adding data to the table. Set to False to avoid checking every element if performance issues occur.

**add\_column**(*name*, *description*, *data*=[], *table*=False, *index*=False, *enum*=False, *col\_cls*=<class 'hdmf.common.table.VectorData'>, *check\_ragged*=True)

Add a column to this table.

If data is provided, it must contain the same number of rows as the current state of the table.

Extra keyword arguments will be passed to the constructor of the column class (“col\_cls”).

**raises ValueError**

if the column has already been added to the table

#### Parameters

- **name** (`str`) – the name of this VectorData
- **description** (`str`) – a description for this column
- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – a dataset where the first dimension is a concatenation of multiple vectors
- **table** (`bool` or `DynamicTable`) – whether or not this is a table region or the table the region applies to
- **index** (`bool` or `VectorIndex` or `ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `int`) –
  - False (default): do not generate a VectorIndex
  - True: generate one empty VectorIndex
  - VectorIndex: Use the supplied VectorIndex
  - array-like of ints: Create a VectorIndex and use these values as the data
  - int: Recursively create *n* VectorIndex objects for a multi-ragged array
- **enum** (`bool` or `ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – whether or not this column contains data from a fixed set of elements
- **col\_cls** (`type`) – class to use to represent the column data. If table=True, this field is ignored and a DynamicTableRegion object is used. If enum=True, this field is ignored and a EnumData object is used.
- **check\_ragged** (`bool`) – whether or not to check for ragged arrays when adding data to the table. Set to False to avoid checking every element if performance issues occur.

**create\_region**(*name*, *region*, *description*)

Create a DynamicTableRegion selecting a region (i.e., rows) in this DynamicTable.

**raises**

IndexError if the provided region contains invalid indices

#### Parameters

- **name** (`str`) – the name of the DynamicTableRegion object

- **region** (*slice* or *list* or *tuple*) – the indices of the table
- **description** (*str*) – a brief description of what the region is

**\_\_getitem\_\_** (*key*)

**get** (*key*, *default=None*, *df=True*, *index=True*, *\*\*kwargs*)

Select a subset from the table.

If the table includes a `DynamicTableRegion` column, then by default, the index/indices of the `DynamicTableRegion` will be returned. If `df=True` and `index=False`, then the returned pandas `DataFrame` will contain a nested `DataFrame` in each row of the `DynamicTableRegion` column. If `df=False` and `index=True`, then a list of lists will be returned where the list containing the `DynamicTableRegion` column contains the indices of the `DynamicTableRegion`. Note that in this case, the `DynamicTable` referenced by the `DynamicTableRegion` can be accessed through the `table` attribute of the `DynamicTableRegion` object. `df=False` and `index=False` is not yet supported.

#### Parameters

**key** – Key defining which elements of the table to select. This may be one of the following:

- 1) string with the name of the column to select
- 2) a tuple consisting of (int, str) where the int selects the row and the string identifies the column to select by name
- 3) int, list of ints, array, or slice selecting a set of full rows in the table. If an int is used, then scalars are returned for each column that has a single value. If a list, array, or slice is used and `df=False`, then lists are returned for each column, even if the list, array, or slice resolves to a single row.

#### Returns

- 1) If key is a string, then return the `VectorData` object representing the column with the string name
- 2) If key is a tuple of (int, str), then return the scalar value of the selected cell
- 3) If key is an int, list, `np.ndarray`, or slice, then return `pandas.DataFrame` or lists consisting of one or more rows

#### Raises

`KeyError`

**get\_foreign\_columns** ()

Determine the names of all columns that link to another `DynamicTable`, i.e., find all `DynamicTableRegion` type columns. Similar to a foreign key in a database, a `DynamicTableRegion` column references elements in another table.

#### Returns

List of strings with the column names

**has\_foreign\_columns** ()

Does the table contain `DynamicTableRegion` columns

#### Returns

True if the table contains a `DynamicTableRegion` column, else False

**get\_linked\_tables** (*other\_tables=None*)

**Get a list of the full list of all tables that are being linked to directly or indirectly**

from this table via foreign `DynamicTableColumns` included in this table or in any table that can be reached through `DynamicTableRegion` columns

**Returns: List of NamedTuple objects with:**

- ‘source\_table’ : The source table containing the DynamicTableRegion column
- ‘source\_column’ : The relevant DynamicTableRegion column in the ‘source\_table’
- ‘target\_table’ : The target DynamicTable; same as source\_column.table.

**Parameters**

**other\_tables** (*list* or *tuple* or *set*) – List of additional tables to consider in the search. Usually this parameter is used for internal purposes, e.g., when we need to consider Aligned-DynamicTable

**to\_dataframe**(*exclude=None, index=False*)

Produce a pandas DataFrame containing this table’s data.

If this table contains a DynamicTableRegion, by default,

If exclude is None, this is equivalent to table.get(slice(None, None, None), index=False).

**Parameters**

- **exclude** (*set*) – Set of column names to exclude from the dataframe
- **index** (*bool*) – Whether to return indices for a DynamicTableRegion column. If False, nested dataframes will be returned.

**generate\_html\_repr**(*level: int = 0, access\_code: str = "", nrows: int = 4*)

**classmethod from\_dataframe**(*df, name, index\_column=None, table\_description="", columns=None*)

Construct an instance of DynamicTable (or a subclass) from a pandas DataFrame.

The columns of the resulting table are defined by the columns of the dataframe and the index by the dataframe’s index (make sure it has a name!) or by a column whose name is supplied to the *index\_column* parameter. We recommend that you supply *columns* - a list/tuple of dictionaries containing the name and description of the column- to help others understand the contents of your table. See [DynamicTable](#) for more details on *columns*.

**Parameters**

- **df** (*DataFrame*) – source DataFrame
- **name** (*str*) – the name of this table
- **index\_column** (*str*) – if provided, this column will become the table’s index
- **table\_description** (*str*) – a description of what is in the resulting table
- **columns** (*list* or *tuple*) – a list/tuple of dictionaries specifying columns in the table

**copy()**

Return a copy of this DynamicTable. This is useful for linking.

**data\_type** = 'DynamicTable'

**namespace** = 'hdmf-common'

```
class hdmf.common.table.DynamicTableRegion(name, data, description, table=None)
```

Bases: [VectorData](#)

`DynamicTableRegion` provides a link from one table to an index or region of another. The `table` attribute is another `DynamicTable`, indicating which table is referenced. The data is `int(s)` indicating the row(s) (0-indexed) of the target array. *DynamicTableRegion's can be used to associate multiple rows with the same meta-data without data duplication. They can also be used to create hierarchical relationships between multiple 'DynamicTable's.* `DynamicTableRegion` objects may be paired with a `VectorIndex` object to create ragged references, so a single cell of a `DynamicTable` can reference many rows of another `DynamicTable`.

#### Parameters

- **name** (`str`) – the name of this `VectorData`
- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator` or `DataIO`) – a dataset where the first dimension is a concatenation of multiple vectors
- **description** (`str`) – a description of what this region represents
- **table** (`DynamicTable`) – the `DynamicTable` this region applies to

#### property table

The `DynamicTable` this `DynamicTableRegion` is pointing to

```
__getitem__(arg)
```

```
get(arg, index=False, df=True, **kwargs)
```

Subset the `DynamicTableRegion`

#### Parameters

- **arg** – Key defining which elements of the table to select. This may be one of the following:
  - 1) string with the name of the column to select
  - 2) a tuple consisting of (`int`, `str`) where the `int` selects the row and the string identifies the column to select by name
  - 3) `int`, list of `ints`, array, or slice selecting a set of full rows in the table. If an `int` is used, then scalars are returned for each column that has a single value. If a list, array, or slice is used and `df=False`, then lists are returned for each column, even if the list, array, or slice resolves to a single row.
- **index** – Boolean indicating whether to return indices of the DTR (default `False`)
- **df** – Boolean indicating whether to return the result as a pandas `DataFrame` (default `True`)

#### Returns

Result from `self.table[...]` with the appropriate selection based on the rows selected by this `DynamicTableRegion`

```
to_dataframe(**kwargs)
```

Convert the whole `DynamicTableRegion` to a pandas dataframe.

Keyword arguments are passed through to the `to_dataframe` method of `DynamicTable` that is being referenced (i.e., `self.table`). This allows specification of the 'exclude' parameter and any other parameters of `DynamicTable.to_dataframe`.

#### property shape

Define the shape, i.e., (`num_rows`, `num_columns`) of the selected table region :return: Shape tuple with two integers indicating the number of rows and number of columns



```
data_type = 'DynamicTableRegion'
```

```
namespace = 'hdmf-common'
```

```
class hdmf.common.table.EnumData(name, description, data=[], elements=[])
```

Bases: [VectorData](#)

A n-dimensional dataset that can contain elements from fixed set of elements.

#### Parameters

- **name** ([str](#)) – the name of this column
- **description** ([str](#)) – a description for this column
- **data** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Array](#) or [StrDataset](#) or [HDMFDataset](#) or [AbstractDataChunkIterator](#) or [DataIO](#)) – integers that index into elements for the value of each row
- **elements** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Array](#) or [StrDataset](#) or [HDMFDataset](#) or [AbstractDataChunkIterator](#) or [DataIO](#) or [VectorData](#)) – lookup values for each integer in data

#### property elements

lookup values for each integer in data

```
data_type = 'EnumData'
```

```
namespace = 'hdmf-experimental'
```

```
__getitem__(arg)
```

```
get(arg, index=False, join=False, **kwargs)
```

Return elements elements for the given argument.

#### Parameters

- **index** ([bool](#)) – Return indices, do not return CV elements
- **join** ([bool](#)) – Concatenate elements together into a single string

#### Returns

CV elements if *join* is False or a concatenation of all selected elements if *join* is True.

```
add_row(val, index=False)
```

Append a data value to this EnumData column

If an element is provided for *val* (i.e. *index* is False), the correct index value will be determined. Otherwise, *val* will be added as provided.

#### Parameters

- **val** ([None](#)) – the value to add to this column
- **index** ([bool](#)) – whether or not the value being added is an index

### 6.1.3 Module contents

This package will contain functions, classes, and objects for reading and writing data in according to the HDMF-common specification

`hdmf.common.load_type_config(config_path)`

**This method will either load the default config or the config provided by the path.**

NOTE: This config is global and shared across all type maps.

**Parameters**

**config\_path** (`str`) – Path to the configuration file.

`hdmf.common.get_loaded_type_config()`

This method returns the entire config file.

`hdmf.common.unload_type_config()`

Unload the configuration file.

`hdmf.common.register_class(data_type, namespace='hdmf-common', container_cls=None)`

**Register an Container class to use for reading and writing a data\_type from a specification**

If container\_cls is not specified, returns a decorator for registering an Container subclass as the class for data\_type in namespace.

**Parameters**

- **data\_type** (`str`) – the data\_type to get the spec for
- **namespace** (`str`) – the name of the namespace
- **container\_cls** (`type`) – the class to map to the specified data\_type

`hdmf.common.register_map(container_cls, mapper_cls=None)`

**Register an ObjectMapper to use for a Container class type**

If mapper\_cls is not specified, returns a decorator for registering an ObjectMapper class as the mapper for container\_cls. If mapper\_cls specified, register the class as the mapper for container\_cls

**Parameters**

- **container\_cls** (`type`) – the Container class for which the given ObjectMapper class gets used for
- **mapper\_cls** (`type`) – the ObjectMapper class to use to map

`hdmf.common.load_namespaces(namespace_path)`

Load namespaces from file

**Parameters**

**namespace\_path** (`str`) – the path to the YAML with the namespace definition

**Returns**

the namespaces loaded from the given file

**Return type**

`tuple`

`hdmf.common.available_namespaces()`

`hdmf.common.get_class(data_type, namespace)`

Get the class object of the Container subclass corresponding to a given `neurdata_type`.

**Parameters**

- **data\_type** (`str`) – the `data_type` to get the Container class for
- **namespace** (`str`) – the namespace the `data_type` is defined in

`hdmf.common.get_type_map(extensions=None)`

**Get a BuildManager to use for I/O using the given extensions. If no extensions are provided, return a BuildManager that uses the core namespace**

**Parameters**

**extensions** (`str` or `TypeMap` or `list`) – a path to a namespace, a `TypeMap`, or a list consisting paths to namespaces and `TypeMaps`

**Returns**

the namespaces loaded from the given file

**Return type**

`tuple`

`hdmf.common.get_manager(extensions=None)`

**Get a BuildManager to use for I/O using the given extensions. If no extensions are provided, return a BuildManager that uses the core namespace**

**Parameters**

**extensions** (`str` or `TypeMap` or `list`) – a path to a namespace, a `TypeMap`, or a list consisting paths to namespaces and `TypeMaps`

**Returns**

a build manager with namespaces loaded from the given file

**Return type**

`BuildManager`

`hdmf.common.validate(io, namespace='hdmf-common', experimental=False)`

Validate an file against a namespace

**Parameters**

- **io** (`HDMFIO`) – the `HDMFIO` object to read from
- **namespace** (`str`) – the namespace to validate against
- **experimental** (`bool`) – data type is an experimental data type

**Returns**

errors in the file

**Return type**

`list`

`hdmf.common.get_hdf5io(path=None, mode='r', manager=None, comm=None, file=None, driver=None, herd_path=None)`

A convenience method for getting an `HDF5IO` object using an `HDMF-common` build manager if none is provided.

**Parameters**

- **path** (`str` or `Path`) – the path to the HDF5 file
- **mode** (`str`) – the mode to open the HDF5 file with, one of (“w”, “r”, “r+”, “a”, “w-”, “x”). See `h5py.File` for more details.
- **manager** (`TypeMap` or `BuildManager`) – the `BuildManager` or a `TypeMap` to construct a `BuildManager` to use for I/O
- **comm** (`Intracomm`) – the MPI communicator to use for parallel I/O
- **file** (`File` or `S3File` or `RemFile`) – a pre-existing `h5py.File`, `S3File`, or `RemFile` object
- **driver** (`str`) – driver for `h5py` to use when opening HDF5 file
- **herd\_path** (`str`) – The path to read/write the HERD file

## 6.2 hdmf.container module

**class** `hdmf.container.HERDManager`

Bases: `object`

This class manages whether to set/attach an instance of HERD to the subclass.

**link\_resources**(*herd*)

Method to attach an instance of HERD in order to auto-add terms/references to data.

**Parameters**

**herd** (`HERD`) – The external resources to be used for the container.

**get\_linked\_resources**()

**class** `hdmf.container.AbstractContainer`(*name*)

Bases: `object`

**Parameters**

**name** (`str`) – the name of this container

**property data\_type**

Return the spec data type associated with this container.

**classmethod** `get_fields_conf`()

**property read\_io**

The `HDMFIO` object used for reading the container.

This property will typically be `None` if this Container is not a root Container (i.e., if *parent* is not `None`). Use `get_read_io` instead if you want to retrieve the `HDMFIO` object used for reading from the parent container.

**get\_read\_io**()

Get the io object used to read this container.

If *self.read\_io* is `None`, this function will iterate through the parents and return the first *io* object found on a parent container

**Returns**

The `HDMFIO` object used to read this container. Returns `None` in case no io object is found, e.g., in case this container has not been read from file.

**property name**

The name of this Container

**get\_ancestor**(*data\_type=None*)

Traverse parent hierarchy and return first instance of the specified *data\_type*

**Parameters**

**data\_type** (*str*) – the *data\_type* to search for

**all\_children**()

Get a list of all child objects and their child objects recursively.

If the object has an *object\_id*, the object will be added to “ret” to be returned. If that object has children, they will be added to the “stack” in order to be: 1) Checked to see if has an *object\_id*, if so then add to “ret” 2) Have children that will also be checked

**property all\_objects**

Get a LabelledDict that indexed all child objects and their children by object ID.

**get\_ancestors**()

**property fields**

Subclasses use this class attribute to add properties to autogenerate. *fields* allows for lists and for dicts with the keys {‘name’, ‘child’, ‘required\_name’, ‘doc’, ‘settable’}. 1. *name*: The name of the field property 2. *child*: A boolean value to set the parent/child relationship between the field property and the container. 3. *required\_name*: The name the field property must have such that *name* matches *required\_name*. 4. *doc*: Documentation of the field property 5. *settable*: If true, a setter function is created so that the field can be changed after creation.

**property object\_id**

**generate\_new\_id**(*recurse=True*)

Changes the object ID of this Container and all of its children to a new UUID string.

**Parameters**

**recurse** (*bool*) – whether or not to change the object ID of this container’s children

**property modified**

**set\_modified**(*modified=True*)

**Parameters**

**modified** (*bool*) – whether or not this Container has been modified

**property children**

**add\_child**(*child=None*)

**Parameters**

**child** (Container) – the child Container for this Container

**classmethod type\_hierarchy**()

**property container\_source**

The source of this Container

**property parent**

The parent Container of this Container

**reset\_parent**()

Reset the parent of this Container to None and remove the Container from the children of its parent.

Use with caution. This can result in orphaned containers and broken links.

**class** hdmf.container.Container(*name*)

Bases: [\*AbstractContainer\*](#)

A container that can contain other containers and has special functionality for printing.

**Parameters**

**name** (*str*) – the name of this container

**property** **css\_style**: *str*

CSS styles for the HTML representation.

**property** **js\_script**: *str*

JavaScript for the HTML representation.

**set\_data\_io**(*dataset\_name: str, data\_io\_class: Type[DataIO], data\_io\_kwargs: dict | None = None, \*\*kwargs*)

Apply DataIO object to a dataset field of the Container.

**Parameters**

- **dataset\_name** (*str*) – Name of dataset to wrap in DataIO
- **data\_io\_class** (*Type[DataIO]*) – Class to use for DataIO, e.g. [H5DataIO](#) or [ZarrDataIO](#)
- **data\_io\_kwargs** (*dict*) – keyword arguments passed to the constructor of the DataIO class.
- **\*\*kwargs** – DEPRECATED. Use `data_io_kwargs` instead. `kwargs` are passed to the constructor of the DataIO class.

**data\_type** = 'Container'

**namespace** = 'hdmf-common'

**class** hdmf.container.Data(*name, data*)

Bases: [\*AbstractContainer\*](#)

A class for representing dataset containers

**Parameters**

- **name** (*str*) – the name of this container
- **data** (*str* or *int* or *float* or *bytes* or *bool* or *ndarray* or *list* or *tuple* or [Dataset](#) or [Array](#) or [StrDataset](#) or [HDMFDataset](#) or [AbstractDataChunkIterator](#) or [DataIO](#)) – the source of the data

**property** **data**

**property** **shape**

Get the shape of the data represented by this container :return: Shape tuple :rtype: tuple of ints

**set\_dataio**(*dataio*)

Apply DataIO object to the data held by this Data object

**Parameters**

**dataio** ([DataIO](#)) – the DataIO to apply to the data held by this Data

**set\_data\_io**(*data\_io\_class: Type[DataIO], data\_io\_kwargs: dict*) → *None*

Apply DataIO object to the data held by this Data object.

**Parameters**

- **data\_io\_class** (*Type* [*DataIO*]) – The DataIO to apply to the data held by this Data.
- **data\_io\_kwargs** (*dict*) – The keyword arguments to pass to the DataIO.

**transform**(*func*)

Transform data from the current underlying state.

This function can be used to permanently load data from disk, or convert to a different representation, such as a torch.Tensor

**Parameters**

**func** (function) – a function to transform *data*

**\_\_getitem\_\_**(*args*)

**get**(*args*)

**append**(*arg*)

**extend**(*arg*)

The extend\_data method adds all the elements of the iterable arg to the end of the data of this Data container.

**Parameters**

**arg** – The iterable to add to the end of this VectorData

**data\_type** = 'Data'

**namespace** = 'hdmf-common'

**class** hdmf.container.DataRegion(*name*, *data*)

Bases: *Data*

**Parameters**

- **name** (*str*) – the name of this container
- **data** (*str* or *int* or *float* or *bytes* or *bool* or *ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the source of the data

**abstract property data**

The target data that this region applies to

**abstract property region**

The region that indexes into data e.g. slice or list of indices

**class** hdmf.container.MultiContainerInterface(*name*)

Bases: *Container*

Class that dynamically defines methods to support a Container holding multiple Containers of the same type.

To use, extend this class and create a dictionary as a class attribute with any of the following keys: \* 'attr' to name the attribute that stores the Container instances \* 'type' to provide the Container object type (type or list/tuple of types, type can be a docval macro) \* 'add' to name the method for adding Container instances \* 'get' to name the method for getting Container instances \* 'create' to name the method for creating Container instances (only if a single type is specified)

If the attribute does not exist in the class, it will be generated. If it does exist, it should behave like a dict.

The keys 'attr', 'type', and 'add' are required.

**Parameters**

**name** (*str*) – the name of this container

**class** `hdmf.container.Row`

Bases: `object`

A class for representing rows from a Table.

The Table class can be indicated with the `__table__`. Doing so will set constructor arguments for the Row class and ensure that Row.idx is set appropriately when a Row is added to the Table. It will also add functionality to the Table class for getting Row objects.

Note, the Row class is not needed for working with Table objects. This is merely convenience functionality for working with Tables.

**property idx**

The index of this row in its respective Table

**property table**

The Table this Row comes from

**class** `hdmf.container.RowGetter(table)`

Bases: `object`

A simple class for providing `__getitem__` functionality that returns Row objects to a Table.

**\_\_getitem\_\_** (*idx*)

**class** `hdmf.container.Table(columns, name, data=[])`

Bases: `Data`

Subclasses should specify the class attribute `__columns__`.

This should be a list of dictionaries with the following keys:

- **name** the column name
- **type** the type of data in this column
- **doc** a brief description of what gets stored in this column

For reference, this list of dictionaries will be used with `docval` to autogenerate the `add_row` method for adding data to this table.

If `__columns__` is not specified, no custom `add_row` method will be added.

The class attribute `__defaultname__` can also be set to specify a default name for the table class. If `__defaultname__` is not specified, then **name** will need to be specified when the class is instantiated.

A Table class can be paired with a Row class for conveniently working with rows of a Table. This pairing must be indicated in the Row class implementation. See Row for more details.

**Parameters**

- **columns** (*list* or *tuple*) – a list of the columns in this table
- **name** (*str*) – the name of this container
- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the source of the data

**property columns**



**add\_row**(*values*)

**Parameters**

**values** (*dict*) – the values for each column

**which**(\*\**kwargs*)

Query a table

**\_\_getitem\_\_**(*args*)

**to\_dataframe**()

Produce a pandas DataFrame containing this table's data.

**classmethod from\_dataframe**(*df*, *name=None*, *extra\_ok=False*)

**Construct an instance of Table (or a subclass) from a pandas DataFrame. The columns of the dataframe**

should match the columns defined on the Table subclass.

**Parameters**

- **df** (*DataFrame*) – input data
- **name** (*str*) – the name of this container
- **extra\_ok** (*bool*) – accept (and ignore) unexpected columns on the input dataframe

## 6.3 hdmf.build package

### 6.3.1 Submodules

#### **hdmf.build.builders module**

**class** hdmf.build.builders.**Builder**(*name*, *parent=None*, *source=None*)

Bases: *dict*

**Parameters**

- **name** (*str*) – the name of the group
- **parent** (*Builder*) – the parent builder of this Builder
- **source** (*str*) – the source of the data in this builder e.g. file name

**property path**

The path of this builder.

**property name**

The name of this builder.

**property source**

The source of this builder.

**property parent**

The parent builder of this builder.

```
class hdmf.build.builders.BaseBuilder(name, attributes={}, parent=None, source=None)
```

Bases: *Builder*

#### Parameters

- **name** (*str*) – The name of the builder.
- **attributes** (*dict*) – A dictionary of attributes to create in this builder.
- **parent** (*GroupBuilder*) – The parent builder of this builder.
- **source** (*str*) – The source of the data represented in this builder

#### property location

The location of this Builder in its source.

#### property attributes

The attributes stored in this Builder object.

```
set_attribute(name, value)
```

Set an attribute for this group.

#### Parameters

- **name** (*str*) – The name of the attribute.
- **value** (*None*) – The attribute value.

```
class hdmf.build.builders.GroupBuilder(name, groups={}, datasets={}, attributes={}, links={},
                                       parent=None, source=None)
```

Bases: *BaseBuilder*

Create a builder object for a group.

#### Parameters

- **name** (*str*) – The name of the group.
- **groups** (*dict* or *list*) – A dictionary or list of subgroups to add to this group. If a dict is provided, only the values are used.
- **datasets** (*dict* or *list*) – A dictionary or list of datasets to add to this group. If a dict is provided, only the values are used.
- **attributes** (*dict*) – A dictionary of attributes to create in this group.
- **links** (*dict* or *list*) – A dictionary or list of links to add to this group. If a dict is provided, only the values are used.
- **parent** (*GroupBuilder*) – The parent builder of this builder.
- **source** (*str*) – The source of the data represented in this builder.

#### property source

The source of this Builder

#### property groups

The subgroups contained in this group.

#### property datasets

The datasets contained in this group.

#### property links

The links contained in this group.

**set\_attribute**(*name*, *value*)

Set an attribute for this group.

**Parameters**

- **name** (*str*) – The name of the attribute.
- **value** (*None*) – The attribute value.

**set\_group**(*builder*)

Add a subgroup to this group.

**Parameters**

**builder** (*GroupBuilder*) – The GroupBuilder to add to this group.

**set\_dataset**(*builder*)

Add a dataset to this group.

**Parameters**

**builder** (*DatasetBuilder*) – The DatasetBuilder to add to this group.

**set\_link**(*builder*)

Add a link to this group.

**Parameters**

**builder** (*LinkBuilder*) – The LinkBuilder to add to this group.

**is\_empty**()

Returns true if there are no datasets, links, attributes, and non-empty subgroups. False otherwise.

**\_\_getitem\_\_**(*key*)

Like dict.\_\_getitem\_\_, but looks in groups, datasets, attributes, and links sub-dictionaries. Key can be a posix path to a sub-builder.

**get**(*key*, *default=None*)

Like dict.get, but looks in groups, datasets, attributes, and links sub-dictionaries. Key can be a posix path to a sub-builder.

**items**()

Like dict.items, but iterates over items in groups, datasets, attributes, and links sub-dictionaries.

**keys**()

Like dict.keys, but iterates over keys in groups, datasets, attributes, and links sub-dictionaries.

**values**()

Like dict.values, but iterates over values in groups, datasets, attributes, and links sub-dictionaries.

**class** hdmf.build.builders.**DatasetBuilder**(*name*, *data=None*, *dtype=None*, *attributes={}*,  
*maxshape=None*, *chunks=False*, *parent=None*, *source=None*)

Bases: *BaseBuilder*

Create a Builder object for a dataset

**Parameters**

- **name** (*str*) – The name of the dataset.
- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Array* or *StrDataset* or *HDMFDataset* or *AbstractDataChunkIterator* or *str* or *int* or *float* or *bytes* or *bool* or *DataIO* or *DatasetBuilder* or *RegionBuilder* or *Iterable* or *datetime* or *date*) – The data in this dataset.

- **dtype** (*type* or *dtype* or *str* or *list*) – The datatype of this dataset.
- **attributes** (*dict*) – A dictionary of attributes to create in this dataset.
- **maxshape** (*int* or *tuple*) – The shape of this dataset. Use None for scalars.
- **chunks** (*bool*) – Whether or not to chunk this dataset.
- **parent** (*GroupBuilder*) – The parent builder of this builder.
- **source** (*str*) – The source of the data in this builder.

**OBJECT\_REF\_TYPE** = 'object'

**REGION\_REF\_TYPE** = 'region'

**property data**

The data stored in the dataset represented by this builder.

**property chunks**

Whether or not this dataset is chunked.

**property maxshape**

The max shape of this dataset.

**property dtype**

The data type of this dataset.

**class** hdmf.build.builders.**LinkBuilder**(*builder*, *name=None*, *parent=None*, *source=None*)

Bases: *Builder*

Create a builder object for a link.

**Parameters**

- **builder** (*DatasetBuilder* or *GroupBuilder*) – The target group or dataset of this link.
- **name** (*str*) – The name of the link
- **parent** (*GroupBuilder*) – The parent builder of this builder
- **source** (*str*) – The source of the data in this builder

**property builder**

The target builder object.

**class** hdmf.build.builders.**ReferenceBuilder**(*builder*)

Bases: *dict*

Create a builder object for a reference.

**Parameters**

**builder** (*DatasetBuilder* or *GroupBuilder*) – The group or dataset this reference applies to.

**property builder**

The target builder object.

**class** hdmf.build.builders.**RegionBuilder**(*region*, *builder*)

Bases: *ReferenceBuilder*

Create a builder object for a region reference.

**Parameters**

- **region** (*slice* or *tuple* or *list* or *RegionReference*) – The region, i.e. slice or indices, into the target dataset.
- **builder** (*DatasetBuilder*) – The dataset this region reference applies to.

**property region**

The selected region of the target dataset.

## hdmf.build.classgenerator module

**class** hdmf.build.classgenerator.**ClassGenerator**

Bases: *object*

**property** custom\_generators

**register\_generator**(*generator*)

Add a custom class generator to this ClassGenerator.

Generators added later are run first. Duplicates are moved to the top of the list.

**Parameters**

**generator** (*type*) – the CustomClassGenerator class to register

**generate\_class**(*data\_type*, *spec*, *parent\_cls*, *attr\_names*, *type\_map*)

**Get the container class from data type specification.**

If no class has been associated with the *data\_type* from namespace, a class will be dynamically created and returned.

**Parameters**

- **data\_type** (*str*) – the data type to create a AbstractContainer class for
- **spec** (*BaseStorageSpec*)
- **parent\_cls** (*type*)
- **attr\_names** (*dict*)
- **type\_map** (*TypeMap*)

**Returns**

the class for the given namespace and *data\_type*

**Return type**

*type*

**exception** hdmf.build.classgenerator.**TypeDoesNotExistError**

Bases: *Exception*

**class** hdmf.build.classgenerator.**CustomClassGenerator**(\*args, \*\*kwargs)

Bases: *object*

Subclass this class and register an instance to alter how classes are auto-generated.

**classmethod** **apply\_generator\_to\_field**(*field\_spec*, *bases*, *type\_map*)

Return True to signal that this generator should return on all fields not yet processed.

**classmethod** `process_field_spec(classdict, docval_args, parent_cls, attr_name, not_inherited_fields, type_map, spec)`

Add `__fields__` to the classdict and update the docval args for the field spec with the given attribute name. :param classdict: The dict to update with `__fields__` (or a different `parent_cls._fieldsname`). :param docval\_args: The list of docval arguments. :param parent\_cls: The parent class. :param attr\_name: The attribute name of the field spec for the container class to generate. :param not\_inherited\_fields: Dictionary of fields not inherited from the parent class. :param type\_map: The type map to use. :param spec: The spec for the container class to generate.

**classmethod** `post_process(classdict, bases, docval_args, spec)`

Convert classdict[`'__fields__'`] to tuple and update docval args for a fixed name and default name. :param classdict: The class dictionary to convert with `'__fields__'` key (or a different `bases[0]._fieldsname`). :param bases: The list of base classes. :param docval\_args: The dict of docval arguments. :param spec: The spec for the container class to generate.

**classmethod** `set_init(classdict, bases, docval_args, not_inherited_fields, name)`

**class** `hdmf.build.classgenerator.MCIClassGenerator(*args, **kwargs)`

Bases: [`CustomClassGenerator`](#)

**classmethod** `apply_generator_to_field(field_spec, bases, type_map)`

Return True if the field spec has quantity `*` or `+`, False otherwise.

**classmethod** `process_field_spec(classdict, docval_args, parent_cls, attr_name, not_inherited_fields, type_map, spec)`

Add `__clsconf__` to the classdict and update the docval args for the field spec with the given attribute name. :param classdict: The dict to update with `__clsconf__`. :param docval\_args: The list of docval arguments. :param parent\_cls: The parent class. :param attr\_name: The attribute name of the field spec for the container class to generate. :param not\_inherited\_fields: Dictionary of fields not inherited from the parent class. :param type\_map: The type map to use. :param spec: The spec for the container class to generate.

**classmethod** `post_process(classdict, bases, docval_args, spec)`

Add `MultiContainerInterface` to the list of base classes. :param classdict: The class dictionary. :param bases: The list of base classes. :param docval\_args: The dict of docval arguments. :param spec: The spec for the container class to generate.

**classmethod** `set_init(classdict, bases, docval_args, not_inherited_fields, name)`

## **hdmf.build.errors module**

Module for build error definitions

**exception** `hdmf.build.errors.BuildError(builder, reason)`

Bases: [`Exception`](#)

Error raised when building a container into a builder.

### **Parameters**

- **builder** ([`Builder`](#)) – the builder that cannot be built
- **reason** (`str`) – the reason for the error

**exception** `hdmf.build.errors.OrphanContainerBuildError(builder, container)`

Bases: [`BuildError`](#)

### **Parameters**

- **builder** (*Builder*) – the builder containing the broken link
- **container** (*AbstractContainer*) – the container that has no parent

**exception** `hdmf.build.errors.ReferenceTargetNotBuiltError`(*builder*, *container*)

Bases: *BuildError*

#### Parameters

- **builder** (*Builder*) – the builder containing the reference that cannot be found
- **container** (*AbstractContainer*) – the container that is not built yet

**exception** `hdmf.build.errors.ContainerConfigurationError`

Bases: *Exception*

Error raised when the container class is improperly configured.

**exception** `hdmf.build.errors.ConstructError`

Bases: *Exception*

Error raised when constructing a container from a builder.

### `hdmf.build.manager` module

**class** `hdmf.build.manager.Proxy`(*manager*, *source*, *location*, *namespace*, *data\_type*)

Bases: *object*

A temporary object to represent a Container. This gets used when resolving the true location of a Container's parent. Proxy objects allow simple bookkeeping of all potential parents a Container may have. This object is used by providing all the necessary information for describing the object. This object gets passed around and candidates are accumulated. Upon calling `resolve`, all saved candidates are matched against the information (provided to the constructor). The candidate that has an exact match is returned.

#### **property** `source`

The source of the object e.g. file source

#### **property** `location`

The location of the object. This can be thought of as a unique path

#### **property** `namespace`

The namespace from which the `data_type` of this Proxy came from

#### **property** `data_type`

The `data_type` of Container that should match this Proxy

**matches**(*object*)

#### Parameters

**object** (*BaseBuilder* or *Container*) – the container or builder to get a proxy for

**add\_candidate**(*container*)

#### Parameters

**container** (*Container*) – the Container to add as a candidate match

**resolve**()

**class** hdmf.build.manager.**BuildManager**(*type\_map*)

Bases: **object**

A class for managing builds of AbstractContainers

**property** namespace\_catalog

**property** type\_map

**get\_proxy**(*object*, *source=None*)

**Parameters**

- **object** (*BaseBuilder* or *AbstractContainer*) – the container or builder to get a proxy for
- **source** (*str*) – the source of container being built i.e. file path

**build**(*container*, *source=None*, *spec\_ext=None*, *export=False*, *root=False*)

Build the GroupBuilder/DatasetBuilder for the given AbstractContainer

**Parameters**

- **container** (*AbstractContainer*) – the container to convert to a Builder
- **source** (*str*) – the source of container being built i.e. file path
- **spec\_ext** (*BaseStorageSpec*) – a spec that further refines the base specification
- **export** (*bool*) – whether this build is for exporting
- **root** (*bool*) – whether the container is the root of the build process

**prebuilt**(*container*, *builder*)

Save the Builder for a given AbstractContainer for future use

**Parameters**

- **container** (*AbstractContainer*) – the AbstractContainer to save as prebuilt
- **builder** (*DatasetBuilder* or *GroupBuilder*) – the Builder representation of the given container

**queue\_ref**(*func*)

Set aside creating ReferenceBuilders

**purge\_outdated**()

**clear\_cache**()

**get\_builder**(*container*)

Return the prebuilt builder for the given container or None if it does not exist.

**Parameters**

- **container** (*AbstractContainer*) – the container to get the builder for

**construct**(*builder*)

Construct the AbstractContainer represented by the given builder

**Parameters**

- **builder** (*DatasetBuilder* or *GroupBuilder*) – the builder to construct the AbstractContainer from



**get\_cls**(*builder*)

Get the class object for the given Builder

**Parameters**

**builder** (*Builder*) – the Builder to get the class object for

**get\_builder\_name**(*container*)

Get the name a Builder should be given

**Parameters**

**container** (*AbstractContainer*) – the container to convert to a Builder

**Returns**

The name a Builder should be given when building this container

**Return type**

*str*

**get\_subspec**(*spec, builder*)

Get the specification from this spec that corresponds to the given builder

**Parameters**

- **spec** (*DatasetSpec* or *GroupSpec*) – the parent spec to search
- **builder** (*DatasetBuilder* or *GroupBuilder* or *LinkBuilder*) – the builder to get the sub-specification for

**get\_builder\_ns**(*builder*)

Get the namespace of a builder

**Parameters**

**builder** (*DatasetBuilder* or *GroupBuilder* or *LinkBuilder*) – the builder to get the sub-specification for

**get\_builder\_dt**(*builder*)

Get the data\_type of a builder

**Parameters**

**builder** (*DatasetBuilder* or *GroupBuilder* or *LinkBuilder*) – the builder to get the data\_type for

**is\_sub\_data\_type**(*builder, parent\_data\_type*)

Return whether or not data\_type of *builder* is a sub-data\_type of *parent\_data\_type*

**Parameters**

- **builder** (*GroupBuilder* or *DatasetBuilder* or *AbstractContainer*) – the builder or container to check
- **parent\_data\_type** (*str*) – the potential parent data\_type that refers to a data\_type

**Returns**

True if data\_type of *builder* is a sub-data\_type of *parent\_data\_type*, False otherwise

**Return type**

*bool*

**class** hdmf.build.manager.**TypeSource**(*namespace, data\_type*)

Bases: *object*

A class to indicate the source of a data\_type in a namespace. This class should only be used by TypeMap

**Parameters**

- **namespace** (*str*) – the namespace the from, which the data\_type originated
- **data\_type** (*str*) – the name of the type

**property namespace**

**property data\_type**

**class** hdmf.build.manager.**TypeMap**(*namespaces=None, mapper\_cls=None, type\_config=None*)

Bases: *object*

A class to maintain the map between ObjectMappers and AbstractContainer classes

**Parameters**

- **namespaces** (*NamespaceCatalog*) – the NamespaceCatalog to use
- **mapper\_cls** (*type*) – the ObjectMapper class to use
- **type\_config** (*TypeConfigurator*) – The TypeConfigurator to use.

**property namespace\_catalog**

**property container\_types**

**copy\_mappers**(*type\_map*)

**merge**(*type\_map, ns\_catalog=False*)

**register\_generator**(*generator*)

Add a custom class generator.

**Parameters**

- **generator** (*type*) – the CustomClassGenerator class to register

**load\_namespaces**(*namespace\_path, resolve=True, reader=None*)

**Load namespaces from a namespace file.**

This method will call `load_namespaces` on the `NamespaceCatalog` used to construct this `TypeMap`. Additionally, it will process the return value to keep track of what types were included in the loaded namespaces. Calling `load_namespaces` here has the advantage of being able to keep track of type dependencies across namespaces.

**Parameters**

- **namespace\_path** (*str*) – the path to the file containing the namespaces(s) to load
- **resolve** (*bool*) – whether or not to include objects from included/parent spec objects
- **reader** (*SpecReader*) – the class to user for reading specifications

**Returns**

the namespaces loaded from the given file

**Return type**

*dict*

**get\_container\_cls**(*namespace, data\_type, autogen=True*)

**Get the container class from data type specification.**

If no class has been associated with the `data_type` from `namespace`, a class will be dynamically created and returned.

**Parameters**

- **namespace** (`str`) – the namespace containing the `data_type`
- **data\_type** (`str`) – the data type to create a `AbstractContainer` class for
- **autogen** (`bool`) – autogenerate class if one does not exist

**Returns**

the class for the given namespace and `data_type`

**Return type**

`type`

**get\_dt\_container\_cls**(*data\_type*, *namespace=None*, *autogen=True*)

**Get the container class from data type specification.**

If no class has been associated with the `data_type` from namespace, a class will be dynamically created and returned.

Replaces `get_container_cls` but namespace is optional. If namespace is unknown, it will be looked up from all namespaces.

**Parameters**

- **data\_type** (`str`) – the data type to create a `AbstractContainer` class for
- **namespace** (`str`) – the namespace containing the `data_type`
- **autogen** (`bool`) – autogenerate class if one does not exist

**Returns**

the class for the given namespace and `data_type`

**Return type**

`type`

**get\_builder\_dt**(*builder*)

Get the `data_type` of a builder

**Parameters**

**builder** (`DatasetBuilder` or `GroupBuilder` or `LinkBuilder`) – the builder to get the `data_type` for

**get\_builder\_ns**(*builder*)

Get the namespace of a builder

**Parameters**

**builder** (`DatasetBuilder` or `GroupBuilder` or `LinkBuilder`) – the builder to get the sub-specification for

**get\_cls**(*builder*)

Get the class object for the given Builder

**Parameters**

**builder** (`Builder`) – the Builder object to get the corresponding `AbstractContainer` class for

**get\_subspec**(*spec*, *builder*)

Get the specification from this spec that corresponds to the given builder

**Parameters**

- **spec** (*DatasetSpec* or *GroupSpec*) – the parent spec to search
- **builder** (*DatasetBuilder* or *GroupBuilder* or *LinkBuilder*) – the builder to get the sub-specification for

**get\_container\_ns\_dt**(*obj*)

**get\_container\_cls\_dt**(*cls*)

**get\_container\_classes**(*namespace=None*)

**Parameters**

**namespace** (*str*) – the namespace to get the container classes for

**get\_map**(*obj*)

Return the ObjectMapper object that should be used for the given container

**Parameters**

**obj** (*AbstractContainer* or *Builder*) – the object to get the ObjectMapper for

**Returns**

the ObjectMapper to use for mapping the given object

**Return type**

ObjectMapper

**register\_container\_type**(*namespace, data\_type, container\_cls*)

Map a container class to a data\_type

**Parameters**

- **namespace** (*str*) – the namespace containing the data\_type to map the class to
- **data\_type** (*str*) – the data\_type to map the class to
- **container\_cls** (*TypeSource* or *type*) – the class to map to the specified data\_type

**register\_map**(*container\_cls, mapper\_cls*)

Map a container class to an ObjectMapper class

**Parameters**

- **container\_cls** (*type*) – the AbstractContainer class for which the given ObjectMapper class gets used for
- **mapper\_cls** (*type*) – the ObjectMapper class to use to map

**build**(*container, manager=None, source=None, builder=None, spec\_ext=None, export=False*)

Build the GroupBuilder/DatasetBuilder for the given AbstractContainer

**Parameters**

- **container** (*AbstractContainer*) – the container to convert to a Builder
- **manager** (*BuildManager*) – the BuildManager to use for managing this build
- **source** (*str*) – the source of container being built i.e. file path
- **builder** (*BaseBuilder*) – the Builder to build on
- **spec\_ext** (*BaseStorageSpec*) – a spec extension
- **export** (*bool*) – whether this build is for exporting

**construct**(*builder*, *build\_manager=None*, *parent=None*)

Construct the AbstractContainer represented by the given builder

**Parameters**

- **builder** (*DatasetBuilder* or *GroupBuilder*) – the builder to construct the Abstract-Container from
- **build\_manager** (*BuildManager*) – the BuildManager for constructing
- **parent** (*Proxy* or *Container*) – the parent Container/Proxy for the Container being built

**get\_builder\_name**(*container*)

Get the name a Builder should be given

**Parameters**

**container** (*AbstractContainer*) – the container to convert to a Builder

**Returns**

The name a Builder should be given when building this container

**Return type**

*str*

## hdmf.build.map module

## hdmf.build.objectmapper module

**class** hdmf.build.objectmapper.**ObjectMapper**(*spec*)

Bases: *object*

A class for mapping between Spec objects and AbstractContainer attributes

Create a map from AbstractContainer attributes to specifications

**Parameters**

**spec** (*DatasetSpec* or *GroupSpec*) – The specification for mapping objects to builders

**classmethod** **no\_convert**(*obj\_type*)

Specify an object type that ObjectMappers should not convert.

**classmethod** **convert\_dtype**(*spec*, *value*, *spec\_dtype=None*)

Convert values to the specified dtype. For example, if a literal int is passed in to a field that is specified as a unsigned integer, this function will convert the Python int to a numpy unsigned int.

**Parameters**

- **spec** – The DatasetSpec or AttributeSpec to which this value is being applied
- **value** – The value being converted to the spec dtype
- **spec\_dtype** – Optional override of the dtype in spec.dtype. Used to specify the parent dtype when the given extended spec lacks a dtype.

**Returns**

The function returns a tuple consisting of 1) the value, and 2) the data type. The value is returned as the function may convert the input value to comply with the dtype specified in the schema.

**static constructor\_arg(*name*)**

Decorator to override the default mapping scheme for a given constructor argument.

Decorate ObjectMapper methods with this function when extending ObjectMapper to override the default scheme for mapping between AbstractContainer and Builder objects. The decorated method should accept as its first argument the Builder object that is being mapped. The method should return the value to be passed to the target AbstractContainer class constructor argument given by *name*.

**Parameters**

**name** (*str*) – the name of the constructor argument

**static object\_attr(*name*)**

Decorator to override the default mapping scheme for a given object attribute.

Decorate ObjectMapper methods with this function when extending ObjectMapper to override the default scheme for mapping between AbstractContainer and Builder objects. The decorated method should accept as its first argument the AbstractContainer object that is being mapped. The method should return the child Builder object (or scalar if the object attribute corresponds to an AttributeSpec) that represents the attribute given by *name*.

**Parameters**

**name** (*str*) – the name of the constructor argument

**property spec**

the Spec used in this ObjectMapper

**get\_container\_name(\*args)****classmethod convert\_dt\_name(*spec*)**

Construct the attribute name corresponding to a specification

**Parameters**

**spec** (*Spec*) – the specification to get the name for

**classmethod get\_attr\_names(*spec*)**

Get the attribute names for each subspecification in a Spec

**Parameters**

**spec** (*Spec*) – the specification to get the object attribute names for

**map\_attr(*attr\_name*, *spec*)**

Map an attribute to spec. Use this to override default behavior

**Parameters**

- **attr\_name** (*str*) – the name of the object to map
- **spec** (*Spec*) – the spec to map the attribute to

**get\_attr\_spec(*attr\_name*)**

Return the Spec for a given attribute

**Parameters**

**attr\_name** (*str*) – the name of the attribute

**get\_carg\_spec**(*carg\_name*)

Return the Spec for a given constructor argument

**Parameters**

**carg\_name** (*str*) – the name of the constructor argument

**map\_const\_arg**(*const\_arg, spec*)

Map an attribute to spec. Use this to override default behavior

**Parameters**

- **const\_arg** (*str*) – the name of the constructor argument to map
- **spec** (*Spec*) – the spec to map the attribute to

**unmap**(*spec*)

Removing any mapping for a specification. Use this to override default mapping

**Parameters**

**spec** (*Spec*) – the spec to map the attribute to

**map\_spec**(*attr\_carg, spec*)

Map the given specification to the construct argument and object attribute

**Parameters**

- **attr\_carg** (*str*) – the constructor argument/object attribute to map this spec to
- **spec** (*Spec*) – the spec to map the attribute to

**get\_attribute**(*spec*)

Get the object attribute name for the given Spec

**Parameters**

**spec** (*Spec*) – the spec to get the attribute for

**Returns**

the attribute name

**Return type**

*str*

**get\_attr\_value**(*spec, container, manager*)

Get the value of the attribute corresponding to this spec from the given container

**Parameters**

- **spec** (*Spec*) – the spec to get the attribute value for
- **container** (*AbstractContainer*) – the container to get the attribute value from
- **manager** (*BuildManager*) – the BuildManager used for managing this build

**get\_const\_arg**(*spec*)

Get the constructor argument for the given Spec

**Parameters**

**spec** (*Spec*) – the spec to get the constructor argument for

**Returns**

the name of the constructor argument

**Return type**

*str*

**build**(*container, manager, parent=None, source=None, builder=None, spec\_ext=None, export=False*)

Convert an AbstractContainer to a Builder representation.

References are not added but are queued to be added in the BuildManager.

#### Parameters

- **container** (*AbstractContainer*) – the container to convert to a Builder
- **manager** (*BuildManager*) – the BuildManager to use for managing this build
- **parent** (*GroupBuilder*) – the parent of the resulting Builder
- **source** (*str*) – the source of container being built i.e. file path
- **builder** (*BaseBuilder*) – the Builder to build on
- **spec\_ext** (*BaseStorageSpec*) – a spec extension
- **export** (*bool*) – whether this build is for exporting

#### Returns

the Builder representing the given AbstractContainer

#### Return type

*Builder*

**construct**(*builder, manager, parent=None*)

Construct an AbstractContainer from the given Builder

#### Parameters

- **builder** (*DatasetBuilder* or *GroupBuilder*) – the builder to construct the Abstract-Container from
- **manager** (*BuildManager*) – the BuildManager for this build
- **parent** (*Proxy* or *AbstractContainer*) – the parent AbstractContainer/Proxy for the AbstractContainer being built

**get\_builder\_name**(*container*)

Get the name of a Builder that represents a AbstractContainer

#### Parameters

**container** (*AbstractContainer*) – the AbstractContainer to get the Builder name for

```
constructor_args = {'name': <function ObjectMapper.get_container_name>}
```

```
obj_attrs = {}
```

## hdmf.build.warnings module

Module for build warnings

**exception** `hdmf.build.warnings.BuildWarning`

Bases: `UserWarning`

Base class for warnings that are raised during the building of a container.



**exception** `hdmf.build.warnings.IncorrectQuantityBuildWarning`

Bases: [\*BuildWarning\*](#)

Raised when a container field contains a number of groups/datasets/links that is not allowed by the spec.

**exception** `hdmf.build.warnings.MissingRequiredBuildWarning`

Bases: [\*BuildWarning\*](#)

Raised when a required field is missing.

**exception** `hdmf.build.warnings.MissingRequiredWarning`

Bases: [\*MissingRequiredBuildWarning\*](#)

Raised when a required field is missing.

**exception** `hdmf.build.warnings.OrphanContainerWarning`

Bases: [\*BuildWarning\*](#)

Raised when a container is built without a parent.

**exception** `hdmf.build.warnings.DtypeConversionWarning`

Bases: [\*UserWarning\*](#)

Raised when a value is converted to a different data type in order to match the specification.

## 6.3.2 Module contents

# 6.4 hdmf.spec package

## 6.4.1 Submodules

### `hdmf.spec.catalog` module

**class** `hdmf.spec.catalog.SpecCatalog`

Bases: [\*object\*](#)

Create a new catalog for storing specifications

**\*\* Private Instance Variables \*\***

#### Variables

- **`__specs`** – Dict with the specification of each registered type
- **`__parent_types`** – Dict with parent types for each registered type
- **`__spec_source_files`** – Dict with the path to the source files (if available) for each registered type
- **`__hierarchy`** – Dict describing the hierarchy for each registered type. NOTE: Always use `SpecCatalog.get_hierarchy(...)` to retrieve the hierarchy as this dictionary is used like a cache, i.e., to avoid repeated calculation of the hierarchy but the contents are computed on first request by `SpecCatalog.get_hierarchy(...)`

**`register_spec(spec, source_file=None)`**

Associate a specified object type with a specification

#### Parameters

- **`spec`** ([\*BaseStorageSpec\*](#)) – a Spec object

- **source\_file** (*str*) – path to the source file from which the spec was loaded

**get\_spec**(*data\_type*)

Get the Spec object for the given type

**Parameters**

**data\_type** (*str*) – the data\_type to get the Spec for

**Returns**

the specification for writing the given object type to HDF5

**Return type**

*Spec*

**get\_registered\_types**()

Return all registered specifications

**get\_spec\_source\_file**(*data\_type*)

**Return the path to the source file from which the spec for the given**

type was loaded from. None is returned if no file path is available for the spec. Note: The spec in the file may not be identical to the object in case the spec is modified after load.

**Parameters**

**data\_type** (*str*) – the data\_type of the spec to get the source file for

**Returns**

the path to source specification file from which the spec was originally loaded or None

**Return type**

*str*

**auto\_register**(*spec*, *source\_file=None*)

Register this specification and all sub-specification using data\_type as object type name

**Parameters**

- **spec** (*BaseStorageSpec*) – the Spec object to register
- **source\_file** (*str*) – path to the source file from which the spec was loaded

**Returns**

the types that were registered with this spec

**Return type**

*tuple*

**get\_hierarchy**(*data\_type*)

For a given type get the type inheritance hierarchy for that type.

E.g., if we have a type MyContainer that inherits from BaseContainer then the result will be a tuple with the strings ('MyContainer', 'BaseContainer')

**Parameters**

**data\_type** (*str* or *type*) – the data\_type to get the hierarchy of

**Returns**

Tuple of strings with the names of the types the given data\_type inherits from.

**Return type**

*tuple*

**get\_full\_hierarchy()**

Get the complete hierarchy of all types. The function attempts to sort types by name using standard Python sorted.

**Returns**

Hierarchically nested OrderedDict with the hierarchy of all the types

**Return type**

OrderedDict

**get\_subtypes(data\_type, recursive=True)**

For a given data type recursively find all the subtypes that inherit from it.

E.g., assume we have the following inheritance hierarchy:

```

-BaseContainer----->AContainer--->ADContainer
      |
      +-->BContainer
  
```

In this case, the subtypes of BaseContainer would be (AContainer, ADContainer, BContainer), the subtypes of AContainer would be (ADContainer), and the subtypes of BContainer would be empty ().

**Parameters**

- **data\_type** (`str` or `type`) – the data\_type to get the subtypes for
- **recursive** (`bool`) – recursively get all subtypes. Set to False to only get the direct subtypes

**Returns**

Tuple of strings with the names of all types of the given data\_type.

**Return type**

tuple

**hdmf.spec.namespace module**

```
class hdmf.spec.namespace.SpecNamespace(doc, name, schema, full_name=None, version=None,
                                         date=None, author=None, contact=None, catalog=None)
```

Bases: `dict`

A namespace for specifications

**Parameters**

- **doc** (`str`) – a description about what this namespace represents
- **name** (`str`) – the name of this namespace
- **schema** (`list`) – location of schema specification files or other Namespaces
- **full\_name** (`str`) – extended full name of this namespace
- **version** (`str` or `tuple` or `list`) – Version number of the namespace
- **date** (`datetime` or `str`) – Date last modified or released. Formatting is `%Y-%m-%d %H:%M:%S`, e.g, 2017-04-25 17:14:13
- **author** (`str` or `list`) – Author or list of authors.

- **contact** (*str* or *list*) – List of emails. Ordering should be the same as for author
- **catalog** (*SpecCatalog*) – The SpecCatalog object for this SpecNamespace

**UNVERSIONED** = None

**classmethod** **types\_key()**

Get the key used for specifying types to include from a file or namespace

Override this method to use a different name for 'data\_types'

**property** **full\_name**

String with full name or None

**property** **contact**

String or list of strings with the contacts or None

**property** **author**

String or list of strings with the authors or None

**property** **version**

String, list, or tuple with the version or SpecNamespace.UNVERSIONED if the version is missing or empty

**property** **date**

Date last modified or released.

**Returns**

datetime object, string, or None

**property** **name**

String with short name or None

**property** **doc**

**property** **schema**

**get\_source\_files()**

Get the list of names of the source files included the schema of the namespace

**get\_source\_description**(*sourcefile*)

**Get the description of a source file as described in the namespace. The result is a**

dict which contains the 'source' and optionally 'title', 'doc' and 'data\_types' imported from the source file

**Parameters**

**sourcefile** (*str*) – Name of the source file

**Returns**

Dict with the source file documentation

**Return type**

*dict*

**property** **catalog**

The SpecCatalog containing all the Specs

**get\_spec**(*data\_type*)

Get the Spec object for the given data type

**Parameters**

**data\_type** (*str* or *type*) – the data\_type to get the spec for

**get\_registered\_types**()

Get the available types in this namespace

**Returns**

the a tuple of the available data types

**Return type**

*tuple*

**get\_hierarchy**(*data\_type*)

Get the extension hierarchy for the given data\_type in this namespace

**Parameters**

**data\_type** (*str* or *type*) – the data\_type to get the hierarchy of

**Returns**

a tuple with the type hierarchy

**Return type**

*tuple*

**classmethod build\_namespace**(\*\**spec\_dict*)

**class** `hdmf.spec.namespace.SpecReader`(*source*)

Bases: *object*

**Parameters**

**source** (*str*) – the source from which this reader reads from

**property source**

**abstract read\_spec**()

**abstract read\_namespace**()

**class** `hdmf.spec.namespace.YAMLSpecReader`(*indir*='.')

Bases: *SpecReader*

**Parameters**

**indir** (*str*) – the path spec files are relative to

**read\_namespace**(*namespace\_path*)

**read\_spec**(*spec\_path*)

**class** `hdmf.spec.namespace.NamespaceCatalog`(*group\_spec\_cls*=<class 'hdmf.spec.spec.GroupSpec'>, *dataset\_spec\_cls*=<class 'hdmf.spec.spec.DatasetSpec'>, *spec\_namespace\_cls*=<class 'hdmf.spec.namespace.SpecNamespace'>)

Bases: *object*

Create a catalog for storing multiple Namespaces

**Parameters**

- **group\_spec\_cls** (*type*) – the class to use for group specifications

- **dataset\_spec\_cls** (*type*) – the class to use for dataset specifications
- **spec\_namespace\_cls** (*type*) – the class to use for specification namespaces

**merge**(*ns\_catalog*)

**property namespaces**

The namespaces in this NamespaceCatalog

**Returns**

a tuple of the available namespaces

**Return type**

*tuple*

**property dataset\_spec\_cls**

The DatasetSpec class used in this NamespaceCatalog

**property group\_spec\_cls**

The GroupSpec class used in this NamespaceCatalog

**property spec\_namespace\_cls**

The SpecNamespace class used in this NamespaceCatalog

**add\_namespace**(*name, namespace*)

Add a namespace to this catalog

**Parameters**

- **name** (*str*) – the name of this namespace
- **namespace** (*SpecNamespace*) – the SpecNamespace object

**get\_namespace**(*name*)

Get the a SpecNamespace

**Parameters**

**name** (*str*) – the name of this namespace

**Returns**

the SpecNamespace with the given name

**Return type**

*SpecNamespace*

**get\_spec**(*namespace, data\_type*)

Get the Spec object for the given type from the given Namespace

**Parameters**

- **namespace** (*str*) – the name of the namespace
- **data\_type** (*str* or *type*) – the data\_type to get the spec for

**Returns**

the specification for writing the given object type to HDF5

**Return type**

*Spec*

**get\_hierarchy**(*namespace, data\_type*)

Get the type hierarchy for a given data\_type in a given namespace

**Parameters**

- **namespace** (*str*) – the name of the namespace
- **data\_type** (*str* or *type*) – the data\_type to get the spec for

**Returns**

a tuple with the type hierarchy

**Return type**

*tuple*

**is\_sub\_data\_type**(*namespace*, *data\_type*, *parent\_data\_type*)

Return whether or not *data\_type* is a sub *data\_type* of *parent\_data\_type*

**Parameters**

- **namespace** (*str*) – the name of the namespace containing the data\_type
- **data\_type** (*str*) – the data\_type to check
- **parent\_data\_type** (*str*) – the potential parent data\_type

**Returns**

True if *data\_type* is a sub *data\_type* of *parent\_data\_type*, False otherwise

**Return type**

*bool*

**get\_sources**()

Get all the source specification files that were loaded in this catalog

**get\_namespace\_sources**(*namespace*)

Get all the source specifications that were loaded for a given namespace

**Parameters**

**namespace** (*str*) – the name of the namespace

**get\_types**(*source*)

Get the types that were loaded from a given source

**Parameters**

**source** (*str*) – the name of the source

**load\_namespaces**(*namespace\_path*, *resolve=True*, *reader=None*)

Load the namespaces in the given file

**Parameters**

- **namespace\_path** (*str*) – the path to the file containing the namespaces(s) to load
- **resolve** (*bool*) – whether or not to include objects from included/parent spec objects
- **reader** (*SpecReader*) – the class to user for reading specifications

**Returns**

a dictionary describing the dependencies of loaded namespaces

**Return type**

*dict*

## hdmf.spec.spec module

**class** `hdmf.spec.spec.DtypeHelper`

Bases: `object`

```
primary_dtype_synonyms = {'ascii': ['ascii', 'bytes'], 'bool': ['bool'], 'double':  
['double', 'float64'], 'float': ['float', 'float32'], 'int': ['int32', 'int'],  
'int8': ['int8'], 'isodatettime': ['isodatettime', 'datetime', 'date'], 'long':  
['int64', 'long'], 'numeric': ['numeric'], 'object': ['object'], 'region':  
['region'], 'short': ['int16', 'short'], 'uint16': ['uint16'], 'uint32':  
['uint32', 'uint'], 'uint64': ['uint64'], 'uint8': ['uint8'], 'utf': ['text',  
'utf', 'utf8', 'utf-8']}
```

```
recommended_primary_dtypes = ['float', 'double', 'short', 'int', 'long', 'utf',  
'ascii', 'bool', 'int8', 'uint8', 'uint16', 'uint32', 'uint64', 'object', 'region',  
'numeric', 'isodatettime']
```

```
valid_primary_dtypes = {'ascii', 'bool', 'bytes', 'date', 'datetime', 'double',  
'float', 'float32', 'float64', 'int', 'int16', 'int32', 'int64', 'int8',  
'isodatettime', 'long', 'numeric', 'object', 'region', 'short', 'text', 'uint',  
'uint16', 'uint32', 'uint64', 'uint8', 'utf', 'utf-8', 'utf8'}
```

**static** `simplify_cpd_type(cpd_type)`

Transform a list of DtypeSpecs into a list of strings. Use for simple representation of compound type and validation.

### Parameters

**cpd\_type** (*list*) – The list of DtypeSpecs to simplify

**static** `check_dtype(dtype)`

Check that the dtype string is a reference or a valid primary dtype.

**class** `hdmf.spec.spec.ConstructableDict`

Bases: `dict`

**classmethod** `build_const_args(spec_dict)`

Build constructor arguments for this ConstructableDict class from a dictionary

**classmethod** `build_spec(spec_dict)`

Build a Spec object from the given Spec dict

**class** `hdmf.spec.spec.Spec(doc, name=None, required=True, parent=None)`

Bases: `ConstructableDict`

A base specification class

### Parameters

- **doc** (*str*) – a description about what this specification represents
- **name** (*str*) – The name of this attribute
- **required** (*bool*) – whether or not this attribute is required
- **parent** (*Spec*) – the parent of this spec

**property** `doc`

Documentation on what this Spec is specifying



**property name**

The name of the object being specified

**property parent**

The parent specification of this specification

**classmethod build\_const\_args(*spec\_dict*)**

Build constructor arguments for this Spec class from a dictionary

**property path**

**class** `hdmf.spec.spec.RefSpec(target_type, reftype)`

Bases: [\*ConstructableDict\*](#)

**Parameters**

- **target\_type** ([\*str\*](#)) – the target type *GroupSpec* or *DatasetSpec*
- **reftype** ([\*str\*](#)) – the type of references this is i.e. *region* or *object*

**property target\_type**

The data\_type of the target of the reference

**property reftype**

The type of reference

**is\_region()****Returns**

True if this *RefSpec* specifies a region reference, False otherwise

**Return type**

[\*bool\*](#)

**class** `hdmf.spec.spec.AttributeSpec(name, doc, dtype, shape=None, dims=None, required=True, parent=None, value=None, default_value=None)`

Bases: [\*Spec\*](#)

Specification for attributes

**Parameters**

- **name** ([\*str\*](#)) – The name of this attribute
- **doc** ([\*str\*](#)) – a description about what this specification represents
- **dtype** ([\*str\*](#) or [\*RefSpec\*](#)) – The data type of this attribute
- **shape** ([\*list\*](#) or [\*tuple\*](#)) – the shape of this dataset
- **dims** ([\*list\*](#) or [\*tuple\*](#)) – the dimensions of this dataset
- **required** ([\*bool\*](#)) – whether or not this attribute is required. ignored when “value” is specified
- **parent** ([\*BaseStorageSpec\*](#)) – the parent of this spec
- **value** (*None*) – a constant value for this attribute
- **default\_value** (*None*) – a default value for this attribute

**property dtype**

The data type of the attribute

**property value**

The constant value of the attribute. “None” if this attribute is not constant

**property default\_value**

The default value of the attribute. “None” if this attribute has no default value

**property required**

True if this attribute is required, False otherwise.

**property dims**

The dimensions of this attribute’s value

**property shape**

The shape of this attribute’s value

**classmethod build\_const\_args(*spec\_dict*)**

Build constructor arguments for this Spec class from a dictionary

```
class hdmf.spec.spec.BaseStorageSpec(doc, name=None, default_name=None, attributes=[],  
                                     linkable=True, quantity=1, data_type_def=None,  
                                     data_type_inc=None)
```

Bases: [Spec](#)

A specification for any object that can hold attributes.

**Parameters**

- **doc** ([str](#)) – a description about what this specification represents
- **name** ([str](#)) – the name of this base storage container, allowed only if quantity is not ‘+’ or ‘\*’
- **default\_name** ([str](#)) – The default name of this base storage container, used only if name is None
- **attributes** ([list](#)) – the attributes on this group
- **linkable** ([bool](#)) – whether or not this group can be linked
- **quantity** ([str](#) or [int](#)) – the required number of allowed instance
- **data\_type\_def** ([str](#)) – the data type this specification represents
- **data\_type\_inc** ([str](#) or [BaseStorageSpec](#)) – the data type this specification extends

**property default\_name**

The default name for this spec

**property resolved****property required**

Whether or not the this spec represents a required field

**resolve\_spec(*inc\_spec*)**

Add attributes from the inc\_spec to this spec and track which attributes are new and overridden.

**Parameters**

- **inc\_spec** ([BaseStorageSpec](#)) – the data type this specification represents

**is\_inherited\_spec(*spec*)**

Return True if this spec was inherited from the parent type, False otherwise.

Returns False if the spec is not found.

**Parameters**

**spec** (*Spec* or *str*) – the specification to check

**is\_overridden\_spec(*spec*)**

Return True if this spec overrides a specification from the parent type, False otherwise.

Returns False if the spec is not found.

**Parameters**

**spec** (*Spec* or *str*) – the specification to check

**is\_inherited\_attribute(*name*)**

Return True if the attribute was inherited from the parent type, False otherwise.

Raises a ValueError if the spec is not found.

**Parameters**

**name** (*str*) – the name of the attribute to check

**is\_overridden\_attribute(*name*)**

Return True if the given attribute overrides the specification from the parent, False otherwise.

Raises a ValueError if the spec is not found.

**Parameters**

**name** (*str*) – the name of the attribute to check

**is\_many()**

**classmethod** **get\_data\_type\_spec**(*data\_type\_def*)

**classmethod** **get\_namespace\_spec**()

**property** **attributes**

Tuple of attribute specifications for this specification

**property** **linkable**

True if object can be a link, False otherwise

**classmethod** **id\_key**()

Get the key used to store data ID on an instance

Override this method to use a different name for 'object\_id'

**classmethod** **type\_key**()

Get the key used to store data type on an instance

Override this method to use a different name for 'data\_type'. HDMF supports combining schema that uses 'data\_type' and at most one different name for 'data\_type'.

**classmethod `inc_key()`**

Get the key used to define a `data_type` include.

Override this method to use a different keyword for `'data_type_inc'`. HDMF supports combining schema that uses `'data_type_inc'` and at most one different name for `'data_type_inc'`.

**classmethod `def_key()`**

Get the key used to define a `data_type` definition.

Override this method to use a different keyword for `'data_type_def'` HDMF supports combining schema that uses `'data_type_def'` and at most one different name for `'data_type_def'`.

**property `data_type_inc`**

The data type this specification inherits

**property `data_type_def`**

The data type this specification defines

**property `data_type`**

The data type of this specification

**property `quantity`**

The number of times the object being specified should be present

**`add_attribute(name, doc, dtype, shape=None, dims=None, required=True, parent=None, value=None, default_value=None)`**

Add an attribute to this specification

**Parameters**

- **name** (`str`) – The name of this attribute
- **doc** (`str`) – a description about what this specification represents
- **dtype** (`str` or `RefSpec`) – The data type of this attribute
- **shape** (`list` or `tuple`) – the shape of this dataset
- **dims** (`list` or `tuple`) – the dimensions of this dataset
- **required** (`bool`) – whether or not this attribute is required. ignored when “value” is specified
- **parent** (`BaseStorageSpec`) – the parent of this spec
- **value** (`None`) – a constant value for this attribute
- **default\_value** (`None`) – a default value for this attribute

**`set_attribute(spec)`**

Set an attribute on this specification

**Parameters**

**spec** (`AttributeSpec`) – the specification for the attribute to add

**`get_attribute(name)`**

Get an attribute on this specification

**Parameters**

**name** (`str`) – the name of the attribute to the Spec for

**classmethod `build_const_args(spec_dict)`**

Build constructor arguments for this Spec class from a dictionary

**class** `hdmf.spec.spec.DtypeSpec(name, doc, dtype)`

Bases: [`ConstructableDict`](#)

A class for specifying a component of a compound type

#### Parameters

- **name** (`str`) – the name of this column
- **doc** (`str`) – a description about what this data type is
- **dtype** (`str` or `list` or [`RefSpec`](#)) – the data type of this column

#### property doc

Documentation about this component

#### property name

The name of this component

#### property dtype

The data type of this component

**static** `assertValidDtype(dtype)`

**static** `check_valid_dtype(dtype)`

**static** `is_ref(spec)`

#### Parameters

**spec** (`str` or `dict`) – the spec object to check

**classmethod** `build_const_args(spec_dict)`

Build constructor arguments for this Spec class from a dictionary

**class** `hdmf.spec.spec.DatasetSpec(doc, dtype=None, name=None, default_name=None, shape=None, dims=None, attributes=[], linkable=True, quantity=1, default_value=None, data_type_def=None, data_type_inc=None)`

Bases: [`BaseStorageSpec`](#)

Specification for datasets

To specify a table-like dataset i.e. a compound data type.

#### Parameters

- **doc** (`str`) – a description about what this specification represents
- **dtype** (`str` or `list` or [`RefSpec`](#)) – The data type of this attribute. Use a list of `DtypeSpecs` to specify a compound data type.
- **name** (`str`) – The name of this dataset
- **default\_name** (`str`) – The default name of this dataset
- **shape** (`list` or `tuple`) – the shape of this dataset
- **dims** (`list` or `tuple`) – the dimensions of this dataset
- **attributes** (`list`) – the attributes on this group
- **linkable** (`bool`) – whether or not this group can be linked
- **quantity** (`str` or `int`) – the required number of allowed instance
- **default\_value** (`None`) – a default value for this dataset

- **data\_type\_def** ([str](#)) – the data type this specification represents
- **data\_type\_inc** ([str](#) or [DatasetSpec](#)) – the data type this specification extends

**resolve\_spec**(*inc\_spec*)

**Parameters**

**inc\_spec** ([DatasetSpec](#)) – the data type this specification represents

**property dims**

The dimensions of this Dataset

**property dtype**

The data type of the Dataset

**property shape**

The shape of the dataset

**property default\_value**

The default value of the dataset or None if not specified

**classmethod dtype\_spec\_cls()**

The class to use when constructing DtypeSpec objects

Override this if extending to use a class other than DtypeSpec to build dataset specifications

**classmethod build\_const\_args**(*spec\_dict*)

Build constructor arguments for this Spec class from a dictionary

**class** `hdmf.spec.spec.LinkSpec`(*doc*, *target\_type*, *quantity=1*, *name=None*)

Bases: [Spec](#)

**Parameters**

- **doc** ([str](#)) – a description about what this link represents
- **target\_type** ([str](#) or [BaseStorageSpec](#)) – the target type GroupSpec or DatasetSpec
- **quantity** ([str](#) or [int](#)) – the required number of allowed instance
- **name** ([str](#)) – the name of this link

**property target\_type**

The data type of target specification

**property data\_type\_inc**

The data type of target specification

**is\_many()**

**property quantity**

The number of times the object being specified should be present

**property required**

Whether or not the this spec represents a required field

**class** `hdmf.spec.spec.GroupSpec`(*doc*, *name=None*, *default\_name=None*, *groups=[]*, *datasets=[]*,  
*attributes=[]*, *links=[]*, *linkable=True*, *quantity=1*, *data\_type\_def=None*,  
*data\_type\_inc=None*)

Bases: [BaseStorageSpec](#)

Specification for groups

**Parameters**

- **doc** (*str*) – a description about what this specification represents
- **name** (*str*) – the name of the Group that is written to the file. If this argument is omitted, users will be required to enter a **name** field when creating instances of this data type in the API. Another option is to specify **default\_name**, in which case this name will be used as the name of the Group if no other name is provided.
- **default\_name** (*str*) – The default name of this group
- **groups** (*list*) – the subgroups in this group
- **datasets** (*list*) – the datasets in this group
- **attributes** (*list*) – the attributes on this group
- **links** (*list*) – the links in this group
- **linkable** (*bool*) – whether or not this group can be linked
- **quantity** (*str* or *int*) – the allowable number of instance of this group in a certain location. See table of options [here](#). Note that if you specify **name**, **quantity** cannot be '\*', '+', or an integer greater than 1, because you cannot have more than one group of the same name in the same parent group.
- **data\_type\_def** (*str*) – the data type this specification represents
- **data\_type\_inc** (*str* or *GroupSpec*) – the data type this specification data\_type\_inc

**resolve\_spec**(*inc\_spec*)**Parameters****inc\_spec** (*GroupSpec*) – the data type this specification represents**is\_inherited\_dataset**(*name*)

Return true if a dataset with the given name was inherited

**Parameters****name** (*str*) – the name of the dataset**is\_overridden\_dataset**(*name*)

Return true if a dataset with the given name overrides a specification from the parent type

**Parameters****name** (*str*) – the name of the dataset**is\_inherited\_group**(*name*)

Return true if a group with the given name was inherited

**Parameters****name** (*str*) – the name of the group**is\_overridden\_group**(*name*)

Return true if a group with the given name overrides a specification from the parent type

**Parameters****name** (*str*) – the name of the group**is\_inherited\_link**(*name*)

Return true if a link with the given name was inherited

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

**is\_overridden\_link**(*name*)

Return true if a link with the given name overrides a specification from the parent type

**Parameters**

**name** (*str*) – the name of the link

**is\_inherited\_spec**(*spec*)

Returns ‘True’ if specification was inherited from a parent type

**Parameters**

**spec** (*Spec* or *str*) – the specification to check

**is\_overridden\_spec**(*spec*)

Returns ‘True’ if specification overrides a specification from the parent type

**Parameters**

**spec** (*Spec* or *str*) – the specification to check

**is\_inherited\_type**(*spec*)

Returns True if *spec* represents a data type that was inherited

**Parameters**

**spec** (*BaseStorageSpec* or *str*) – the specification to check

**is\_overridden\_type**(*spec*)

Returns True if *spec* represents a data type that overrides a specification from a parent type

**Parameters**

**spec** (*BaseStorageSpec* or *str*) – the specification to check

**is\_inherited\_target\_type**(*spec*)

Returns True if *spec* represents a target type that was inherited

**Parameters**

**spec** (*LinkSpec* or *str*) – the specification to check

**is\_overridden\_target\_type**(*spec*)

Returns True if *spec* represents a target type that overrides a specification from a parent type

**Parameters**

**spec** (*LinkSpec* or *str*) – the specification to check

**get\_data\_type**(*data\_type*)

Get a specification by “data\_type”

NOTE: If there is only one spec for a given data type, then it is returned. If there are multiple specs for a given data type and they are all named, then they are returned in a list. If there are multiple specs for a given data type and only one is unnamed, then the unnamed spec is returned. The other named specs can be returned using `get_group` or `get_dataset`.

NOTE: this method looks for an exact match of the data type and does not consider the type hierarchy.

**Parameters**

**data\_type** (*str*) – the data\_type to retrieve

**get\_target\_type**(*target\_type*)

Get a specification by “target\_type”



NOTE: If there is only one spec for a given target type, then it is returned. If there are multiple specs for a given target type and they are all named, then they are returned in a list. If there are multiple specs for a given target type and only one is unnamed, then the unnamed spec is returned. The other named specs can be returned using `get_link`.

NOTE: this method looks for an exact match of the target type and does not consider the type hierarchy.

#### Parameters

**target\_type** (`str`) – the target\_type to retrieve

#### property groups

The groups specified in this GroupSpec

#### property datasets

The datasets specified in this GroupSpec

#### property links

The links specified in this GroupSpec

**add\_group**(*doc*, *name*=None, *default\_name*=None, *groups*=[], *datasets*=[], *attributes*=[], *links*=[], *linkable*=True, *quantity*=1, *data\_type\_def*=None, *data\_type\_inc*=None)

Add a new specification for a subgroup to this group specification

#### Parameters

- **doc** (`str`) – a description about what this specification represents
- **name** (`str`) – the name of the Group that is written to the file. If this argument is omitted, users will be required to enter a `name` field when creating instances of this data type in the API. Another option is to specify `default_name`, in which case this name will be used as the name of the Group if no other name is provided.
- **default\_name** (`str`) – The default name of this group
- **groups** (`list`) – the subgroups in this group
- **datasets** (`list`) – the datasets in this group
- **attributes** (`list`) – the attributes on this group
- **links** (`list`) – the links in this group
- **linkable** (`bool`) – whether or not this group can be linked
- **quantity** (`str` or `int`) – the allowable number of instance of this group in a certain location. See table of options [here](#). Note that if you specify `name`, `quantity` cannot be '\*', '+', or an integer greater than 1, because you cannot have more than one group of the same name in the same parent group.
- **data\_type\_def** (`str`) – the data type this specification represents
- **data\_type\_inc** (`str` or `GroupSpec`) – the data type this specification data\_type\_inc

#### set\_group(*spec*)

Add the given specification for a subgroup to this group specification

#### Parameters

**spec** (`GroupSpec`) – the specification for the subgroup

**get\_group**(*name*)

Get a specification for a subgroup to this group specification

**Parameters**

**name** (*str*) – the name of the group to the Spec for

**add\_dataset**(*doc*, *dtype=None*, *name=None*, *default\_name=None*, *shape=None*, *dims=None*, *attributes=[]*, *linkable=True*, *quantity=1*, *default\_value=None*, *data\_type\_def=None*, *data\_type\_inc=None*)

Add a new specification for a dataset to this group specification

**Parameters**

- **doc** (*str*) – a description about what this specification represents
- **dtype** (*str* or *list* or *RefSpec*) – The data type of this attribute. Use a list of *DtypeSpecs* to specify a compound data type.
- **name** (*str*) – The name of this dataset
- **default\_name** (*str*) – The default name of this dataset
- **shape** (*list* or *tuple*) – the shape of this dataset
- **dims** (*list* or *tuple*) – the dimensions of this dataset
- **attributes** (*list*) – the attributes on this group
- **linkable** (*bool*) – whether or not this group can be linked
- **quantity** (*str* or *int*) – the required number of allowed instance
- **default\_value** (*None*) – a default value for this dataset
- **data\_type\_def** (*str*) – the data type this specification represents
- **data\_type\_inc** (*str* or *DatasetSpec*) – the data type this specification extends

**set\_dataset**(*spec*)

Add the given specification for a dataset to this group specification

**Parameters**

**spec** (*DatasetSpec*) – the specification for the dataset

**get\_dataset**(*name*)

Get a specification for a dataset to this group specification

**Parameters**

**name** (*str*) – the name of the dataset to the Spec for

**add\_link**(*doc*, *target\_type*, *quantity=1*, *name=None*)

Add a new specification for a link to this group specification

**Parameters**

- **doc** (*str*) – a description about what this link represents
- **target\_type** (*str* or *BaseStorageSpec*) – the target type *GroupSpec* or *DatasetSpec*
- **quantity** (*str* or *int*) – the required number of allowed instance
- **name** (*str*) – the name of this link

**set\_link**(*spec*)

Add a given specification for a link to this group specification

**Parameters**

**spec** (*LinkSpec*) – the specification for the object to link to

**get\_link**(*name*)

Get a specification for a link to this group specification

**Parameters**

**name** (*str*) – the name of the link to the Spec for

**classmethod dataset\_spec\_cls()**

The class to use when constructing DatasetSpec objects

Override this if extending to use a class other than DatasetSpec to build dataset specifications

**classmethod link\_spec\_cls()**

The class to use when constructing LinkSpec objects

Override this if extending to use a class other than LinkSpec to build link specifications

**classmethod build\_const\_args**(*spec\_dict*)

Build constructor arguments for this Spec class from a dictionary

**hdmf.spec.write module**

**class** hdmf.spec.write.SpecWriter

Bases: *object*

**abstract write\_spec**(*spec\_file\_dict*, *path*)

**abstract write\_namespace**(*namespace*, *path*)

**class** hdmf.spec.write.YAMLSpecWriter(*outdir='.'*)

Bases: *SpecWriter*

**Parameters**

**outdir** (*str*) – the path to write the directory to output the namespace and specs too

**write\_spec**(*spec\_file\_dict*, *path*)

**write\_namespace**(*namespace*, *path*)

Write the given namespace key-value pairs as YAML to the given path.

**Parameters**

- **namespace** – SpecNamespace holding the key-value pairs that define the namespace
- **path** – File path to write the namespace to as YAML under the key ‘namespaces’

**reorder\_yaml**(*path*)

Open a YAML file, load it as python data, sort the data alphabetically, and write it back out to the same path.

**sort\_keys**(*obj*)

**class** hdmf.spec.write.NamespaceBuilder(*doc*, *name*, *full\_name=None*, *version=None*, *author=None*, *contact=None*, *date=None*, *namespace\_cls=<class 'hdmf.spec.namespace.SpecNamespace'>*)

Bases: *object*

A class for building namespace and spec files

**Parameters**

- **doc** (*str*) – Description about what the namespace represents
- **name** (*str*) – Name of the namespace
- **full\_name** (*str*) – Extended full name of the namespace
- **version** (*str* or *tuple* or *list*) – Version number of the namespace
- **author** (*str* or *list*) – Author or list of authors.
- **contact** (*str* or *list*) – List of emails. Ordering should be the same as for author
- **date** (*datetime* or *str*) – Date last modified or released. Formatting is %Y-%m-%d %H:%M:%S, e.g, 2017-04-25 17:14:13
- **namespace\_cls** (*type*) – the SpecNamespace type

**add\_spec**(*source, spec*)

Add a Spec to the namespace

**Parameters**

- **source** (*str*) – the path to write the spec to
- **spec** (*GroupSpec* or *DatasetSpec*) – the Spec to add

**add\_source**(*source, doc=None, title=None*)

Add a source file to the namespace

**Parameters**

- **source** (*str*) – the path to write the spec to
- **doc** (*str*) – additional documentation for the source file
- **title** (*str*) – optional heading to be used for the source

**include\_type**(*data\_type, source=None, namespace=None*)

Include a data type from an existing namespace or source

**Parameters**

- **data\_type** (*str*) – the data type to include
- **source** (*str*) – the source file to include the type from
- **namespace** (*str*) – the namespace from which to include the data type

**include\_namespace**(*namespace*)

Include an entire namespace

**Parameters**

- **namespace** (*str*) – the namespace to include

**export**(*path, outdir='.', writer=None*)

Export the namespace to the given path.

All new specification source files will be written in the same directory as the given path.

**Parameters**

- **path** (*str*) – the path to write the spec to
- **outdir** (*str*) – the path to write the directory to output the namespace and specs too

- **writer** (*SpecWriter*) – the SpecWriter to use to write the namespace

**property name**

**class** `hdmf.spec.write.SpecFileBuilder`

Bases: `dict`

**add\_spec**(*spec*)

**Parameters**

**spec** (*GroupSpec* or *DatasetSpec*) – the Spec to add

`hdmf.spec.write.export_spec(ns_builder, new_data_types, output_dir)`

Create YAML specification files for a new namespace and extensions with the given data type specs.

**Parameters**

- **ns\_builder** – NamespaceBuilder instance used to build the namespace and extension
- **new\_data\_types** – Iterable of specs that represent new data types to be added

## 6.4.2 Module contents

# 6.5 hdmf.backends package

## 6.5.1 Subpackages

**hdmf.backends.hdf5 package**

**Submodules**

**hdmf.backends.hdf5.h5\_utils module**

Utilities for the HDF5 I/O backend, e.g., for wrapping HDF5 datasets on read, wrapping arrays for configuring write, or writing the spec among others

**class** `hdmf.backends.hdf5.h5_utils.HDF5IODataChunkIteratorQueue`

Bases: `deque`

Helper class used by HDF5IO to manage the write for DataChunkIterators

Each queue element must be a tuple of two elements: 1) the dataset to write to and 2) the AbstractDataChunkIterator with the data

**exhaust\_queue**()

Read and write from any queued DataChunkIterators in a round-robin fashion

**append**(*dataset, data*)

Append a value to the queue

**Parameters**

- **dataset** (*Dataset*) – The dataset where the DataChunkIterator is written to
- **data** (*AbstractDataChunkIterator*) – DataChunkIterator with the data to be written

```
class hdmf.backends.hdf5.h5_utils.H5Dataset(dataset, io)
```

Bases: [\*HDMFDataset\*](#)

**Parameters**

- **dataset** ([\*Dataset\*](#) or [\*Array\*](#)) – the HDF5 file lazily evaluate
- **io** ([\*HDF5IO\*](#)) – the IO object that was used to read the underlying dataset

**property io**

**property regionref**

**property ref**

**property shape**

```
class hdmf.backends.hdf5.h5_utils.DatasetOfReferences(dataset, io)
```

Bases: [\*H5Dataset\*](#), [\*ReferenceResolver\*](#)

An extension of the base ReferenceResolver class to add more abstract methods for subclasses that will read HDF5 references

**Parameters**

- **dataset** ([\*Dataset\*](#) or [\*Array\*](#)) – the HDF5 file lazily evaluate
- **io** ([\*HDF5IO\*](#)) – the IO object that was used to read the underlying dataset

**abstract get\_object(h5obj)**

A class that maps an HDF5 object to a Builder or Container

**invert()**

Return an object that defers reference resolution but in the opposite direction.

```
class hdmf.backends.hdf5.h5_utils.BuilderResolverMixin
```

Bases: [\*BuilderResolver\*](#)

A mixin for adding to HDF5 reference-resolving types the get\_object method that returns Builders

**get\_object(h5obj)**

A class that maps an HDF5 object to a Builder

```
class hdmf.backends.hdf5.h5_utils.ContainerResolverMixin
```

Bases: [\*ContainerResolver\*](#)

A mixin for adding to HDF5 reference-resolving types the get\_object method that returns Containers

**get\_object(h5obj)**

A class that maps an HDF5 object to a Container

```
class hdmf.backends.hdf5.h5_utils.AbstractH5TableDataset(dataset, io, types)
```

Bases: [\*DatasetOfReferences\*](#)

**Parameters**

- **dataset** ([\*Dataset\*](#) or [\*Array\*](#)) – the HDF5 file lazily evaluate
- **io** ([\*HDF5IO\*](#)) – the IO object that was used to read the underlying dataset
- **types** ([\*list\*](#) or [\*tuple\*](#)) – the IO object that was used to read the underlying dataset

**property types**

property dtype

`__getitem__(arg)`

`resolve(manager)`

**class** `hdmf.backends.hdf5.h5_utils.AbstractH5ReferenceDataset(dataset, io)`

Bases: [`DatasetOfReferences`](#)

**Parameters**

- **dataset** ([`Dataset`](#) or [`Array`](#)) – the HDF5 file lazily evaluate
- **io** ([`HDF5IO`](#)) – the IO object that was used to read the underlying dataset

`__getitem__(arg)`

property dtype

**class** `hdmf.backends.hdf5.h5_utils.AbstractH5RegionDataset(dataset, io)`

Bases: [`AbstractH5ReferenceDataset`](#)

**Parameters**

- **dataset** ([`Dataset`](#) or [`Array`](#)) – the HDF5 file lazily evaluate
- **io** ([`HDF5IO`](#)) – the IO object that was used to read the underlying dataset

`__getitem__(arg)`

property dtype

**class** `hdmf.backends.hdf5.h5_utils.ContainerH5TableDataset(dataset, io, types)`

Bases: [`ContainerResolverMixin`](#), [`AbstractH5TableDataset`](#)

A reference-resolving dataset for resolving references inside tables (i.e. compound dtypes) that returns resolved references as Containers

**Parameters**

- **dataset** ([`Dataset`](#) or [`Array`](#)) – the HDF5 file lazily evaluate
- **io** ([`HDF5IO`](#)) – the IO object that was used to read the underlying dataset
- **types** ([`list`](#) or [`tuple`](#)) – the IO object that was used to read the underlying dataset

**classmethod** `get_inverse_class()`

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

`BuilderResolver.get_inverse_class` should return a class that subclasses `ContainerResolver`.

`ContainerResolver.get_inverse_class` should return a class that subclasses `BuilderResolver`.

**class** `hdmf.backends.hdf5.h5_utils.BuilderH5TableDataset(dataset, io, types)`

Bases: [`BuilderResolverMixin`](#), [`AbstractH5TableDataset`](#)

A reference-resolving dataset for resolving references inside tables (i.e. compound dtypes) that returns resolved references as Builders

**Parameters**

- **dataset** ([`Dataset`](#) or [`Array`](#)) – the HDF5 file lazily evaluate
- **io** ([`HDF5IO`](#)) – the IO object that was used to read the underlying dataset
- **types** ([`list`](#) or [`tuple`](#)) – the IO object that was used to read the underlying dataset

**classmethod** `get_inverse_class()`

Return the class that represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get\_inverse\_class should return a class that subclasses ContainerResolver.

ContainerResolver.get\_inverse\_class should return a class that subclasses BuilderResolver.

**class** `hdmf.backends.hdf5.h5_utils.ContainerH5ReferenceDataset(dataset, io)`

Bases: *ContainerResolverMixin*, *AbstractH5ReferenceDataset*

A reference-resolving dataset for resolving object references that returns resolved references as Containers

**Parameters**

- **dataset** (*Dataset* or *Array*) – the HDF5 file lazily evaluate
- **io** (*HDF5IO*) – the IO object that was used to read the underlying dataset

**classmethod** `get_inverse_class()`

Return the class that represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get\_inverse\_class should return a class that subclasses ContainerResolver.

ContainerResolver.get\_inverse\_class should return a class that subclasses BuilderResolver.

**class** `hdmf.backends.hdf5.h5_utils.BuilderH5ReferenceDataset(dataset, io)`

Bases: *BuilderResolverMixin*, *AbstractH5ReferenceDataset*

A reference-resolving dataset for resolving object references that returns resolved references as Builders

**Parameters**

- **dataset** (*Dataset* or *Array*) – the HDF5 file lazily evaluate
- **io** (*HDF5IO*) – the IO object that was used to read the underlying dataset

**classmethod** `get_inverse_class()`

Return the class that represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get\_inverse\_class should return a class that subclasses ContainerResolver.

ContainerResolver.get\_inverse\_class should return a class that subclasses BuilderResolver.

**class** `hdmf.backends.hdf5.h5_utils.ContainerH5RegionDataset(dataset, io)`

Bases: *ContainerResolverMixin*, *AbstractH5RegionDataset*

A reference-resolving dataset for resolving region references that returns resolved references as Containers

**Parameters**

- **dataset** (*Dataset* or *Array*) – the HDF5 file lazily evaluate
- **io** (*HDF5IO*) – the IO object that was used to read the underlying dataset

**classmethod** `get_inverse_class()`

Return the class that represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get\_inverse\_class should return a class that subclasses ContainerResolver.

ContainerResolver.get\_inverse\_class should return a class that subclasses BuilderResolver.

**class** `hdmf.backends.hdf5.h5_utils.BuilderH5RegionDataset(dataset, io)`

Bases: *BuilderResolverMixin*, *AbstractH5RegionDataset*

A reference-resolving dataset for resolving region references that returns resolved references as Builders

**Parameters**



- **dataset** ([Dataset](#) or [Array](#)) – the HDF5 file lazily evaluate
- **io** ([HDF5IO](#)) – the IO object that was used to read the underlying dataset

**classmethod** `get_inverse_class()`

Return the class that represents the ReferenceResolver that resolves references to the opposite type.

`BuilderResolver.get_inverse_class` should return a class that subclasses `ContainerResolver`.

`ContainerResolver.get_inverse_class` should return a class that subclasses `BuilderResolver`.

**class** `hdmf.backends.hdf5.h5_utils.H5SpecWriter(group)`

Bases: [SpecWriter](#)

**Parameters**

**group** ([Group](#)) – the HDF5 file to write specs to

**static** `stringify(spec)`

Converts a spec into a JSON string to write to a dataset

**write\_spec(spec, path)**

**write\_namespace(namespace, path)**

**class** `hdmf.backends.hdf5.h5_utils.H5SpecReader(group)`

Bases: [SpecReader](#)

Class that reads cached JSON-formatted namespace and spec data from an HDF5 group.

**Parameters**

**group** ([Group](#)) – the HDF5 group to read specs from

**read\_spec(spec\_path)**

**read\_namespace(ns\_path)**

**class** `hdmf.backends.hdf5.h5_utils.H5RegionSlicer(dataset, region)`

Bases: [RegionSlicer](#)

**Parameters**

- **dataset** ([Dataset](#) or [H5Dataset](#)) – the HDF5 dataset to slice
- **region** ([RegionReference](#)) – the region reference to use to slice

**\_\_getitem\_\_(idx)**

Must be implemented by subclasses

**class** `hdmf.backends.hdf5.h5_utils.H5DataIO(data=None, maxshape=None, chunks=None, compression=None, compression_opts=None, fillvalue=None, shuffle=None, fletcher32=None, link_data=False, allow_plugin_filters=False, shape=None, dtype=None)`

Bases: [DataIO](#)

Wrap data arrays for write via HDF5IO to customize I/O behavior, such as compression and chunking for data arrays.

**Parameters**

- **data** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Iterable](#)) – the data to be written. NOTE: If an `h5py.Dataset` is used, all other settings but `link_data` will be ignored as the dataset will either be linked to or copied as is in `H5DataIO`.

- **maxshape** (`tuple`) – Dataset will be resizable up to this shape (Tuple). Automatically enables chunking. Use None for the axes you want to be unlimited.
- **chunks** (`bool` or `tuple`) – Chunk shape or True to enable auto-chunking
- **compression** (`str` or `bool` or `int`) – Compression strategy. If a bool is given, then gzip compression will be used by default. <http://docs.h5py.org/en/latest/high/dataset.html#dataset-compression>
- **compression\_opts** (`int` or `tuple`) – Parameter for compression filter
- **fillvalue** (`None`) – Value to be returned when reading uninitialized parts of the dataset
- **shuffle** (`bool`) – Enable shuffle I/O filter. <http://docs.h5py.org/en/latest/high/dataset.html#dataset-shuffle>
- **fletcher32** (`bool`) – Enable fletcher32 checksum. <http://docs.h5py.org/en/latest/high/dataset.html#dataset-fletcher32>
- **link\_data** (`bool`) – If data is an `h5py.Dataset` should it be linked to or copied. NOTE: This parameter is only allowed if data is an `h5py.Dataset`
- **allow\_plugin\_filters** (`bool`) – Enable passing dynamically loaded filters as compression parameter
- **shape** (`tuple`) – the shape of the new dataset, used only if data is None
- **dtype** (`str` or `type` or `dtype`) – the data type of the new dataset, used only if data is None

#### property dataset

#### get\_io\_params()

Returns a dict with the I/O parameters specified in this DataIO.

#### static filter\_available(*filter*, *allow\_plugin\_filters*)

Check if a given I/O filter is available

##### Parameters

- **filter** (`str`, `int`) – String with the name of the filter, e.g., gzip, szip etc. int with the registered filter ID, e.g. 307
- **allow\_plugin\_filters** – bool indicating whether the given filter can be dynamically loaded

##### Returns

bool indicating whether the given filter is available

#### property link\_data

#### property io\_settings

#### property valid

bool indicating if the data object is valid

**hdmf.backends.hdf5.h5tools module**

```
class hdmf.backends.hdf5.h5tools.HDF5IO(path=None, mode='r', manager=None, comm=None, file=None,
                                         driver=None, herd_path=None)
```

Bases: [HDMFIO](#)

Open an HDF5 file for IO.

**Parameters**

- **path** ([str](#) or [Path](#)) – the path to the HDF5 file
- **mode** ([str](#)) – the mode to open the HDF5 file with, one of (“w”, “r”, “r+”, “a”, “w-”, “x”). See [h5py.File](#) for more details.
- **manager** ([TypeMap](#) or [BuildManager](#)) – the BuildManager or a TypeMap to construct a BuildManager to use for I/O
- **comm** ([Intracomm](#)) – the MPI communicator to use for parallel I/O
- **file** ([File](#) or [S3File](#) or [RemFile](#)) – a pre-existing h5py.File, S3File, or RemFile object
- **driver** ([str](#)) – driver for h5py to use when opening HDF5 file
- **herd\_path** ([str](#)) – The path to read/write the HERD file

```
static can_read(path)
```

Determines whether a given path is readable by the HDF5IO class

**property** **comm**

The MPI communicator to use for parallel I/O.

**property** **driver**

```
classmethod load_namespaces(namespace_catalog, path=None, namespaces=None, file=None,
                             driver=None)
```

Load cached namespaces from a file.

If *file* is not supplied, then an [h5py.File](#) object will be opened for the given *path*, the namespaces will be read, and the File object will be closed. If *file* is supplied, then the given File object will be read from and not closed.

**raises** **ValueError**

if both *path* and *file* are supplied but *path* is not the same as the path of *file*.

**Parameters**

- **namespace\_catalog** ([NamespaceCatalog](#) or [TypeMap](#)) – the NamespaceCatalog or TypeMap to load namespaces into
- **path** ([str](#) or [Path](#)) – the path to the HDF5 file
- **namespaces** ([list](#)) – the namespaces to load
- **file** ([File](#)) – a pre-existing h5py.File object
- **driver** ([str](#)) – driver for h5py to use when opening HDF5 file

**Returns**

dict mapping the names of the loaded namespaces to a dict mapping included namespace names and the included data types

**Return type**`dict`**classmethod** `get_namespaces(path=None, file=None, driver=None)`

Get the names and versions of the cached namespaces from a file.

If `file` is not supplied, then an `h5py.File` object will be opened for the given `path`, the namespaces will be read, and the File object will be closed. If `file` is supplied, then the given File object will be read from and not closed.

If there are multiple versions of a namespace cached in the file, then only the latest one (using alphanumeric ordering) is returned. This is the version of the namespace that is loaded by `HDF5IO.load_namespaces(...)`.

**raises ValueError**

if both `path` and `file` are supplied but `path` is not the same as the path of `file`.

**Parameters**

- **path** (`str` or `Path`) – the path to the HDF5 file
- **file** (`File`) – a pre-existing `h5py.File` object
- **driver** (`str`) – driver for `h5py` to use when opening HDF5 file

**Returns**

dict mapping names to versions of the namespaces in the file

**Return type**`dict`**classmethod** `copy_file(source_filename, dest_filename, expand_external=True, expand_refs=False, expand_soft=False)`

Convenience function to copy an HDF5 file while allowing external links to be resolved.

**Warning:** As of HDMF 2.0, this method is no longer supported and may be removed in a future version. Please use the `export` method or `h5py.File.copy` method instead.

---

**Note:** The source file will be opened in ‘r’ mode and the destination file will be opened in ‘w’ mode using `h5py`. To avoid possible collisions, care should be taken that, e.g., the source file is not opened already when calling this function.

---

**Parameters**

- **source\_filename** (`str`) – the path to the HDF5 file to copy
- **dest\_filename** (`str`) – the name of the destination file
- **expand\_external** (`bool`) – expand external links into new objects
- **expand\_refs** (`bool`) – copy objects which are pointed to by reference
- **expand\_soft** (`bool`) – expand soft links into new objects

**write**(*container*, *cache\_spec=True*, *link\_data=True*, *exhaust\_dci=True*, *herd=None*)

Write the container to an HDF5 file.

#### Parameters

- **container** (*Container*) – the Container object to write
- **cache\_spec** (*bool*) – If True (default), cache specification to file (highly recommended). If False, do not cache specification to file. The appropriate specification will then need to be loaded prior to reading the file.
- **link\_data** (*bool*) – If True (default), create external links to HDF5 Datasets. If False, copy HDF5 Datasets.
- **exhaust\_dci** (*bool*) – If True (default), exhaust DataChunkIterators one at a time. If False, exhaust them concurrently.
- **herd** (*HERD*) – A HERD object to populate with references.

**export**(*src\_io*, *container=None*, *write\_args=None*, *cache\_spec=True*)

Export data read from a file from any backend to HDF5.

See [hdmf.backends.io.HDMFIO.export](#) for more details.

#### Parameters

- **src\_io** (*HDMFIO*) – the HDMFIO object for reading the data to export
- **container** (*Container*) – the Container object to export. If None, then the entire contents of the HDMFIO object will be exported
- **write\_args** (*dict*) – arguments to pass to [write\\_builder](#)
- **cache\_spec** (*bool*) – whether to cache the specification to file

**classmethod export\_io**(*path*, *src\_io*, *comm=None*, *container=None*, *write\_args=None*, *cache\_spec=True*)

Export from one backend to HDF5 (class method).

Convenience function for [export](#) where you do not need to instantiate a new HDF5IO object for writing. An HDF5IO object is created with mode 'w' and the given arguments.

Example usage:

```
old_io = HDF5IO('old.h5', 'r')
HDF5IO.export_io(path='new_copy.h5', src_io=old_io)
```

See [export](#) for more details.

#### Parameters

- **path** (*str*) – the path to the destination HDF5 file
- **src\_io** (*HDMFIO*) – the HDMFIO object for reading the data to export
- **comm** (*Intracomm*) – the MPI communicator to use for parallel I/O
- **container** (*Container*) – the Container object to export. If None, then the entire contents of the HDMFIO object will be exported
- **write\_args** (*dict*) – arguments to pass to [write\\_builder](#)
- **cache\_spec** (*bool*) – whether to cache the specification to file

**read()**

Read a container from the IO source.

**Returns**

the Container object that was read in

**Return type**

*Container*

**read\_builder()**

Read data and return the GroupBuilder representing it.

NOTE: On read, the Builder.source may will usually not be set of the Builders. NOTE: The Builder.location is used internally to ensure correct handling of links (in particular on export) and should be set on read for all GroupBuilder, DatasetBuilder, and LinkBuilder objects.

**Returns**

a GroupBuilder representing the data object

**Return type**

*GroupBuilder*

**get\_written(builder)**

Return True if this builder has been written to (or read from) disk by this IO object, False otherwise.

**Parameters**

**builder** (*Builder*) – Builder object to get the written flag for

**Returns**

True if the builder is found in self.\_written\_builders using the builder ID, False otherwise

**get\_builder(h5obj)**

Get the builder for the corresponding h5py Group or Dataset

**raises ValueError**

When no builder has been constructed yet for the given h5py object

**Parameters**

**h5obj** (*Dataset* or *Group*) – the HDF5 object to the corresponding Builder object for

**get\_container(h5obj)**

Get the container for the corresponding h5py Group or Dataset

**raises ValueError**

When no builder has been constructed yet for the given h5py object

**Parameters**

**h5obj** (*Dataset* or *Group*) – the HDF5 object to the corresponding Container/Data object for

**open()**

Open this HDMFIO object for writing of the builder

**close(close\_links=True)**

Close this file and any files linked to from this file.

**Parameters**

**close\_links** (*bool*) – Whether to close all files linked to from this file. (default: True)

**close\_linked\_files()**

Close all opened, linked-to files.

MacOS and Linux automatically release the linked-to file after the linking file is closed, but Windows does not, which prevents the linked-to file from being deleted or truncated. Use this method to close all opened, linked-to files.

**write\_builder**(*builder*, *link\_data=True*, *exhaust\_dci=True*, *export\_source=None*)

**Parameters**

- **builder** (*GroupBuilder*) – the GroupBuilder object representing the HDF5 file
- **link\_data** (*bool*) – If not specified otherwise link (True) or copy (False) HDF5 Datasets
- **exhaust\_dci** (*bool*) – exhaust DataChunkIterators one at a time. If False, exhaust them concurrently
- **export\_source** (*str*) – The source of the builders when exporting

**classmethod** **get\_type**(*data*)

**set\_attributes**(*obj*, *attributes*)

**Parameters**

- **obj** (*Group* or *Dataset*) – the HDF5 object to add attributes to
- **attributes** (*dict*) – a dict containing the attributes on the Group or Dataset, indexed by attribute name

**write\_group**(*parent*, *builder*, *link\_data=True*, *exhaust\_dci=True*, *export\_source=None*)

**Parameters**

- **parent** (*Group*) – the parent HDF5 object
- **builder** (*GroupBuilder*) – the GroupBuilder to write
- **link\_data** (*bool*) – If not specified otherwise link (True) or copy (False) HDF5 Datasets
- **exhaust\_dci** (*bool*) – exhaust DataChunkIterators one at a time. If False, exhaust them concurrently
- **export\_source** (*str*) – The source of the builders when exporting

**Returns**

the Group that was created

**Return type**

*Group*

**write\_link**(*parent*, *builder*, *export\_source=None*)

**Parameters**

- **parent** (*Group*) – the parent HDF5 object
- **builder** (*LinkBuilder*) – the LinkBuilder to write
- **export\_source** (*str*) – The source of the builders when exporting

**Returns**

the Link that was created

**Return type**

*SoftLink* or *ExternalLink*

**write\_dataset**(*parent, builder, link\_data=True, exhaust\_dci=True, export\_source=None*)

Write a dataset to HDF5

The function uses other dataset-dependent write functions, e.g, `__scalar_fill__`, `__list_fill__`, and `__setup_chunked_dset__` to write the data.

#### Parameters

- **parent** (*Group*) – the parent HDF5 object
- **builder** (*DatasetBuilder*) – the DatasetBuilder to write
- **link\_data** (*bool*) – If not specified otherwise link (True) or copy (False) HDF5 Datasets
- **exhaust\_dci** (*bool*) – exhaust DataChunkIterators one at a time. If False, exhaust them concurrently
- **export\_source** (*str*) – The source of the builders when exporting

#### Returns

the Dataset that was created

#### Return type

*Dataset*

#### property mode

Return the HDF5 file mode. One of ("w", "r", "r+", "a", "w-", "x").

**classmethod set\_dataio**(*data=None, maxshape=None, chunks=None, compression=None, compression\_opts=None, fillvalue=None, shuffle=None, fletcher32=None, link\_data=False, allow\_plugin\_filters=False, shape=None, dtype=None*)

Wrap the given Data object with an H5DataIO.

This method is provided merely for convenience. It is the equivalent of the following:

```
from hdmf.backends.hdf5 import H5DataIO
data = ...
data = H5DataIO(data)
```

#### Parameters

- **data** (*ndarray* or *list* or *tuple* or *Dataset* or *Iterable*) – the data to be written. NOTE: If an `h5py.Dataset` is used, all other settings but `link_data` will be ignored as the dataset will either be linked to or copied as is in `H5DataIO`.
- **maxshape** (*tuple*) – Dataset will be resizable up to this shape (Tuple). Automatically enables chunking. Use None for the axes you want to be unlimited.
- **chunks** (*bool* or *tuple*) – Chunk shape or True to enable auto-chunking
- **compression** (*str* or *bool* or *int*) – Compression strategy. If a bool is given, then gzip compression will be used by default. <http://docs.h5py.org/en/latest/high/dataset.html#dataset-compression>
- **compression\_opts** (*int* or *tuple*) – Parameter for compression filter
- **fillvalue** (*None*) – Value to be returned when reading uninitialized parts of the dataset
- **shuffle** (*bool*) – Enable shuffle I/O filter. <http://docs.h5py.org/en/latest/high/dataset.html#dataset-shuffle>



- **fletcher32** (*bool*) – Enable fletcher32 checksum. <http://docs.h5py.org/en/latest/high/dataset.html#dataset-fletcher32>
- **link\_data** (*bool*) – If data is an `h5py.Dataset` should it be linked to or copied. NOTE: This parameter is only allowed if data is an `h5py.Dataset`
- **allow\_plugin\_filters** (*bool*) – Enable passing dynamically loaded filters as compression parameter
- **shape** (*tuple*) – the shape of the new dataset, used only if data is `None`
- **dtype** (*str* or *type* or *dtype*) – the data type of the new dataset, used only if data is `None`

## Module contents

### 6.5.2 Submodules

#### `hdmf.backends.errors` module

Module for I/O backend errors

**exception** `hdmf.backends.errors.UnsupportedOperation`

Bases: `ValueError`

#### `hdmf.backends.io` module

**class** `hdmf.backends.io.HDMFIO`(*manager=None, source=None, herd\_path=None*)

Bases: `object`

##### Parameters

- **manager** (*BuildManager*) – the BuildManager to use for I/O
- **source** (*str* or *Path*) – the source of container being built i.e. file path
- **herd\_path** (*str*) – The path to read/write the HERD file

**abstract static** `can_read`(*path*)

Determines whether a given path is readable by this HDMFIO class

**property** `manager`

The BuildManager this instance is using

**property** `source`

The source of the container being read/written i.e. file path

**read**()

Read a container from the IO source.

##### Returns

the Container object that was read in

##### Return type

*Container*

**write**(*container*, *herd*=None)

**Parameters**

- **container** (*Container*) – the Container object to write
- **herd** (*HERD*) – A HERD object to populate with references.

**export**(*src\_io*, *container*=None, *write\_args*={}, *clear\_cache*=False)

Export from one backend to the backend represented by this class.

If *container* is provided, then the build manager of *src\_io* is used to build the container, and the resulting builder will be exported to the new backend. So if *container* is provided, *src\_io* must have a non-None manager property. If *container* is None, then the contents of *src\_io* will be read and exported to the new backend.

The provided container must be the root of the hierarchy of the source used to read the container (i.e., you cannot read a file and export a part of that file).

Arguments can be passed in for the *write\_builder* method using *write\_args*. Some arguments may not be supported during export.

Example usage:

```
old_io = HDF5IO('old.nwb', 'r')
with HDF5IO('new_copy.nwb', 'w') as new_io:
    new_io.export(old_io)
```

**NOTE: When implementing export support on custom backends. Export does not update the Builder.source**

on the Builders. As such, when writing LinkBuilders we need to determine if LinkBuilder.source and LinkBuilder.builder.source are the same, and if so the link should be internal to the current file (even if the Builder.source points to a different location).

**Parameters**

- **src\_io** (*HDMFIO*) – the HDMFIO object for reading the data to export
- **container** (*Container*) – the Container object to export. If None, then the entire contents of the HDMFIO object will be exported
- **write\_args** (*dict*) – arguments to pass to *write\_builder*
- **clear\_cache** (*bool*) – whether to clear the build manager cache

**abstract read\_builder()**

Read data and return the GroupBuilder representing it

**Returns**

a GroupBuilder representing the read data

**Return type**

GroupBuilder

**abstract write\_builder(builder)**

Write a GroupBuilder representing an Container object

**Parameters**

**builder** (*GroupBuilder*) – the GroupBuilder object representing the Container

**abstract open()**

Open this HDMFIO object for writing of the builder

**abstract close()**

Close this HDMFIO object to further reading/writing

## hdmf.backends.utils module

Module with utility functions and classes used for implementation of I/O backends

**class hdmf.backends.utils.WriteStatusTracker**

Bases: `dict`

Helper class used for tracking the write status of builders. I.e., to track whether a builder has been written or not.

**set\_written(builder)**

Mark this builder as written.

**Parameters**

**builder** (`Builder`) – Builder object to be marked as written

**get\_written(builder)**

Return True if this builder has been written to (or read from) disk by this IO object, False otherwise.

**Parameters**

**builder** (`Builder`) – Builder object to get the written flag for

**Returns**

True if the builder is found in self.\_written\_builders using the builder ID, False otherwise

**class hdmf.backends.utils.NamespaceToBuilderHelper**

Bases: `object`

Helper class used in HDF5IO (and possibly elsewhere) to convert a namespace to a builder for I/O

**classmethod convert\_namespace(ns\_catalog, namespace)**

Convert a namespace to a builder

**Parameters**

- **ns\_catalog** (`NamespaceCatalog`) – the namespace catalog with the specs
- **namespace** (`str`) – the name of the namespace to be converted to a builder

**classmethod get\_source\_name(source)**

**Parameters**

**source** (`str`) – source path

## hdmf.backends.warnings module

**exception** `hdmf.backends.warnings.BrokenLinkWarning`

Bases: `UserWarning`

Raised when a group has a key with a None value.

## 6.5.3 Module contents

## 6.6 hdmf.data\_utils module

`hdmf.data_utils.append_data(data, arg)`

`hdmf.data_utils.extend_data(data, arg)`

Add all the elements of the iterable arg to the end of data.

### Parameters

**data** (*list*, *DataIO*, *numpy.ndarray*, *h5py.Dataset*) – The array to extend

**class** `hdmf.data_utils.AbstractDataChunkIterator`

Bases: `object`

Abstract iterator class used to iterate over DataChunks.

Derived classes must ensure that all abstract methods and abstract properties are implemented, in particular, `dtype`, `maxshape`, `__iter__`, `__next__`, `recommended_chunk_shape`, and `recommended_data_shape`.

Iterating over AbstractContainer objects is not yet supported.

**abstract** `__iter__()`

Return the iterator object

**abstract** `__next__()`

Return the next data chunk or raise a `StopIteration` exception if all chunks have been retrieved.

HINT: `numpy.s_` provides a convenient way to generate index tuples using standard array slicing. This is often useful to define the `DataChunk.selection` of the current chunk

### Returns

`DataChunk` object with the data and selection of the current chunk

### Return type

*DataChunk*

**abstract** `recommended_chunk_shape()`

Recommend the chunk shape for the data array.

### Returns

NumPy-style shape tuple describing the recommended shape for the chunks of the target array or None. This may or may not be the same as the shape of the chunks returned in the iteration process.

**abstract** `recommended_data_shape()`

Recommend the initial shape for the data array.

This is useful in particular to avoid repeated resized of the target array when reading from this data iterator. This should typically be either the final size of the array or the known minimal shape of the array.

**Returns**

NumPy-style shape tuple indicating the recommended initial shape for the target array. This may or may not be the final full shape of the array, i.e., the array is allowed to grow. This should not be None.

**abstract property dtype**

Define the data type of the array

**Returns**

NumPy style dtype or otherwise compliant dtype string

**abstract property maxshape**

Property describing the maximum shape of the data array that is being iterated over

**Returns**

NumPy-style shape tuple indicating the maximum dimensions up to which the dataset may be resized. Axes with None are unlimited.

```
class hdmf.data_utils.GenericDataChunkIterator(buffer_gb=None, buffer_shape=None,
                                              chunk_mb=None, chunk_shape=None,
                                              display_progress=False, progress_bar_options=None)
```

Bases: *AbstractDataChunkIterator*

DataChunkIterator that lets the user specify chunk and buffer shapes.

Break a dataset into buffers containing multiple chunks to be written into an HDF5 dataset.

Basic users should set the `buffer_gb` argument to as much free RAM space as can be safely allocated. Advanced users are offered full control over the shape parameters for the buffer and the chunks; however, the chunk shape must perfectly divide the buffer shape along each axis.

HDF5 recommends chunk size in the range of 2 to 16 MB for optimal cloud performance. <https://youtu.be/rcS5vt-mKok?t=621>

**Parameters**

- **buffer\_gb** (`float` or `int`) – If `buffer_shape` is not specified, it will be inferred as the smallest chunk below the `buffer_gb` threshold. Defaults to 1GB.
- **buffer\_shape** (`tuple`) – Manually defined shape of the buffer.
- **chunk\_mb** (`float` or `int`) – ('If `chunk_shape` is not specified, it will be inferred as the smallest chunk below the `chunk_mb` threshold.', 'Defaults to 10MB.')
- **chunk\_shape** (`tuple`) – Manually defined shape of the chunks.
- **display\_progress** (`bool`) – Display a progress bar with iteration rate and estimated completion time.
- **progress\_bar\_options** (`None`) – Dictionary of keyword arguments to be passed directly to `tqdm`.

**abstract \_get\_data**(*selection: Tuple[slice]*) → `ndarray`

Retrieve the data specified by the selection using minimal I/O.

The developer of a new implementation of the `GenericDataChunkIterator` must ensure the data is actually loaded into memory, and not simply mapped.

**Parameters**

**selection** (`Tuple[slice]`) – tuple of slices, each indicating the selection indexed with respect to `maxshape` for that axis. Each axis of tuple is a slice of the full shape from which to pull data into the buffer.

**Returns**

Array of data specified by selection

**Return type**

`numpy.ndarray`

**abstract** `_get_maxshape()` → `Tuple[int, ...]`

Retrieve the maximum bounds of the data shape using minimal I/O.

**abstract** `_get_dtype()` → `dtype`

Retrieve the dtype of the data using minimal I/O.

**recommended\_chunk\_shape()** → `Tuple[int, ...]`

Recommend the chunk shape for the data array.

**Returns**

NumPy-style shape tuple describing the recommended shape for the chunks of the target array or None. This may or may not be the same as the shape of the chunks returned in the iteration process.

**recommended\_data\_shape()** → `Tuple[int, ...]`

Recommend the initial shape for the data array.

This is useful in particular to avoid repeated resized of the target array when reading from this data iterator. This should typically be either the final size of the array or the known minimal shape of the array.

**Returns**

NumPy-style shape tuple indicating the recommended initial shape for the target array. This may or may not be the final full shape of the array, i.e., the array is allowed to grow. This should not be None.

**property** `maxshape`: `Tuple[int, ...]`

Property describing the maximum shape of the data array that is being iterated over

**Returns**

NumPy-style shape tuple indicating the maximum dimensions up to which the dataset may be resized. Axes with None are unlimited.

**property** `dtype`: `dtype`

Define the data type of the array

**Returns**

NumPy style dtype or otherwise compliant dtype string

**class** `hdmf.data_utils.DataChunkIterator`(*data=None, maxshape=None, dtype=None, buffer\_size=1, iter\_axis=0*)

Bases: `AbstractDataChunkIterator`

Custom iterator class used to iterate over chunks of data.

This default implementation of `AbstractDataChunkIterator` accepts any iterable and assumes that we iterate over a single dimension of the data array (default: the first dimension). `DataChunkIterator` supports buffered read, i.e., multiple values from the input iterator can be combined to a single chunk. This is useful for buffered I/O operations, e.g., to improve performance by accumulating data in memory and writing larger blocks at once.

---

**Note:** `DataChunkIterator` assumes that the iterator that it wraps returns one element along the iteration dimension at a time. I.e., the iterator is expected to return chunks that are one dimension lower than the array itself. For example, when iterating over the first dimension of a dataset with shape (1000, 10, 10), then the iterator would

return 1000 chunks of shape (10, 10) one-chunk-at-a-time. If this pattern does not match your use-case then using [GenericDataChunkIterator](#) or [AbstractDataChunkIterator](#) may be more appropriate.

---

### Initialize the DataChunkIterator.

If 'data' is an iterator and 'dtype' is not specified, then next is called on the iterator in order to determine the dtype of the data.

#### Parameters

- **data** (*None*) – The data object used for iteration
- **maxshape** (*tuple*) – The maximum shape of the full data array. Use None to indicate unlimited dimensions
- **dtype** (*dtype*) – The Numpy data type for the array
- **buffer\_size** (*int*) – Number of values to be buffered in a chunk
- **iter\_axis** (*int*) – The dimension to iterate over

**classmethod** `from_iterable(data=None, maxshape=None, dtype=None, buffer_size=1, iter_axis=0)`

#### Parameters

- **data** (*None*) – The data object used for iteration
- **maxshape** (*tuple*) – The maximum shape of the full data array. Use None to indicate unlimited dimensions
- **dtype** (*dtype*) – The Numpy data type for the array
- **buffer\_size** (*int*) – Number of values to be buffered in a chunk
- **iter\_axis** (*int*) – The dimension to iterate over

### next()

Return the next data chunk or raise a StopIteration exception if all chunks have been retrieved.

---

**Tip:** `numpy.s_` provides a convenient way to generate index tuples using standard array slicing. This is often useful to define the `DataChunk.selection` of the current chunk

---

### Returns

DataChunk object with the data and selection of the current chunk

### Return type

*DataChunk*

### recommended\_chunk\_shape()

Recommend a chunk shape.

To optimize iterative write the chunk should be aligned with the common shape of chunks returned by `__next__` or if those chunks are too large, then a well-aligned subset of those chunks. This may also be any other value in case one wants to recommend chunk shapes to optimize read rather than write. The default implementation returns None, indicating no preferential chunking option.

**recommended\_data\_shape()**

**Recommend an initial shape of the data. This is useful when progressively writing data and**  
we want to recommend an initial size for the dataset

**property maxshape**

Get a shape tuple describing the maximum shape of the array described by this DataChunkIterator.

---

**Note:** If an iterator is provided and no data has been read yet, then the first chunk will be read (i.e., next will be called on the iterator) in order to determine the maxshape. The iterator is expected to return single chunks along the iterator dimension, this means that maxshape will add an additional dimension along the iteration dimension. E.g., if we iterate over the first dimension and the iterator returns chunks of shape (10, 10), then the maxshape would be (None, 10, 10) or (len(self.data), 10, 10), depending on whether size of the iteration dimension is known.

---

**Returns**

Shape tuple. None is used for dimensions where the maximum shape is not known or unlimited.

**property dtype**

Get the value data type

**Returns**

np.dtype object describing the datatype

**class** hdmf.data\_utils.**DataChunk**(data=None, selection=None)

Bases: `object`

Class used to describe a data chunk. Used in DataChunkIterator.

**Parameters**

- **data** (`ndarray`) – Numpy array with the data value(s) of the chunk
- **selection** (`None`) – Numpy index tuple describing the location of the chunk

**astype(dtype)**

Get a new DataChunk with the self.data converted to the given type

**property dtype**

Data type of the values in the chunk

**Returns**

np.dtype of the values in the DataChunk

**get\_min\_bounds()**

Helper function to compute the minimum dataset size required to fit the selection of this chunk.

**Raises**

`TypeError` – If the the selection is not a single int, slice, or tuple of slices.

**Returns**

Tuple with the minimum shape required to store the selection

**hdmf.data\_utils.assertEqualShape**(data1, data2, axes1=None, axes2=None, name1=None, name2=None,  
ignore\_undetermined=True)

Ensure that the shape of data1 and data2 match along the given dimensions

**Parameters**



- **data1** (*List, Tuple, `numpy.ndarray`, `DataChunkIterator`*) – The first input array
- **data2** (*List, Tuple, `numpy.ndarray`, `DataChunkIterator`*) – The second input array
- **name1** – Optional string with the name of data1
- **name2** – Optional string with the name of data2
- **axes1** (*`int`, `Tuple(int)`, `List(int)`, `None`*) – The dimensions of data1 that should be matched to the dimensions of data2. Set to `None` to compare all axes in order.
- **axes2** – The dimensions of data2 that should be matched to the dimensions of data1. Must have the same length as axes1. Set to `None` to compare all axes in order.
- **ignore\_undetermined** – Boolean indicating whether non-matching unlimited dimensions should be ignored, i.e., if two dimension don't match because we can't determine the shape of either one, then should we ignore that case or treat it as no match

#### Returns

Bool indicating whether the check passed and a string with a message about the matching process

```
class hdmf.data_utils.ShapeValidatorResult(result=False, message=None, ignored=(), unmatched=(),
                                           error=None, shape1=(), shape2=(), axes1=(), axes2=())
```

Bases: `object`

Class for storing results from validating the shape of multi-dimensional arrays.

This class is used to store results generated by `ShapeValidator`

#### Variables

- **result** – Boolean indicating whether results matched or not
- **message** – Message indicating the result of the matching procedure

#### Parameters

- **result** (`bool`) – Result of the shape validation
- **message** (`str`) – Message describing the result of the shape validation
- **ignored** (`tuple`) – Axes that have been ignored in the validation process
- **unmatched** (`tuple`) – List of axes that did not match during shape validation
- **error** (`str`) – Error that may have occurred. One of `ERROR_TYPE`
- **shape1** (`tuple`) – Shape of the first array for comparison
- **shape2** (`tuple`) – Shape of the second array for comparison
- **axes1** (`tuple`) – Axes for the first array that should match
- **axes2** (`tuple`) – Axes for the second array that should match

```
SHAPE_ERROR = {'AXIS_LEN_ERROR': 'Unequal length of axes.', 'AXIS_OUT_OF_BOUNDS':
'Axis index for comparison out of bounds.', 'NUM_AXES_ERROR': 'Unequal number of
axes for comparison.', 'NUM_DIMS_ERROR': 'Unequal number of dimensions.', None:
'All required axes matched'}
```

Dict where the Keys are the type of errors that may have occurred during shape comparison and the values are strings with default error messages for the type.

**class** `hdmf.data_utils.DataIO(data=None, dtype=None, shape=None)`

Bases: `object`

Base class for wrapping data arrays for I/O. Derived classes of `DataIO` are typically used to pass dataset-specific I/O parameters to the particular HDMFIO backend.

**Parameters**

- **data** (`ndarray` or `list` or `tuple` or `Dataset` or `Array` or `StrDataset` or `HDMFDataset` or `AbstractDataChunkIterator`) – the data to be written
- **dtype** (`type` or `dtype`) – the data type of the dataset. Not used if data is specified.
- **shape** (`tuple`) – the shape of the dataset. Not used if data is specified.

**get\_io\_params()**

Returns a dict with the I/O parameters specified in this `DataIO`.

**property data**

Get the wrapped data object

**property dtype**

Get the wrapped data object

**property shape**

Get the wrapped data object

**append(arg)**

**extend(arg)**

**\_\_getitem\_\_(item)**

Delegate slicing to the data object

**property valid**

bool indicating if the data object is valid

**exception** `hdmf.data_utils.InvalidDataIOError`

Bases: `Exception`

## 6.7 hdmf.utils module

**class** `hdmf.utils.AllowPositional(value)`

Bases: `Enum`

An enumeration.

**ALLOWED = 1**

**WARNING = 2**

**ERROR = 3**

`hdmf.utils.docval_macro(macro)`

Class decorator to add the class to a list of types associated with the key `macro` in the `__macros` dict

`hdmf.utils.get_docval_macro(key=None)`

Return a deepcopy of the docval macros, i.e., strings that represent a customizable list of types for use in docval.

**Parameters**

**key** – Name of the macro. If `key=None`, then a dictionary of all macros is returned. Otherwise, a tuple of the types associated with the key is returned.

`hdmf.utils.check_type(value, argtype, allow_none=False)`

Check a value against a type

The difference between this function and `isinstance` is that it allows specifying a type as a string. Furthermore, strings allow for specifying more general types, such as a simple numeric type (i.e. `argtype="num"`).

**Parameters**

- **value** (*Any*) – the value to check
- **argtype** (*type*, *str*) – the type to check for
- **allow\_none** (*bool*) – whether or not to allow `None` as a valid value

**Returns**

True if value is a valid instance of argtype

**Return type**

`bool`

`hdmf.utils.get_docval(func, *args)`

Get a copy of docval arguments for a function. If args are supplied, return only docval arguments with value for 'name' key equal to the args

`hdmf.utils.fmt_docval_args(func, kwargs)`

Separate positional and keyword arguments

Useful for methods that wrap other methods

`hdmf.utils.call_docval_func(func, kwargs)`

Call the function with only the keyword arguments that are accepted by the function's docval.

Extra keyword arguments are not passed to the function unless the function's docval has `allow_extra=True`.

`hdmf.utils.docval(*validator, **options)`

A decorator for documenting and enforcing type for instance method arguments.

This decorator takes a list of dictionaries that specify the method parameters. These dictionaries are used for enforcing type and building a Sphinx docstring.

The first arguments are dictionaries that specify the positional arguments and keyword arguments of the decorated function. These dictionaries must contain the following keys: 'name', 'type', and 'doc'. This will define a positional argument. To define a keyword argument, specify a default value using the key 'default'. To validate the dimensions of an input array add the optional 'shape' parameter. To allow a `None` value for an argument, either the default value must be `None` or a different default value must be provided and 'allow\_none': `True` must be passed.

The decorated method must take `self` and `**kwargs` as arguments.

When using this decorator, the functions `getargs` and `popargs` can be used for easily extracting arguments from kwargs.

The following code example demonstrates the use of this decorator:

```

@docval({'name': 'arg1:', 'type': str, 'doc': 'this is the first_
↪positional argument'},
        {'name': 'arg2:', 'type': int, 'doc': 'this is the second_
↪positional argument'},
        {'name': 'kwarg1:', 'type': (list, tuple), 'doc': 'this is a keyword_
↪argument', 'default': list()},
        returns='foo object', rtype='Foo'))
def foo(self, **kwargs):
    arg1, arg2, kwarg1 = getargs('arg1', 'arg2', 'kwarg1', **kwargs)
    ...

```

### Parameters

- **enforce\_type** – Enforce types of input parameters (Default=True)
- **returns** – String describing the return values
- **rtype** – String describing the data type of the return values
- **is\_method** – True if this is decorating an instance or class method, False otherwise (Default=True)
- **enforce\_shape** – Enforce the dimensions of input arrays (Default=True)
- **validator** – `dict` objects specifying the method parameters
- **allow\_extra** – Allow extra arguments (Default=False)
- **allow\_positional** – Allow positional arguments (Default=True)
- **options** – additional options for documenting and validating method parameters

`hdmf.utils.getargs(*argnames, argdict)`

Convenience function to retrieve arguments from a dictionary in batch.

The last argument should be a dictionary, and the other arguments should be the keys (argument names) for which to retrieve the values.

### Raises

**ValueError** – if a argument name is not found in the dictionary or there is only one argument passed to this function or the last argument is not a dictionary

### Returns

a single value if there is only one argument, or a list of values corresponding to the given argument names

`hdmf.utils.popargs(*argnames, argdict)`

Convenience function to retrieve and remove arguments from a dictionary in batch.

The last argument should be a dictionary, and the other arguments should be the keys (argument names) for which to retrieve the values.

### Raises

**ValueError** – if a argument name is not found in the dictionary or there is only one argument passed to this function or the last argument is not a dictionary

### Returns

a single value if there is only one argument, or a list of values corresponding to the given argument names

`hdmf.utils.popargs_to_dict(keys, argdict)`

Convenience function to retrieve and remove arguments from a dictionary in batch into a dictionary.

Same as `{key: argdict.pop(key) for key in keys}` with a custom `ValueError`

#### Parameters

- **keys** (*Iterable*) – Iterable of keys to pull out of `argdict`
- **argdict** – Dictionary to process

#### Raises

**ValueError** – if an argument name is not found in the dictionary

#### Returns

a dict of arguments removed

**class** `hdmf.utils.ExtenderMeta(name, bases, namespace, **kwargs)`

Bases: `ABCMeta`

A metaclass that will extend the base class initialization routine by executing additional functions defined in classes that use this metaclass

In general, this class should only be used by core developers.

**classmethod** `pre_init(func)`

A decorator that sets a `__preinit` attribute on the target function and then returns the function as a class-method.

**classmethod** `post_init(func)`

A decorator for defining a routine to run after creation of a type object.

An example use of this method would be to define a classmethod that gathers any defined methods or attributes after the base Python type construction (i.e. after `type` has been called)

`hdmf.utils.get_data_shape(data, strict_no_data_load=False)`

Helper function used to determine the shape of the given array.

In order to determine the shape of nested tuples, lists, and sets, this function recursively inspects elements along the dimensions, assuming that the data has a regular, rectangular shape. In the case of out-of-core iterators, this means that the first item along each dimension would potentially be loaded into memory. Set `strict_no_data_load=True` to enforce that this does not happen, at the cost that we may not be able to determine the shape of the array.

#### Parameters

- **data** (*List*, *numpy.ndarray*, *DataChunkIterator*) – Array for which we should determine the shape. Can be any object that supports `__len__` or `.shape`.
- **strict\_no\_data\_load** – If True and data is an out-of-core iterator, None may be returned. If False (default), the first element of data may be loaded into memory.

#### Returns

Tuple of ints indicating the size of known dimensions. Dimensions for which the size is unknown will be set to None.

`hdmf.utils.pystr(s)`

Convert a string of characters to Python str object

`hdmf.utils.to_uint_array(arr)`

Convert a numpy array or array-like object to a numpy array of unsigned integers with the same dtype itemsize.

For example, a list of int32 values is converted to a numpy array with dtype uint32. :raises ValueError: if input array contains values that are not unsigned integers or non-negative integers.

`hdmf.utils.is_ragged(data)`

Test whether a list of lists or array is ragged / jagged

**class** `hdmf.utils.LabelledDict(label, key_attr='name', add_callable=None, remove_callable=None)`

Bases: `dict`

A dict wrapper that allows querying by an attribute of the values and running a callable on removed items.

For example, if the key attribute is set as 'name' in `__init__`, then all objects added to the `LabelledDict` must have a 'name' attribute and a particular object in the `LabelledDict` can be accessed using the syntax `['object_name']` if `object.name == 'object_name'`. In this way, `LabelledDict` acts like a set where values can be retrieved using square brackets around the value of the key attribute. An 'add' method makes clear the association between the key attribute of the `LabelledDict` and the values of the `LabelledDict`.

`LabelledDict` also supports retrieval of values with the syntax `my_dict['attr == val']`, which returns a set of objects in the `LabelledDict` which have an attribute 'attr' with a string value 'val'. If no objects match that condition, a `KeyError` is raised. Note that if 'attr' equals the key attribute, then the single matching value is returned, not a set.

`LabelledDict` does not support changing items that have already been set. A `TypeError` will be raised when using `__setitem__` on keys that already exist in the dict. The `setdefault` and `update` methods are not supported. A `TypeError` will be raised when these are called.

A callable function may be passed to the constructor to be run on an item after adding it to this dict using the `__setitem__` and `add` methods.

A callable function may be passed to the constructor to be run on an item after removing it from this dict using the `__delitem__` (the `del` operator), `pop`, and `popitem` methods. It will also be run on each removed item when using the `clear` method.

#### Usage:

```
LabelledDict(label='my_objects', key_attr='name') my_dict[obj.name] = obj my_dict.add(obj) # simpler syntax
```

### Example

```
# MyTestClass is a class with attributes 'prop1' and 'prop2'. MyTestClass.__init__ sets those attributes. ld =
LabelledDict(label='all_objects', key_attr='prop1') obj1 = MyTestClass('a', 'b') obj2 = MyTestClass('d', 'b')
ld[obj1.prop1] = obj1 # obj1 is added to the LabelledDict with the key obj1.prop1. Any other key is not allowed.
ld.add(obj2) # Simpler 'add' syntax enforces the required relationship ld['a'] # Returns obj1 ld['prop1 == a'] #
Also returns obj1 ld['prop2 == b'] # Returns set([obj1, obj2]) - the set of all values v in ld where v.prop2 == 'b'
```

#### Parameters

- **label** (`str`) – the label on this dictionary
- **key\_attr** (`str`) – the attribute name to use as the key
- **add\_callable** (`function`) – function to call on an element after adding it to this dict using the `add` or `__setitem__` methods
- **remove\_callable** (`function`) – function to call on an element after removing it from this dict using the `pop`, `popitem`, `clear`, or `__delitem__` methods

**property label**

Return the label of this LabelledDict

**property key\_attr**

Return the attribute used as the key for values in this LabelledDict

**\_\_getitem\_\_(args)**

Get a value from the LabelledDict with the given key.

Supports syntax `my_dict['attr == val']`, which returns a set of objects in the LabelledDict which have an attribute 'attr' with a string value 'val'. If no objects match that condition, an empty set is returned. Note that if 'attr' equals the key attribute of this LabelledDict, then the single matching value is returned, not a set.

**add(value)**

Add a value to the dict with the key `value.key_attr`.

Raises `ValueError` if value does not have attribute `key_attr`.

**pop(k)**

Remove an item that matches the key. If `remove_callable` was initialized, call that on the returned value.

**popitem()**

Remove the last added item. If `remove_callable` was initialized, call that on the returned value.

Note: `popitem` returns a tuple (key, value) but the `remove_callable` will be called only on the value.

Note: in Python 3.5 and earlier, dictionaries are not ordered, so `popitem` removes an arbitrary item.

**clear()**

Remove all items. If `remove_callable` was initialized, call that on each returned value.

The order of removal depends on the `popitem` method.

**setdefault(k)**

`setdefault` is not supported. A `TypeError` will be raised.

**update(other)**

`update` is not supported. A `TypeError` will be raised.

**class** `hdmf.utils.StrDataset(dset, encoding, errors='strict')`

Bases: `Dataset`

Wrapper to decode strings on reading the dataset

Create a new `Dataset` object by binding to a low-level `DatasetID`.

**\_\_getitem\_\_(args)**

Read a slice from the HDF5 dataset.

Takes slices and recarray-style field names (more than one is allowed!) in any order. Obeys basic NumPy rules, including broadcasting.

Also supports:

- Boolean “mask” array indexing

## 6.8 hdmf.validate package

### 6.8.1 Submodules

#### hdmf.validate.errors module

**class** hdmf.validate.errors.**Error**(*name, reason, location=None*)

Bases: [object](#)

##### Parameters

- **name** ([str](#)) – the name of the component that is erroneous
- **reason** ([str](#)) – the reason for the error
- **location** ([str](#)) – the location of the error

property **name**

property **reason**

property **location**

**class** hdmf.validate.errors.**DtypeError**(*name, expected, received, location=None*)

Bases: [Error](#)

##### Parameters

- **name** ([str](#)) – the name of the component that is erroneous
- **expected** ([dtype](#) or [type](#) or [str](#) or [list](#)) – the expected dtype
- **received** ([dtype](#) or [type](#) or [str](#) or [list](#)) – the received dtype
- **location** ([str](#)) – the location of the error

**class** hdmf.validate.errors.**MissingError**(*name, location=None*)

Bases: [Error](#)

##### Parameters

- **name** ([str](#)) – the name of the component that is erroneous
- **location** ([str](#)) – the location of the error

**class** hdmf.validate.errors.**ExpectedArrayError**(*name, expected, received, location=None*)

Bases: [Error](#)

##### Parameters

- **name** ([str](#)) – the name of the component that is erroneous
- **expected** ([tuple](#) or [list](#)) – the expected shape
- **received** ([str](#)) – the received data
- **location** ([str](#)) – the location of the error

**class** hdmf.validate.errors.**ShapeError**(*name, expected, received, location=None*)

Bases: [Error](#)

##### Parameters

- **name** ([str](#)) – the name of the component that is erroneous



- **expected** (`tuple` or `list`) – the expected shape
- **received** (`tuple` or `list`) – the received shape
- **location** (`str`) – the location of the error

**class** hdmf.validate.errors.**MissingDataType**(*name, data\_type, location=None, missing\_dt\_name=None*)

Bases: [Error](#)

#### Parameters

- **name** (`str`) – the name of the component that is erroneous
- **data\_type** (`str`) – the missing data type
- **location** (`str`) – the location of the error
- **missing\_dt\_name** (`str`) – the name of the missing data type

property `data_type`

**class** hdmf.validate.errors.**IllegalLinkError**(*name, location=None*)

Bases: [Error](#)

A validation error for indicating that a link was used where an actual object (i.e. a dataset or a group) must be used

#### Parameters

- **name** (`str`) – the name of the component that is erroneous
- **location** (`str`) – the location of the error

**class** hdmf.validate.errors.**IncorrectDataType**(*name, expected, received, location=None*)

Bases: [Error](#)

A validation error for indicating that the incorrect `data_type` (not `dtype`) was used.

#### Parameters

- **name** (`str`) – the name of the component that is erroneous
- **expected** (`str`) – the expected `data_type`
- **received** (`str`) – the received `data_type`
- **location** (`str`) – the location of the error

**class** hdmf.validate.errors.**IncorrectQuantityError**(*name, data\_type, expected, received, location=None*)

Bases: [Error](#)

A validation error indicating that a child group/dataset/link has the incorrect quantity of matching elements

#### Parameters

- **name** (`str`) – the name of the component that is erroneous
- **data\_type** (`str`) – the data type which has the incorrect quantity
- **expected** (`str` or `int`) – the expected quantity
- **received** (`str` or `int`) – the received quantity
- **location** (`str`) – the location of the error

**hdmf.validate.validator module**

`hdmf.validate.validator.check_type(expected, received, string_format=None)`

*expected* should come from the spec *received* should come from the data

`hdmf.validate.validator.get_iso8601_regex()`

`hdmf.validate.validator.get_string_format(data)`

Return the string format of the given data. Possible outputs are “isodatetime” and None.

**exception** `hdmf.validate.validator.EmptyArrayError`

Bases: `Exception`

`hdmf.validate.validator.get_type(data, builder_dtype=None)`

Return a tuple of (the string representation of the type, the format of the string data) for the given data.

`hdmf.validate.validator.check_shape(expected, received)`

**class** `hdmf.validate.validator.ValidatorMap(namespace)`

Bases: `object`

A class for keeping track of Validator objects for all data types in a namespace

**Parameters**

**namespace** (`SpecNamespace`) – the namespace to builder map for

**property namespace**

**valid\_types(spec)**

Get all valid types for a given data type

**Parameters**

**spec** (`Spec` or `str`) – the specification to use to validate

**Returns**

all valid sub data types for the given spec

**Return type**

`tuple`

**get\_validator(data\_type)**

Return the validator for a given data type

**Parameters**

**data\_type** (`BaseStorageSpec` or `str`) – the data type to get the validator for

**validate(builder)**

Validate a builder against a Spec

*builder* must have the attribute used to specifying data type by the namespace used to construct this ValidatorMap.

**Parameters**

**builder** (`BaseBuilder`) – the builder to validate

**Returns**

a list of errors found

**Return type**

`list`

**class** hdmf.validate.validator.**Validator**(*spec*, *validator\_map*)

Bases: [object](#)

A base class for classes that will be used to validate against Spec subclasses

**Parameters**

- **spec** ([Spec](#)) – the specification to use to validate
- **validator\_map** ([ValidatorMap](#)) – the ValidatorMap to use during validation

**property** *spec*

**property** *vmap*

**abstract** *validate*(*value*)

**Parameters**

**value** (*None*) – either in the form of a value or a Builder

**Returns**

a list of Errors

**Return type**

[list](#)

**classmethod** *get\_spec\_loc*(*spec*)

**classmethod** *get\_builder\_loc*(*builder*)

**class** hdmf.validate.validator.**AttributeValidator**(*spec*, *validator\_map*)

Bases: [Validator](#)

A class for validating values against AttributeSpecs

**Parameters**

- **spec** ([AttributeSpec](#)) – the specification to use to validate
- **validator\_map** ([ValidatorMap](#)) – the ValidatorMap to use during validation

**validate**(*value*)

**Parameters**

**value** (*None*) – the value to validate

**Returns**

a list of Errors

**Return type**

[list](#)

**class** hdmf.validate.validator.**BaseStorageValidator**(*spec*, *validator\_map*)

Bases: [Validator](#)

A base class for validating against Spec objects that have attributes i.e. BaseStorageSpec

**Parameters**

- **spec** ([BaseStorageSpec](#)) – the specification to use to validate
- **validator\_map** ([ValidatorMap](#)) – the ValidatorMap to use during validation

**validate**(*builder*)

**Parameters**

**builder** (*BaseBuilder*) – the builder to validate

**Returns**

a list of Errors

**Return type**

*list*

**class** hdmf.validate.validator.**DatasetValidator**(*spec, validator\_map*)

Bases: *BaseStorageValidator*

A class for validating DatasetBuilders against DatasetSpecs

**Parameters**

- **spec** (*DatasetSpec*) – the specification to use to validate
- **validator\_map** (*ValidatorMap*) – the ValidatorMap to use during validation

**validate**(*builder*)

**Parameters**

**builder** (*DatasetBuilder*) – the builder to validate

**Returns**

a list of Errors

**Return type**

*list*

**class** hdmf.validate.validator.**GroupValidator**(*spec, validator\_map*)

Bases: *BaseStorageValidator*

A class for validating GroupBuilders against GroupSpecs

**Parameters**

- **spec** (*GroupSpec*) – the specification to use to validate
- **validator\_map** (*ValidatorMap*) – the ValidatorMap to use during validation

**validate**(*builder*)

**Parameters**

**builder** (*GroupBuilder*) – the builder to validate

**Returns**

a list of Errors

**Return type**

*list*

**class** hdmf.validate.validator.**SpecMatches**(*spec*)

Bases: *object*

A utility class to hold a spec and the builders matched to it

**add**(*builder*)

**class** `hdmf.validate.validator.SpecMatcher(vmap, specs)`

Bases: `object`

Matches a set of builders against a set of specs

This class is intended to isolate the task of choosing which spec a builder should be validated against from the task of performing that validation.

**property** `unmatched_builders`

Returns the builders for which no matching spec was found

These builders can be considered superfluous, and will generate a warning in the future.

**property** `spec_matches`

Returns a list of tuples of (spec, assigned builders)

**assign\_to\_specs**(*builders*)

Assigns a set of builders against a set of specs (many-to-one)

In the case that no matching spec is found, a builder will be added to a list of unmatched builders.

## 6.8.2 Module contents

# 6.9 hdmf.testing package

## 6.9.1 Submodules

### `hdmf.testing.testcase` module

**class** `hdmf.testing.testcase.TestCase(methodName='runTest')`

Bases: `TestCase`

Extension of unittest's `TestCase` to add useful functions for unit testing in HDMF.

Create an instance of the class that will use the named test method when executed. Raises a `ValueError` if the instance does not have a method with the specified name.

**assertRaisesWith**(*exc\_type, exc\_msg, \*args, \*\*kwargs*)

Asserts the given invocation raises the expected exception. This is similar to unittest's `assertRaises` and `assertRaisesRegex`, but checks for an exact match.

**assertWarnsWith**(*warn\_type, exc\_msg, \*args, \*\*kwargs*)

Asserts the given invocation raises the expected warning. This is similar to unittest's `assertWarns` and `assertWarnsRegex`, but checks for an exact match.

**assertContainerEqual**(*container1, container2, ignore\_name=False, ignore\_hdmf\_attrs=False, ignore\_string\_to\_byte=False, message=None*)

Asserts that the two `AbstractContainers` have equal contents. This applies to both `Container` and `Data` types.

#### Parameters

- **container1** (`AbstractContainer`) – First container
- **container2** (`AbstractContainer`) – Second container to compare with container 1
- **ignore\_name** – whether to ignore testing equality of name of the top-level container

- **ignore\_hdmf\_attrs** – whether to ignore testing equality of HDMF container attributes, such as `container_source` and `object_id`
- **ignore\_string\_to\_byte** – ignore conversion of str to bytes and compare as unicode instead
- **message** – custom additional message to show when assertions as part of this assert are failing

**assertBuilderEqual**(*builder1*, *builder2*, *check\_path=True*, *check\_source=True*, *message=None*)

Test whether two builders are equal. Like `assertDictEqual` but also checks type, name, path, and source.

**Parameters**

- **builder1** (*Builder*) – The first builder
- **builder2** (*Builder*) – The second builder
- **check\_path** (*bool*) – Check that the builder.path values are equal
- **check\_source** (*bool*) – Check that the builder.source values are equal
- **message** (*str* or *None*) – Custom message to add when any asserts as part of this assert are failing (default=None)

**class** `hdmf.testing.testcase.H5RoundTripMixin`

Bases: `object`

Mixin class for methods to run a roundtrip test writing a container to and reading the container from an HDF5 file. The `setUp`, `test_roundtrip`, and `tearDown` methods will be run by unittest.

The abstract method `setUpContainer` needs to be implemented by classes that include this mixin.

Example:

```
class TestMyContainerRoundTrip(H5RoundTripMixin, TestCase):
    def setUpContainer(self):
        # return the Container to read/write
```

NOTE: This class is a mix-in and not a subclass of `TestCase` so that unittest does not discover it, try to run it, and skip it.

**setUp()**

**tearDown()**

**abstract setUpContainer()**

Return the Container to read/write.

**test\_roundtrip()**

Test whether the container read from a written file is the same as the original file.

**test\_roundtrip\_export()**

Test whether the container read from a written and then exported file is the same as the original file.

**roundtripContainer**(*cache\_spec=False*)

Write the container to an HDF5 file, read the container from the file, and return it.

**roundtripExportContainer**(*cache\_spec=False*)

Write the container to an HDF5 file, read it, export it to a new file, read that file, and return it.

**validate**(*experimental=False*)

Validate the written and exported files, if they exist.

## hdmf.testing.utils module

`hdmf.testing.utils.remove_test_file(path)`

A helper function for removing intermediate test files

This checks if the environment variable `CLEAN_HDMF` has been set to `False` before removing the file. If `CLEAN_HDMF` is set to `False`, it does not remove the file.

## hdmf.testing.validate\_spec module

`hdmf.testing.validate_spec.validate_spec(fpath_spec, fpath_schema)`

Validate a yaml specification file against the json schema file that defines the specification language. Can be used to validate changes to the NWB and HDMF core schemas, as well as any extensions to either.

### Parameters

- `fpath_spec` – path-like
- `fpath_schema` – path-like

`hdmf.testing.validate_spec.main()`

## 6.9.2 Module contents

## 6.10 hdmf package

### 6.10.1 Subpackages

### 6.10.2 Submodules

## hdmf.array module

`class hdmf.array.Array(data)`

Bases: `object`

**property** `data`

`get_data()`

`__getitem__(arg)`

`class hdmf.array.AbstractSortedArray(data)`

Bases: `Array`

An abstract class for representing sorted array

**abstract** `find_point(val)`

`get_data()`

`class hdmf.array.SortedArray(array)`

Bases: `AbstractSortedArray`

A class for wrapping sorted arrays. This class overrides `<`, `>`, `<=`, `>=`, `==`, and `!=` to leverage the sorted content for efficiency.

**find\_point**(*val*)

**class** `hdmf.array.LinSpace`(*start*, *stop*, *step*)

Bases: [\*SortedArray\*](#)

**find\_point**(*val*)

## **hdmf.monitor module**

**exception** `hdmf.monitor.NotYetExhausted`

Bases: [\*Exception\*](#)

**class** `hdmf.monitor.DataChunkProcessor`(*data*)

Bases: [\*AbstractDataChunkIterator\*](#)

Initialize the DataChunkIterator

### **Parameters**

**data** ([\*DataChunkIterator\*](#)) – the DataChunkIterator to analyze

**recommended\_chunk\_shape**()

Recommend the chunk shape for the data array.

### **Returns**

NumPy-style shape tuple describing the recommended shape for the chunks of the target array or None. This may or may not be the same as the shape of the chunks returned in the iteration process.

**recommended\_data\_shape**()

Recommend the initial shape for the data array.

This is useful in particular to avoid repeated resized of the target array when reading from this data iterator. This should typically be either the final size of the array or the known minimal shape of the array.

### **Returns**

NumPy-style shape tuple indicating the recommended initial shape for the target array. This may or may not be the final full shape of the array, i.e., the array is allowed to grow. This should not be None.

**get\_final\_result**(\*\**kwargs*)

Return the result of processing data fed by this DataChunkIterator

**abstract process\_data\_chunk**(*data\_chunk*)

**This method should take in a DataChunk,**  
and process it.

### **Parameters**

**data\_chunk** ([\*DataChunk\*](#)) – a chunk to process

**abstract compute\_final\_result**()

**Return the result of processing this stream**

Should raise NotYetExhausted exception



```
class hdmf.monitor.NumSampleCounter(data)
```

Bases: [DataChunkProcessor](#)

Initialize the DataChunkIterator

**Parameters**

**data** ([DataChunkIterator](#)) – the DataChunkIterator to analyze

```
process_data_chunk(data_chunk)
```

**Parameters**

**data\_chunk** ([DataChunk](#)) – a chunk to process

```
compute_final_result()
```

## hdmf.query module

```
class hdmf.query.Query(obj, op, arg)
```

Bases: [object](#)

```
evaluate(expand=True)
```

**Parameters**

**expand** ([bool](#)) – None

```
class hdmf.query.HDMFDataset(dataset)
```

Bases: [object](#)

**Parameters**

**dataset** ([ndarray](#) or [list](#) or [tuple](#) or [Dataset](#) or [Array](#) or [StrDataset](#) or [Array](#)) – the HDF5 file lazily evaluate

```
__getitem__(key)
```

```
property dataset
```

```
property dtype
```

```
next()
```

```
class hdmf.query.ReferenceResolver
```

Bases: [object](#)

A base class for classes that resolve references

```
abstract classmethod get_inverse_class()
```

Return the class the represents the ReferenceResolver that resolves references to the opposite type.

BuilderResolver.get\_inverse\_class should return a class that subclasses ContainerResolver.

ContainerResolver.get\_inverse\_class should return a class that subclasses BuilderResolver.

```
abstract invert()
```

Return an object that defers reference resolution but in the opposite direction.

```
class hdmf.query.BuilderResolver
```

Bases: [ReferenceResolver](#)

A reference resolver that resolves references to Builders

Subclasses should implement the invert method and the get\_inverse\_class classmethod

BuilderResolver.get\_inverse\_class should return a class that subclasses ContainerResolver.

**class** hdmf.query.ContainerResolver

Bases: [ReferenceResolver](#)

A reference resolver that resolves references to Containers

Subclasses should implement the invert method and the get\_inverse\_class classmethod

ContainerResolver.get\_inverse\_class should return a class that subclasses BuilderResolver.

## hdmf.region module

**class** hdmf.region.RegionSlicer(*target, slice*)

Bases: [DataRegion](#)

A abstract base class to control getting using a region

Subclasses must implement `__getitem__` and `__len__`

### Parameters

- **target** (*None*) – the target to slice
- **slice** (*None*) – the region to slice

### property data

The target data. Same as self.target

### property region

The selected region. Same as self.slice

### property target

The target data

### property slice

The selected slice

### abstract property \_\_getitem\_\_

Must be implemented by subclasses

### abstract property \_\_len\_\_

Must be implemented by subclasses

**class** hdmf.region.ListSlicer(*dataset, region*)

Bases: [RegionSlicer](#)

Implementation of RegionSlicer for slicing Lists and Data

### Parameters

- **dataset** (*list* or *tuple* or [Data](#)) – the dataset to slice
- **region** (*list* or *tuple* or *slice*) – the region reference to use to slice

### \_\_getitem\_\_(*idx*)

Get data values from selected data

## hdmf.term\_set module

**class** hdmf.term\_set.**TermSet**(*term\_schema\_path*: *str* | *None* = *None*, *schemasheets\_folder*: *str* | *None* = *None*, *dynamic*: *bool* = *False*)

Bases: `object`

Class for implementing term sets from ontologies and other resources used to define the meaning and/or identify of terms.

### Variables

- **term\_schema\_path** – The path to the LinkML YAML enumeration schema
- **sources** – The prefixes for the ontologies used in the TermSet
- **view** – SchemaView of the term set schema
- **schemasheets\_folder** – The path to the folder containing the LinkML TSV files
- **expanded\_termset\_path** – The path to the schema with the expanded enumerations

### Parameters

- **term\_schema\_path** – The path to the LinkML YAML enumeration schema
- **schemasheets\_folder** – The path to the folder containing the LinkML TSV files
- **dynamic** – Boolean parameter denoting whether the schema uses Dynamic Enumerations

**validate**(*term*)

Validate term in dataset towards a termset.

### Parameters

**term** (*str*) – term to be validated

**property view\_set**

Property method to return a view of all terms in the the LinkML YAML Schema.

**\_\_getitem\_\_**(*term*)

Method to retrieve a term and term information (LinkML description and LinkML meaning) from the set of terms.

**class** hdmf.term\_set.**TermSetWrapper**(*termset*, *value*, *field*=*None*)

Bases: `object`

This class allows any HDF5 dataset or attribute to have a TermSet.

### Parameters

- **termset** (*TermSet*) – The TermSet to be used.
- **value** (*list* or *ndarray* or *dict* or *str* or *tuple*) – The target item that is wrapped, either data or attribute.
- **field** (*str*) – The field within a compound array.

**property value**

**property field**

**property termset**

**property dtype**

**\_\_getitem\_\_**(*val*)

This is used when we want to index items.

**append**(*arg*)

This append resolves the wrapper to use the append of the container using the wrapper.

**extend**(*arg*)

This append resolves the wrapper to use the extend of the container using the wrapper.

**class** `hdmf.term_set.TypeConfigurator`(*path=None*)

Bases: `object`

This class allows users to toggle on/off a global configuration for defined data types. When toggled on, every instance of a configuration file supported data type will be validated according to the corresponding TermSet.

**Parameters**

**path** (`str`) – Path to the configuration file.

**get\_config**(*data\_type, namespace*)

Return the config for that data type in the given namespace.

**Parameters**

- **data\_type** (`str`) – The desired data type within the configuration file.
- **namespace** (`str`) – The namespace for the data type.

**load\_type\_config**(*config\_path*)

Load the configuration file for validation on the fields defined for the objects within the file.

**Parameters**

**config\_path** (`str`) – Path to the configuration file.

**unload\_type\_config**()

Remove validation according to termset configuration file.

### 6.10.3 Module contents

`hdmf.get_region_slicer`(*dataset, region*)

**Parameters**

- **dataset** (*None*) – the HDF5 dataset to slice
- **region** (*None*) – the region reference to use to slice

modindex

## EXTENDING STANDARDS

The following page will discuss how to extend a standard using HDMF.

### 7.1 Creating new Extensions

Standards specified using HDMF are designed to be extended. Extension for a standard can be done so using classes provided in the `hdmf.spec` module. The classes `GroupSpec`, `DatasetSpec`, `AttributeSpec`, and `LinkSpec` can be used to define custom types.

#### 7.1.1 Attribute Specifications

Specifying attributes is done with `AttributeSpec`.

```
from hdmf.spec import AttributeSpec

spec = AttributeSpec('bar', 'a value for bar', 'float')
```

#### 7.1.2 Dataset Specifications

Specifying datasets is done with `DatasetSpec`.

```
from hdmf.spec import DatasetSpec

spec = DatasetSpec('A custom data type',
                   name='qux',
                   attribute=[
                       AttributeSpec('baz', 'a value for baz', 'str'),
                   ],
                   shape=(None, None))
```

## Using datasets to specify tables

Tables can be specified using *DtypeSpec*. To specify a table, provide a list of *DtypeSpec* objects to the *dtype* argument.

```
from hdmf.spec import DatasetSpec, DtypeSpec

spec = DatasetSpec('A custom data type',
                   name='qux',
                   attribute=[
                       AttributeSpec('baz', 'a value for baz', 'str'),
                   ],
                   dtype=[
                       DtypeSpec('foo', 'column for foo', 'int'),
                       DtypeSpec('bar', 'a column for bar', 'float')
                   ])

```

### 7.1.3 Group Specifications

Specifying groups is done with the *GroupSpec* class.

```
from hdmf.spec import GroupSpec

spec = GroupSpec('A custom data type',
                 name='quux',
                 attributes=[...],
                 datasets=[...],
                 groups=[...])

```

### 7.1.4 Data Type Specifications

*GroupSpec* and *DatasetSpec* use the arguments *data\_type\_inc* and *data\_type\_def* for declaring new types and extending existing types. New types are specified by setting the argument *data\_type\_def*. New types can extend an existing type by specifying the argument *data\_type\_inc*.

Create a new type

```
from hdmf.spec import GroupSpec

# A list of AttributeSpec objects to specify new attributes
addl_attributes = [...]
# A list of DatasetSpec objects to specify new datasets
addl_datasets = [...]
# A list of DatasetSpec objects to specify new groups
addl_groups = [...]
spec = GroupSpec('A custom data type',
                 attributes=addl_attributes,
                 datasets=addl_datasets,
                 groups=addl_groups,
                 data_type_def='MyNewType')

```

Extend an existing type

```

from hdmf.spec import GroupSpec

# A list of AttributeSpec objects to specify additional attributes or attributes to be
↳ overridden
addl_attributes = [...]
# A list of DatasetSpec objects to specify additional datasets or datasets to be
↳ overridden
addl_datasets = [...]
# A list of GroupSpec objects to specify additional groups or groups to be overridden
addl_groups = [...]
spec = GroupSpec('An extended data type',
                  attributes=addl_attributes,
                  datasets=addl_datasets,
                  groups=addl_groups,
                  data_type_inc='SpikeEventSeries',
                  data_type_def='MyExtendedSpikeEventSeries')

```

Existing types can be instantiated by specifying *data\_type\_inc* alone.

```

from hdmf.spec import GroupSpec

# use another GroupSpec object to specify that a group of type
# ElectricalSeries should be present in the new type defined below
addl_groups = [ GroupSpec('An included ElectricalSeries instance',
                           data_type_inc='ElectricalSeries') ]

spec = GroupSpec('An extended data type',
                  groups=addl_groups,
                  data_type_inc='SpikeEventSeries',
                  data_type_def='MyExtendedSpikeEventSeries')

```

Datasets can be extended in the same manner (with regard to *data\_type\_inc* and *data\_type\_def*, by using the class *DatasetSpec*.

## 7.2 Saving Extensions

Extensions are used by including them in a loaded namespace. Namespaces and extensions need to be saved to file for downstream use. The class *NamespaceBuilder* can be used to create new namespace and specification files.

Create a new namespace with extensions

```

from hdmf.spec import GroupSpec, NamespaceBuilder

# create a builder for the namespace
ns_builder = NamespaceBuilder("Extension for use in my laboratory", "mylab", ...)

# create extensions
ext1 = GroupSpec('A custom SpikeEventSeries interface',
                  attributes=[...],
                  datasets=[...],
                  groups=[...],
                  data_type_inc='SpikeEventSeries',

```

(continues on next page)

(continued from previous page)

```
data_type_def='MyExtendedSpikeEventSeries')

ext2 = GroupSpec('A custom EventDetection interface',
                 attributes=[...],
                 datasets=[...],
                 groups=[...],
                 data_type_inc='EventDetection',
                 data_type_def='MyExtendedEventDetection')

# add the extension
ext_source = 'mylab.specs.yaml'
ns_builder.add_spec(ext_source, ext1)
ns_builder.add_spec(ext_source, ext2)

# include an existing namespace - this will include all specifications in that namespace
ns_builder.include_namespace('collab_ns')

# save the namespace and extensions
ns_path = 'mylab.namespace.yaml'
ns_builder.export(ns_path)
```

---

**Tip:** Using the API to generate extensions (rather than writing YAML sources directly) helps avoid errors in the specification (e.g., due to missing required keys or invalid values) and ensure compliance of the extension definition with the HDMF specification language. It also helps with maintenance of extensions, e.g., if extensions have to be ported to newer versions of the [specification language](#) in the future.

---

## 7.3 Incorporating extensions

HDMF supports extending existing data types. Extensions must be registered with HDMF to be used for reading and writing of custom data types.

The following code demonstrates how to load custom namespaces.

```
from hdmf import load_namespaces
namespace_path = 'my_namespace.yaml'
load_namespaces(namespace_path)
```

---

**Note:** This will register all namespaces defined in the file 'my\_namespace.yaml'.

---



### 7.3.1 Container : Representing custom data

To read and write custom data, corresponding *Container* classes must be associated with their respective specifications. *Container* classes are associated with their respective specification using the decorator *register\_class*.

The following code demonstrates how to associate a specification with the *Container* class that represents it.

```
from hdmf.common import register_class
from hdmf.container import Container

@register_class('MyExtension', 'my_namespace')
class MyExtensionContainer(Container):
    ...
```

*register\_class* can also be used as a function.

```
from hdmf.common import register_class
from hdmf.container import Container

class MyExtensionContainer(Container):
    ...

register_class(data_type='MyExtension', namespace='my_namespace', container_
    ↪cls=MyExtensionContainer)
```

If you do not have an *Container* subclass to associate with your extension specification, a dynamically created class is created by default.

To use the dynamic class, you will need to retrieve the class object using the function *get\_class*. Once you have retrieved the class object, you can use it just like you would a statically defined class.

```
from hdmf.common import get_class
MyExtensionContainer = get_class('my_namespace', 'MyExtension')
my_ext_inst = MyExtensionContainer(...)
```

If using iPython, you can access documentation for the class's constructor using the help command.

### 7.3.2 ObjectMapper : Customizing the mapping between Container and the Spec

If your *Container* extension requires custom mapping of the *Container* class for reading and writing, you will need to implement and register a custom *ObjectMapper*.

*ObjectMapper* extensions are registered with the decorator *register\_map*.

```
from hdmf.common import register_map
from hdmf.build import ObjectMapper

@register_map(MyExtensionContainer)
class MyExtensionMapper(ObjectMapper):
    ...
```

*register\_map* can also be used as a function.

```
from hdmf.common import register_map
from hdmf.build import ObjectMapper
```

(continues on next page)

(continued from previous page)

```
class MyExtensionMapper(ObjectMapper)
    ...

register_map(MyExtensionContainer, MyExtensionMapper)
```

---

**Tip:** ObjectMappers allow you to customize how objects in the spec are mapped to attributes of your Container in Python. This is useful, e.g., in cases where you want to customize the default mapping. For an overview of the concepts of containers, spec, builders, object mappers in HDMF see also *Software Architecture*

---

## 7.4 Documenting Extensions

Coming soon!

## 7.5 Further Reading

- **Specification Language:** For a detailed overview of the specification language itself see <https://hdmf-schema-language.readthedocs.io/en/latest/index.html>

## BUILDING API CLASSES

After you have written an extension, you will need a Pythonic way to interact with the data model. To do this, you will need to write some classes that represent the data you defined in your specification extensions.

The `hdmf.container` module defines two base classes that represent the primitive structures supported by the schema. `Data` represents datasets and `Container` represents groups. See the classes in the `:py:mod:hdmf.common` package for examples.

### 8.1 The `register_class` function/decorator

When defining a class that represents a *data\_type* (i.e. anything that has a *data\_type\_def*) from your extension, you can tell HDMF which *data\_type* it represents using the function `register_class`. This class can be called on its own, or used as a class decorator. The first argument should be the *data\_type* and the second argument should be the *namespace* name.

The following example demonstrates how to register a class as the Python class representation of the *data\_type* “MyContainer” from the *namespace* “my\_ns”. The namespace must be loaded prior to the below code using the `load_namespaces` function.

```
from hdmf.common import register_class
from hdmf.container import Container

class MyContainer(Container):
    ...

register_class(data_type='MyContainer', namespace='my_ns', container_cls=MyContainer)
```

Alternatively, you can use `register_class` as a decorator.

```
from hdmf.common import register_class
from hdmf.container import Container

@type_map.register_class('MyContainer', 'my_ns')
class MyContainer(Container):
    ...
```

`register_class` is used with `Data` the same way it is used with `Container`.



## EXPORT

Export is a new feature in HDMF 2.0. You can use export to take a container that was read from a file and write it to a different file, with or without modifications to the container in memory. The in-memory container being exported will be written to the exported file as if it was never read from a file.

To export a container, first read the container from a file, then create a new `HDF5IO` object for exporting the data, then call `export` on the `HDF5IO` object, passing in the IO object used to read the container and optionally, the container itself, which may be modified in memory between reading and exporting.

For example:

```
with HDF5IO(self.read_path, manager=manager, mode='r') as read_io:
    with HDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io)
```

### 9.1 FAQ

#### 9.1.1 Can I read a container from disk, modify it, and then export the modified container?

Yes, you can export the in-memory container after modifying it in memory. The modifications will appear in the exported file and not the read file.

- If the modifications are removals or additions of containers, then no special action must be taken, as long as the container hierarchy is updated correspondingly.
- If the modifications are changes to attributes, then `Container.set_modified()` must be called on the container before exporting.

```
with HDF5IO(self.read_path, manager=manager, mode='r') as read_io:
    container = read_io.read()
    # ... # modify container
    container.set_modified() # this may be necessary if the modifications are changes_
    ↪ to attributes
    with HDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io, container=container)
```

---

**Note:** Modifications to `h5py.Dataset` objects act *directly* on the read file on disk. Changes are applied immediately and do not require exporting or writing the file. If you want to modify a dataset only in the new file, then you should

replace the whole object with a new array holding the modified data. To prevent unintentional changes to the source file, the source file should be opened with `mode='r'`.

---

### 9.1.2 Can I export a newly instantiated container?

No, you can only export containers that have been read from a file. The `src_io` argument is required in [HDMFIO.export](#).

### 9.1.3 Can I read a container from disk and export only part of the container?

It depends. You can only export the root container from a file. To export the root container without certain other sub-containers in the hierarchy, you can remove those other containers before exporting. However, you cannot export only a sub-container of the container hierarchy.

### 9.1.4 Can I write a newly instantiated container to two different files?

HDMF does not allow you to write a container that was not read from a file to two different files. For example, if you instantiate container A and write it file 1 and then try to write it to file 2, an error will be raised. However, you can read container A from file 1 and then export it to file 2, with or without modifications to container A in memory.

### 9.1.5 What happens to links when I export?

The exported file will not contain any links to the original file.

All links (such as internal links (i.e., HDF5 soft links) and links to other files (i.e., HDF5 external links)) will be preserved in the exported file.

If a link to an [h5py.Dataset](#) in another file is added to the in-memory container after reading it from file and then exported, then by default, the export process will create an external link to the existing [h5py.Dataset](#) object. To instead copy the data from the [h5py.Dataset](#) in another file to the exported file, pass the keyword argument `write_args={'link_data': False}` to [HDF5IO.export](#). This is similar to passing the keyword argument `link_data=False` to [HDF5IO.write](#) when writing a file with a copy of externally linked datasets.

### 9.1.6 What happens to references when I export?

References will be preserved in the exported file. NOTE: Exporting a file involves loading into memory all datasets that contain references and attributes that are references. The HDF5 reference IDs within an exported file may differ from the reference IDs in the original file.

### 9.1.7 What happens to object IDs when I export?

After exporting a container, the object IDs of the container and its child containers will be identical to the object IDs of the read container and its child containers. The object ID of a container uniquely identifies the container within a file, but should *not* be used to distinguish between two different files.

If you would like all object IDs to change on export, then first call the method [generate\\_new\\_id](#) on the root container to generate a new set of IDs for the root container and all of its children, recursively. Then export the container with its new IDs. Note: calling the [generate\\_new\\_id](#) method changes the object IDs of the containers in memory. These

changes are not reflected in the original file from which the containers were read unless the *HDF5IO.write* method is subsequently called.

```
with HDF5IO(self.read_path, manager=manager, mode='r') as read_io:
    container = read_io.read()
    container.generate_new_id()
    with HDF5IO(self.export_path, mode='w') as export_io:
        export_io.export(src_io=read_io, container=container)
```





## VALIDATING HDMF DATA

Validation of NWB files is available through `pynwb`. See the [PyNWB documentation](#) for more information.

---

**Note:** A simple interface for validating HDMF structured data through the command line like for PyNWB files is not yet implemented. If you would like this functionality to be available through `hdmf`, then please upvote [this issue](#).

---



## SUPPORT FOR THE HDMF SPECIFICATION LANGUAGE

The HDMF API provides nearly full support for all features of the [HDMF Specification Language](#) version 3.0.0, except for the following:

1. Attributes containing multiple references (see [#833](#))
2. Certain text and integer values for quantity (see [#423](#), [#531](#))
3. Datasets that do not have a `data_type_inc/data_type_def` and contain either a reference dtype or a compound dtype (see [#737](#))
4. Passing dataset dtype and shape from parent data type to child data type (see [#320](#))



## INSTALLING HDMF FOR DEVELOPERS

### 12.1 Set up a virtual environment

For development, we recommend installing HDMF in a virtual environment in editable mode. You can use the `venv` tool that comes packaged with Python to create a new virtual environment. Or you can use the `conda package and environment management system` for managing virtual environments.

#### 12.1.1 Option 1: Using `venv`

First, create a new virtual environment using the `venv` tool. This virtual environment will be stored in a new directory called "`hdmf-env`" in the current directory.

```
venv hdmf-env
```

On macOS or Linux, run the following to activate your new virtual environment:

```
source hdmf-env/bin/activate
```

On Windows, run the following to activate your new virtual environment:

```
hdmf-env\Scripts\activate
```

This virtual environment is a space where you can install Python packages that are isolated from other virtual environments. This is especially useful when working on multiple Python projects that have different package requirements and for testing Python code with different sets of installed packages or versions of Python.

Activate your newly created virtual environment using the above command whenever you want to work on HDMF. You can also deactivate it using the `deactivate` command to return to the base environment. And you can delete the virtual environment by deleting the directory that was created.

#### 12.1.2 Option 2: Using `conda`

The `conda package and environment management system` is an alternate way of managing virtual environments. First, install `Anaconda` to install the `conda` tool. Then create and activate a new virtual environment called "`hdmf-env`" with Python 3.12 installed.

```
conda create --name hdmf-env python=3.12
conda activate hdmf-env
```

Similar to a virtual environment created with `venv`, a `conda` environment is a space where you can install Python packages that are isolated from other virtual environments. In general, you should use `conda install` instead of `pip install` to install packages in a `conda` environment.

Activate your newly created virtual environment using the above command whenever you want to work on HDMF. You can also deactivate it using the `conda deactivate` command to return to the base environment. And you can delete the virtual environment by using the `conda remove --name hdmf-venv --all` command.

---

**Note:** For advanced users, we recommend using [Mambaforge](#), a faster version of the `conda` package manager that includes `conda-forge` as a default channel.

---

## 12.2 Install from GitHub

After you have created and activated a virtual environment, clone the HDMF git repository from GitHub, install the package requirements using the `pip` Python package manager, and install HDMF in editable mode.

```
git clone --recurse-submodules https://github.com/hdmf-dev/hdmf.git
cd hdmf
pip install -r requirements.txt -r requirements-dev.txt -r requirements-doc.txt -r
requirements-opt.txt
pip install -e .
```

---

**Note:** When using `conda`, you may use `pip install` to install dependencies as shown above; however, it is generally recommended that dependencies should be installed via `conda install`.

---

## 12.3 Run tests

You can run the full test suite by running:

```
pytest
```

This will run all the tests and compute the test coverage. The coverage report can be found in `/htmlcov`. You can also run a specific test module or class, or you can configure `pytest` to start the Python debugger (PDB) prompt on an error, e.g.,

```
pytest tests/unit/test_container.py # run all tests in
the module
pytest tests/unit/test_container.py::TestContainer # run all tests in
this class
pytest tests/unit/test_container.py::TestContainer::test_constructor # run this test
method
pytest --pdb tests/unit/test_container.py # start pdb on
error
```

You can run tests across multiple Python versions using the `tox` automated testing tool. Running `tox` will create a virtual environment, install dependencies, and run the test suite for different versions of Python. This can take some time to run.

```
tox
```

You can also test that the Sphinx Gallery files run without warnings or errors by running:

```
python test_gallery.py
```

## 12.4 Install latest pre-release

To try out the latest features or set up continuous integration of your own project against the latest version of HDMF, install the latest release from GitHub.

```
pip install -U hdmf --find-links https://github.com/hdmf-dev/hdmf/releases/tag/latest --  
↪no-index
```





## CONTRIBUTING GUIDE

### 13.1 Code of Conduct

This project and everyone participating in it is governed by our [code of conduct guidelines](#). By participating, you are expected to uphold this code. Please report unacceptable behavior.

### 13.2 Types of Contributions

#### 13.2.1 Did you find a bug? or Do you intend to add a new feature or change an existing one?

- **Submit issues and requests** using our [issue tracker](#)
- **Ensure the feature or change was not already reported** by searching on GitHub under [HDMF Issues](#)
- If you are unable to find an open issue addressing the problem then open a new issue on the respective repository. Be sure to use our issue templates and include:
  - **brief and descriptive title**
  - **clear description of the problem you are trying to solve.** Describing the use case is often more important than proposing a specific solution. By describing the use case and problem you are trying to solve gives the development team community a better understanding for the reasons of changes and enables others to suggest solutions.
  - **context** providing as much relevant information as possible and if available a **code sample** or an **executable test case** demonstrating the expected behavior and/or problem.
- Be sure to select the appropriate label (bug report or feature request) for your tickets so that they can be processed accordingly.
- HDMF is currently being developed primarily by staff at scientific research institutions and industry, most of which work on many different research projects. Please be patient, if our development team is not able to respond immediately to your issues. In particular issues that belong to later project milestones may not be reviewed or processed until work on that milestone begins.

### 13.2.2 Did you write a patch that fixes a bug or implements a new feature?

See the *Contributing Patches and Changes* section below for details.

### 13.2.3 Did you fix whitespace, format code, or make a purely cosmetic patch in source code?

Source code changes that are purely cosmetic in nature and do not add anything substantial to the stability, functionality, or testability will generally not be accepted unless they have been approved beforehand. One of the main reasons is that there are a lot of hidden costs in addition to writing the code itself, and with the limited resources of the project, we need to optimize developer time. E.g., someone needs to test and review PRs, backporting of bug fixes gets harder, it creates noise and pollutes the git repo and many other cost factors.

### 13.2.4 Do you have questions about HDMF?

See our [hdmf-dev.github.io](https://hdmf-dev.github.io) website for details.

### 13.2.5 Informal discussions between developers and users?

The <https://nwb-users.slack.com> slack is currently used for informal discussions between developers and users.

## 13.3 Contributing Patches and Changes

To contribute to HDMF you must submit your changes to the dev branch via a [Pull Request](#).

From your local copy directory, use the following commands.

- 1) First create a new branch to work on

```
$ git checkout -b <new_branch>
```

- 2) Make your changes.
- 3) Push your feature branch to origin (i.e. GitHub)

```
$ git push origin <new_branch>
```

- 4) Once you have tested and finalized your changes, create a pull request targeting dev as the base branch. Be sure to use our [pull request template](#) and:
  - Ensure the PR description clearly describes the problem and solution.
  - Include the relevant issue number if applicable.
  - Before submitting, please ensure that: \* The proposed changes include an addition to `CHANGELOG.md` describing your changes. To label the change with the PR number, you will have to first create the PR, then edit the `CHANGELOG.md` with the PR number, and push that change. \* The code follows our coding style. This can be checked running `ruff` from the source directory.
  - **NOTE:** Contributed branches will be removed by the development team after the merge is complete and should, hence, not be used after the pull request is complete.

## 13.4 Style Guides

### 13.4.1 Python Code Style Guide

Before you create a Pull Request, make sure you are following the HDMF style guide. To check whether your code conforms to the HDMF style guide, simply run the `ruff` tool in the project's root directory. `ruff` will also sort imports automatically and check against additional code style rules.

We also use `ruff` to sort python imports automatically and double-check that the codebase conforms to PEP8 standards, while using the `codespell` tool to check spelling.

`ruff` and `codespell` are installed when you follow the developer installation instructions. See *Installing HDMF for Developers*.

```
$ ruff check .  
$ codespell
```

### 13.4.2 Pre-Commit

We encourage developers to use `pre-commit` tool to automatically process the codebase to follow the style guide, as well as identify issues before making a commit. See installation and operation instructions in the `pre-commit` documentation.

### 13.4.3 Git Commit Message Styleguide

- Use the present tense (“Add feature” not “Added feature”)
- The first should be short and descriptive.
- Additional details may be included in further paragraphs.
- If a commit fixes an issue, then include “Fix #X” where X is the number of the issue.
- Reference relevant issues and pull requests liberally after the first line.

### 13.4.4 Documentation Styleguide

All documentations is written in reStructuredText (RST) using Sphinx.

## 13.5 Endorsement

Please do not take working with an organization (e.g., during a hackathon or via GitHub) as an endorsement of your work or your organization. It's okay to say e.g., “We worked with XXXXX to advance science” but not e.g., “XXXXX supports our work on HDMF”.

## 13.6 License and Copyright

See the [license](#) files for details about the copyright and license.

As indicated in the HDMF license: *“You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code (“Enhancements”) to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.”*

Contributors to the HDMF code base are expected to use a permissive, non-copyleft open source license. Typically 3-clause BSD is used, but any compatible license is allowed, the MIT and Apache 2.0 licenses being good alternative choices. The GPL and other copyleft licenses are not allowed due to the consternation it generates across many organizations.

Also, make sure that you are permitted to contribute code. Some organizations, even academic organizations, have agreements in place that discuss IP ownership in detail (i.e., address IP rights and ownership that you create while under the employ of the organization). These are typically signed documents that you looked at on your first day of work and then promptly forgot. We don’t want contributed code to be yanked later due to IP issues.

## HOW TO MAKE A ROUNDTrip TEST

The HDMF test suite has tools for easily doing round-trip tests of container classes. These tools exist in the [hdmf.testing](#) module. Round-trip tests exist for the container classes in the [hdmf.common](#) module. We recommend you write any additional round-trip tests in the `tests/unit/common` subdirectory of the Git repository.

For executing your new tests, we recommend using the `test.py` script in the top of the Git repository. Roundtrip tests will get executed as part of the full test suite, which can be executed with the following command:

```
$ python test.py
```

The roundtrip test will generate a new HDMF file with the name `test_<CLASS_NAME>.h5` where `CLASS_NAME` is the class name of the container class you are roundtripping. The test will write an HDMF file with an instance of the container to disk, read this instance back in, and compare it to the instance that was used for writing to disk. Once the test is complete, the HDMF file will be deleted. You can keep the HDMF file around after the test completes by setting the environment variable `CLEAN_HDMF` to `0`, `false`, `False`, or `FALSE`. Setting `CLEAN_HDMF` to any value not listed here will cause the roundtrip HDMF file to be deleted once the test has completed.

Before writing tests, we also suggest you familiarize yourself with the [software architecture](#) of HDMF.

### 14.1 H5RoundTripMixin

To write a roundtrip test, you will need to subclass the [H5RoundTripMixin](#) class and the [TestCase](#) class, in that order, and override some of the instance methods of the [H5RoundTripMixin](#) class to test the process of going from in-memory Python object to data stored on disk and back.

#### 14.1.1 setUpContainer

To configure the test for a particular container class, you need to override the [setUpContainer](#) method. This method should take no arguments, and return an instance of the container class you are testing.

Here is an example using a [CSRMatrix](#):

```
from hdmf.common import CSRMatrix
from hdmf.testing import TestCase, H5RoundTripMixin
import numpy as np

class TestCSRMatrixRoundTrip(H5RoundTripMixin, TestCase):

    def setUpContainer(self):
        data = np.array([1, 2, 3, 4, 5, 6])
        indices = np.array([0, 2, 2, 0, 1, 2])
```

(continues on next page)

(continued from previous page)

```
indptr = np.array([0, 2, 3, 6])  
return CSRMatrix(data, indices, indptr, (3, 3))
```

## SOFTWARE PROCESS

### 15.1 Continuous Integration

HDMF is tested against Ubuntu, macOS, and Windows operating systems. The project has both unit and integration tests. Tests run on [GitHub Actions](#).

Each time a PR is created or updated, the project is built, packaged, and tested on all supported operating systems and python distributions. That way, as a contributor, you know if you introduced regressions or coding style inconsistencies.

There are badges in the [README](#) file which shows the current condition of the dev branch.

### 15.2 Coverage

Code coverage is computed and reported using the [coverage](#) tool. There are two coverage-related badges in the [README](#) file. One shows the status of the [GitHub Action workflow](#) which runs the [coverage](#) tool and uploads the report to [codecov](#), and the other badge shows the percentage coverage reported from [codecov](#). A detailed report can be found on [codecov](#), which shows line by line which lines are covered by the tests.

### 15.3 Installation Requirements

[pyproject.toml](#) contains a list of package dependencies and their version ranges allowed for running HDMF. As a library, upper bound version constraints create more harm than good in the long term (see this [blog post](#)) so we avoid setting upper bounds on requirements.

If some of the packages are outdated, see [How to Update Requirements Files](#).

### 15.4 Testing Requirements

There are several kinds of requirements files used for testing PyNWB.

The first one is [requirements-min.txt](#), which lists the package dependencies and their minimum versions for installing HDMF.

The second one is [requirements.txt](#), which lists the pinned (concrete) dependencies to reproduce an entire development environment to use HDMF.

The third one is [requirements-dev.txt](#), which list the pinned (concrete) dependencies to reproduce an entire development environment to use HDMF, run HDMF tests, check code style, compute coverage, and create test environments.

The fourth one is `requirements-opt.txt`, which lists the pinned (concrete) optional dependencies to use all available features in HDMF.

The final one is `environment-ros3.yml`, which lists the dependencies used to test ROS3 streaming in HDMF.

## 15.5 Documentation Requirements

`requirements-doc.txt` lists the dependencies to generate the documentation for HDMF. Both this file and `requirements.txt` are used by `ReadTheDocs` to initialize the local environment for Sphinx to run.

## 15.6 Versioning and Releasing

HDMF uses `setuptools_scm` for versioning source and wheel distributions. `setuptools_scm` creates a semi-unique release name for the wheels that are created based on git tags. After all the tests pass, the “Deploy release” GitHub Actions workflow creates both a wheel (`\*.whl`) and source distribution (`\*.tar.gz`) for Python 3 and uploads them back to GitHub as a [release](#).

It is important to note that GitHub automatically generates source code archives in `.zip` and `.tar.gz` formats and attaches those files to all releases as an asset. These files currently do not contain the submodules within HDMF and thus do not serve as a complete installation. For a complete source code archive, use the source distribution generated by GitHub Actions, typically named `hdmf-{version}.tar.gz`.



## HOW TO MAKE A RELEASE

A core developer should use the following steps to create a release `X.Y.Z` of **hdmf**.

---

**Note:** Since the hdmf wheels do not include compiled code, they are considered *pure* and could be generated on any supported platform.

That said, considering the instructions below have been tested on a Linux system, they may have to be adapted to work on macOS or Windows.

---

### 16.1 Prerequisites

- All CI tests are passing on [GitHub Actions](#).
- You have a [GPG signing key](#).
- Dependency versions in `requirements.txt`, `requirements-dev.txt`, `requirements-opt.txt`, `requirements-doc.txt`, and `requirements-min.txt` are up-to-date.
- Legal information and copyright dates in `Legal.txt`, `license.txt`, `README.rst`, `docs/source/conf.py`, and any other files are up-to-date.
- Package information in `setup.py` is up-to-date.
- `README.rst` information is up-to-date.
- The `hdmf-common-schema` submodule is up-to-date. The version number should be checked manually in case syncing the git submodule does not work as expected.
- Documentation reflects any new features and changes in HDMF functionality.
- Documentation builds locally.
- Documentation builds on the [ReadTheDocs project](#) on the “dev” build.
- Release notes have been prepared.
- An appropriate new version number has been selected.

## 16.2 Documentation conventions

The commands reported below should be evaluated in the same terminal session.

Commands to evaluate starts with a dollar sign. For example:

```
$ echo "Hello"
Hello
```

means that `echo "Hello"` should be copied and evaluated in the terminal.

## 16.3 Publish release on PyPI: Step-by-step

1. Make sure that all CI tests are passing on [GitHub Actions](#).
2. List all tags sorted by version.

```
$ git tag -l | sort -V
```

3. Choose the next release version number and store it in a variable.

```
$ release=X.Y.Z
```

**Warning:** To ensure the packages are uploaded on PyPI, tags must match this regular expression:  
`^[0-9]+.[0-9]+.[0-9]+$`.

4. Download the latest sources.

```
$ cd /tmp && git clone --recurse-submodules git@github.com:hdmf-dev/hdmf && cd _
→ hdmf
```

5. Tag the release.

```
$ git tag --sign -m "hdmf ${release}" ${release} origin/dev
```

**Warning:** This step requires a [GPG signing key](#).

6. Publish the release tag.

```
$ git push origin ${release}
```

---

**Important:** This will trigger the “Deploy release” GitHub Actions workflow which will automatically upload the wheels and source distribution to both the [HDMF PyPI project page](#) and a new [GitHub release](#) using the hdmf-bot account.

---

7. Check the status of the builds on [GitHub Actions](#).
8. Once the builds are completed, check that the distributions are available on [HDMF PyPI project page](#) and that a new [GitHub release](#) was created.
9. Copy the release notes from `CHANGELOG.md` to the newly created [GitHub release](#).

10. Create a clean testing environment to test the installation.

On bash/zsh:

```
$ python -m venv hdmf-${release}-install-test && \
source hdmf-${release}-install-test/bin/activate
```

On other shells, see the [Python instructions for creating a virtual environment](#).

11. Test the installation:

```
$ pip install hdmf && \
python -c "import hdmf; print(hdmf.__version__)"
```

10. Cleanup

On bash/zsh:

```
$ deactivate && \
rm -rf dist/* && \
rm -rf hdmf-${release}-install-test
```

## 16.4 Publish release on conda-forge: Step-by-step

**Warning:** Publishing on conda requires you to have the corresponding package version uploaded on PyPI. So you have to do the PyPI and GitHub release before you do the conda release.

**Note:** Conda-forge maintains a bot called “regro-cf-autotick-bot” that regularly monitors PyPI for new releases of packages that are also on conda-forge. When a new release is detected, usually within 24 hours of publishing on PyPI, the bot will create a Pull Request with the correct modifications to the version and sha256 values in `meta.yaml`. If the requirements in `setup.py` have been changed, then you need to modify the requirements/run section in `meta.yaml` manually to reflect these changes. Once tests pass, merge the PR, and a new release will be published on Anaconda cloud. This is the easiest way to update the package version on conda-forge.

In order to release a new version on conda-forge manually, follow the steps below:

1. Store the release version string (this should match the PyPI version that you already published).

```
$ release=X.Y.Z
```

2. Fork the [hdmf-feedstock](#) repository to your GitHub user account.
3. Clone the forked feedstock to your local filesystem.

Fill the YOURGITHUBUSER part.

```
$ cd /tmp && git clone https://github.com/YOURGITHUBUSER/hdmf-feedstock.git
```

4. Download the corresponding source for the release version.

```
$ cd /tmp && \
  wget https://github.com/hdmf-dev/hdmf/releases/download/$release/hdmf-
  ↪$release.tar.gz
```

5. Create a new branch.

```
$ cd hdmf-feedstock && \
  git checkout -b $release
```

6. Modify `meta.yaml`.

Update the `version string` (line 2) and `sha256` (line 3).

We have to modify the sha and the version string in the `meta.yaml` file.

For linux flavors:

```
$ sed -i "2s/.*/{% set version = \"${release}\" %}/" recipe/meta.yaml
$ sha=$(openssl sha256 /tmp/hdmf-${release}.tar.gz | awk '{print $2}')
$ sed -i "3s/.*/{% set sha256 = \"${sha}\" %}/" recipe/meta.yaml
```

For macOS:

```
$ sed -i -- "2s/.*/{% set version = \"${release}\" %}/" recipe/meta.yaml
$ sha=$(openssl sha256 /tmp/hdmf-${release}.tar.gz | awk '{print $2}')
$ sed -i -- "3s/.*/{% set sha256 = \"${sha}\" %}/" recipe/meta.yaml
```

If the requirements in `setup.py` have been changed, then modify the requirements/run list in the `meta.yaml` file to reflect these changes.

7. Push the changes to your fork.

```
$ git push origin $release
```

8. Create a Pull Request.

Create a pull request against the [main feedstock repository](#). After the tests pass, merge the PR, and a new release will be published on Anaconda cloud.

## HOW TO UPDATE REQUIREMENTS FILES

The different requirements files introduced in *Software Process* section are the following:

- requirements.txt
- requirements-dev.txt
- requirements-doc.txt
- requirements-min.txt
- requirements-opt.txt

### 17.1 requirements.txt

*requirements.txt* of the project can be created or updated and then captured using the following script:

```
mkvirtualenv hdmf-requirements

cd hdmf
pip install .
pip check # check for package conflicts
pip freeze > requirements.txt

deactivate
rmvirtualenv hdmf-requirements
```

### 17.2 requirements-(dev|doc|opt).txt

Any of these requirements files can be updated using the following scripts:

```
cd hdmf

# Set the requirements file to update
target_requirements=requirements-dev.txt

mkvirtualenv hdmf-requirements

# Install updated requirements
pip install -U -r $target_requirements
```

(continues on next page)

(continued from previous page)

```
# If relevant, you could pip install new requirements now
# pip install -U <name-of-new-requirement>

# Check for any conflicts in installed packages
pip check

# Update list of pinned requirements
pip freeze > $target_requirements

deactivate
rmvirtualenv hdmf-requirements
```

## 17.3 requirements-min.txt

Minimum requirements should be updated manually if a new feature or bug fix is added in a dependency that is required for proper running of HDMF. Minimum requirements should also be updated if a user requests that HDMF be installable with an older version of a dependency, all tests pass using the older version, and there is no valid reason for the minimum version to be as high as it is.

## COPYRIGHT

“hdmf” Copyright (c) 2017-2024, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from the U.S. Dept. of Energy). All rights reserved.

If you have questions about your rights to use or distribute this software, please contact Berkeley Lab’s Innovation & Partnerships Office at [IPO@lbl.gov](mailto:IPO@lbl.gov).

NOTICE. This Software was developed under funding from the U.S. Department of Energy and the U.S. Government consequently retains certain rights. As such, the U.S. Government has been granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, distribute copies to the public, prepare derivative works, and perform publicly and display publicly, and to permit other to do so.





## LICENSE

“hdmf” Copyright (c) 2017-2024, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from the U.S. Dept. of Energy). All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- (3) Neither the name of the University of California, Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code (“Enhancements”) to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form.



## PYTHON MODULE INDEX

### h

- hdmf, 168
- hdmf.array, 163
- hdmf.backends, 144
- hdmf.backends.errors, 141
- hdmf.backends.hdf5, 141
- hdmf.backends.hdf5.h5\_utils, 129
- hdmf.backends.hdf5.h5tools, 135
- hdmf.backends.io, 141
- hdmf.backends.utils, 143
- hdmf.backends.warnings, 144
- hdmf.build, 109
- hdmf.build.builders, 93
- hdmf.build.classgenerator, 97
- hdmf.build.errors, 98
- hdmf.build.manager, 99
- hdmf.build.map, 105
- hdmf.build.objectmapper, 105
- hdmf.build.warnings, 108
- hdmf.common, 86
- hdmf.common.alignedtable, 63
- hdmf.common.hierarchicaltable, 68
- hdmf.common.io, 63
- hdmf.common.io.alignedtable, 61
- hdmf.common.io.multi, 61
- hdmf.common.io.resources, 62
- hdmf.common.io.table, 62
- hdmf.common.multi, 69
- hdmf.common.resources, 70
- hdmf.common.sparse, 77
- hdmf.common.table, 78
- hdmf.container, 88
- hdmf.data\_utils, 144
- hdmf.monitor, 164
- hdmf.query, 165
- hdmf.region, 166
- hdmf.spec, 129
- hdmf.spec.catalog, 109
- hdmf.spec.namespace, 111
- hdmf.spec.spec, 116
- hdmf.spec.write, 127
- hdmf.term\_set, 167
- hdmf.testing, 163
- hdmf.testing.testcase, 161
- hdmf.testing.utils, 163
- hdmf.testing.validate\_spec, 163
- hdmf.utils, 150
- hdmf.validate, 161
- hdmf.validate.errors, 156
- hdmf.validate.validator, 158



## Symbols

- `__getitem__` (*hdmf.region.RegionSlicer* property), 166
  - `__getitem__` () (*hdmf.array.Array* method), 163
  - `__getitem__` () (*hdmf.backends.hdf5.h5\_utils.AbstractH5ReferenceDataset* method), 131
  - `__getitem__` () (*hdmf.backends.hdf5.h5\_utils.AbstractH5RegionDataset* method), 131
  - `__getitem__` () (*hdmf.backends.hdf5.h5\_utils.AbstractH5TableDataset* method), 131
  - `__getitem__` () (*hdmf.backends.hdf5.h5\_utils.H5RegionSlicer* method), 133
  - `__getitem__` () (*hdmf.build.builders.GroupBuilder* method), 95
  - `__getitem__` () (*hdmf.common.alignedtable.AlignedDynamicTable* method), 66
  - `__getitem__` () (*hdmf.common.multi.SimpleMultiContainer* method), 69
  - `__getitem__` () (*hdmf.common.table.DynamicTable* method), 82
  - `__getitem__` () (*hdmf.common.table.DynamicTableRegion* method), 84
  - `__getitem__` () (*hdmf.common.table.EnumData* method), 85
  - `__getitem__` () (*hdmf.common.table.VectorIndex* method), 79
  - `__getitem__` () (*hdmf.container.Data* method), 91
  - `__getitem__` () (*hdmf.container.RowGetter* method), 92
  - `__getitem__` () (*hdmf.container.Table* method), 93
  - `__getitem__` () (*hdmf.data\_utils.DataIO* method), 150
  - `__getitem__` () (*hdmf.query.HDMFDataset* method), 165
  - `__getitem__` () (*hdmf.region.ListSlicer* method), 166
  - `__getitem__` () (*hdmf.term\_set.TermSet* method), 167
  - `__getitem__` () (*hdmf.term\_set.TermSetWrapper* method), 167
  - `__getitem__` () (*hdmf.utils.LabelledDict* method), 155
  - `__getitem__` () (*hdmf.utils.StrDataset* method), 155
  - `__iter__` () (*hdmf.data\_utils.AbstractDataChunkIterator* method), 144
  - `__len__` (*hdmf.region.RegionSlicer* property), 166
  - `__next__` () (*hdmf.data\_utils.AbstractDataChunkIterator* method), 144
  - `_get_data()` (*hdmf.data\_utils.GenericDataChunkIterator* method), 145
  - `_get_dtype()` (*hdmf.data\_utils.GenericDataChunkIterator* method), 146
  - `_get_maxshape()` (*hdmf.data\_utils.GenericDataChunkIterator* method), 146
- ## A
- AbstractContainer* (class in *hdmf.container*), 88
  - AbstractDataChunkIterator* (class in *hdmf.data\_utils*), 144
  - AbstractH5ReferenceDataset* (class in *hdmf.backends.hdf5.h5\_utils*), 131
  - AbstractH5RegionDataset* (class in *hdmf.backends.hdf5.h5\_utils*), 131
  - AbstractH5TableDataset* (class in *hdmf.backends.hdf5.h5\_utils*), 130
  - AbstractSortedArray* (class in *hdmf.array*), 163
  - `add()` (*hdmf.utils.LabelledDict* method), 155
  - `add()` (*hdmf.validate.validator.SpecMatches* method), 160
  - `add_attribute()` (*hdmf.spec.spec.BaseStorageSpec* method), 120
  - `add_candidate()` (*hdmf.build.manager.Proxy* method), 99
  - `add_category()` (*hdmf.common.alignedtable.AlignedDynamicTable* method), 64
  - `add_child()` (*hdmf.container.AbstractContainer* method), 89
  - `add_column()` (*hdmf.common.alignedtable.AlignedDynamicTable* method), 64
  - `add_column()` (*hdmf.common.table.DynamicTable* method), 81
  - `add_container()` (*hdmf.common.multi.SimpleMultiContainer* method), 70
  - `add_dataset()` (*hdmf.spec.spec.GroupSpec* method), 126
  - `add_group()` (*hdmf.spec.spec.GroupSpec* method), 125
  - `add_link()` (*hdmf.spec.spec.GroupSpec* method), 126
  - `add_namespace()` (*hdmf.spec.namespace.NamespaceCatalog* method), 114
  - `add_ref()` (*hdmf.common.resources.HERD* method), 75

[add\\_ref\\_container\(\)](#) ([hdmf.common.resources.HERD](#) method), 75  
[add\\_ref\\_termset\(\)](#) ([hdmf.common.resources.HERD](#) method), 75  
[add\\_row\(\)](#) ([hdmf.common.alignedtable.AlignedDynamicTable](#) method), 65  
[add\\_row\(\)](#) ([hdmf.common.resources.EntityKeyTable](#) method), 73  
[add\\_row\(\)](#) ([hdmf.common.resources.EntityTable](#) method), 71  
[add\\_row\(\)](#) ([hdmf.common.resources.FileTable](#) method), 71  
[add\\_row\(\)](#) ([hdmf.common.resources.KeyTable](#) method), 70  
[add\\_row\(\)](#) ([hdmf.common.resources.ObjectKeyTable](#) method), 73  
[add\\_row\(\)](#) ([hdmf.common.resources.ObjectTable](#) method), 72  
[add\\_row\(\)](#) ([hdmf.common.table.DynamicTable](#) method), 80  
[add\\_row\(\)](#) ([hdmf.common.table.EnumData](#) method), 85  
[add\\_row\(\)](#) ([hdmf.common.table.VectorData](#) method), 78  
[add\\_row\(\)](#) ([hdmf.common.table.VectorIndex](#) method), 79  
[add\\_row\(\)](#) ([hdmf.container.Table](#) method), 92  
[add\\_source\(\)](#) ([hdmf.spec.write.NamespaceBuilder](#) method), 128  
[add\\_spec\(\)](#) ([hdmf.spec.write.NamespaceBuilder](#) method), 128  
[add\\_spec\(\)](#) ([hdmf.spec.write.SpecFileBuilder](#) method), 129  
[add\\_vector\(\)](#) ([hdmf.common.table.VectorIndex](#) method), 79  
[AlignedDynamicTable](#) (class in [hdmf.common.alignedtable](#)), 63  
[AlignedDynamicTableMap](#) (class in [hdmf.common.io.alignedtable](#)), 61  
[all\\_children\(\)](#) ([hdmf.container.AbstractContainer](#) method), 89  
[all\\_objects](#) ([hdmf.container.AbstractContainer](#) property), 89  
[ALLOWED](#) ([hdmf.utils.AllowPositional](#) attribute), 150  
[AllowPositional](#) (class in [hdmf.utils](#)), 150  
[append\(\)](#) ([hdmf.backends.hdf5.h5\\_utils.HDF5IODataChunk](#) method), 129  
[append\(\)](#) ([hdmf.container.Data](#) method), 91  
[append\(\)](#) ([hdmf.data\\_utils.DataIO](#) method), 150  
[append\(\)](#) ([hdmf.term\\_set.TermSetWrapper](#) method), 168  
[append\\_data\(\)](#) (in module [hdmf.data\\_utils](#)), 144  
[apply\\_generator\\_to\\_field\(\)](#) ([hdmf.build.classgenerator.CustomClassGenerator](#) class method), 97  
[apply\\_generator\\_to\\_field\(\)](#) ([hdmf.build.classgenerator.MCIClassGenerator](#) class method), 98  
[apply\\_generator\\_to\\_field\(\)](#) ([hdmf.common.io.table.DynamicTableGenerator](#) class method), 63  
[Array](#) (class in [hdmf.array](#)), 163  
[assert\\_external\\_resources\\_equal\(\)](#) ([hdmf.common.resources.HERD](#) static method), 74  
[assertBuilderEqual\(\)](#) ([hdmf.testing.testcase.TestCase](#) method), 162  
[assertContainerEqual\(\)](#) ([hdmf.testing.testcase.TestCase](#) method), 161  
[assertEqualShape\(\)](#) (in module [hdmf.data\\_utils](#)), 148  
[assertRaisesWith\(\)](#) ([hdmf.testing.testcase.TestCase](#) method), 161  
[assertValidDtype\(\)](#) ([hdmf.spec.spec.DtypeSpec](#) static method), 121  
[assertWarnsWith\(\)](#) ([hdmf.testing.testcase.TestCase](#) method), 161  
[assign\\_to\\_specs\(\)](#) ([hdmf.validate.validator.SpecMatcher](#) method), 161  
[astype\(\)](#) ([hdmf.data\\_utils.DataChunk](#) method), 148  
[attr\\_columns\(\)](#) ([hdmf.common.io.table.DynamicTableMap](#) method), 62  
[attributes](#) ([hdmf.build.builders.BaseBuilder](#) property), 94  
[attributes](#) ([hdmf.spec.spec.BaseStorageSpec](#) property), 119  
[AttributeSpec](#) (class in [hdmf.spec.spec](#)), 117  
[AttributeValidator](#) (class in [hdmf.validate.validator](#)), 159  
[author](#) ([hdmf.spec.namespace.SpecNamespace](#) property), 112  
[auto\\_register\(\)](#) ([hdmf.spec.catalog.SpecCatalog](#) method), 110  
[available\\_namespaces\(\)](#) (in module [hdmf.common](#)), 86  

## B

[BaseBuilder](#) (class in [hdmf.build.builders](#)), 93  
[BaseStorageSpec](#) (class in [hdmf.spec.spec](#)), 118  
[BaseStorageValidator](#) (class in [hdmf.validate.validator](#)), 159  
[BrokenLinkWarning](#), 144  
[build\(\)](#) ([hdmf.build.manager.BuildManager](#) method), 100  
[build\(\)](#) ([hdmf.build.manager.TypeMap](#) method), 104  
[build\(\)](#) ([hdmf.build.objectmapper.ObjectMapper](#) method), 107  
[build\\_const\\_args\(\)](#) ([hdmf.spec.spec.AttributeSpec](#) class method), 118

build\_const\_args() (*hdmf.spec.spec.BaseStorageSpec class method*), 120  
 build\_const\_args() (*hdmf.spec.spec.ConstructableDict class method*), 116  
 build\_const\_args() (*hdmf.spec.spec.DatasetSpec class method*), 122  
 build\_const\_args() (*hdmf.spec.spec.DtypeSpec class method*), 121  
 build\_const\_args() (*hdmf.spec.spec.GroupSpec class method*), 127  
 build\_const\_args() (*hdmf.spec.spec.Spec class method*), 117  
 build\_namespace() (*hdmf.spec.namespace.SpecNamespace class method*), 113  
 build\_spec() (*hdmf.spec.spec.ConstructableDict class method*), 116  
 Builder (*class in hdmf.build.builders*), 93  
 builder (*hdmf.build.builders.LinkBuilder property*), 96  
 builder (*hdmf.build.builders.ReferenceBuilder property*), 96  
 BuilderH5ReferenceDataset (*class in hdmf.backends.hdf5.h5\_utils*), 132  
 BuilderH5RegionDataset (*class in hdmf.backends.hdf5.h5\_utils*), 132  
 BuilderH5TableDataset (*class in hdmf.backends.hdf5.h5\_utils*), 131  
 BuilderResolver (*class in hdmf.query*), 165  
 BuilderResolverMixin (*class in hdmf.backends.hdf5.h5\_utils*), 130  
 BuildError, 98  
 BuildManager (*class in hdmf.build.manager*), 99  
 BuildWarning, 108

## C

call\_docval\_func() (*in module hdmf.utils*), 151  
 can\_read() (*hdmf.backends.hdf5.h5tools.HDF5IO static method*), 135  
 can\_read() (*hdmf.backends.io.HDMFIO static method*), 141  
 catalog (*hdmf.spec.namespace.SpecNamespace property*), 112  
 categories (*hdmf.common.alignedtable.AlignedDynamicTable property*), 64  
 category\_tables (*hdmf.common.alignedtable.AlignedDynamicTable attribute*), 61  
 check\_dtype() (*hdmf.spec.spec.DtypeHelper static method*), 116  
 check\_shape() (*in module hdmf.validate.validator*), 158  
 check\_type() (*in module hdmf.utils*), 151  
 check\_type() (*in module hdmf.validate.validator*), 158  
 check\_valid\_dtype() (*hdmf.spec.spec.DtypeSpec static method*), 121  
 children (*hdmf.container.AbstractContainer property*), 89  
 chunks (*hdmf.build.builders.DatasetBuilder property*), 96  
 ClassGenerator (*class in hdmf.build.classgenerator*), 97  
 clear() (*hdmf.utils.LabelledDict method*), 155  
 clear\_cache() (*hdmf.build.manager.BuildManager method*), 100  
 close() (*hdmf.backends.hdf5.h5tools.HDF5IO method*), 138  
 close() (*hdmf.backends.io.HDMFIO method*), 143  
 close\_linked\_files() (*hdmf.backends.hdf5.h5tools.HDF5IO method*), 138  
 colnames (*hdmf.common.table.DynamicTable property*), 80  
 columns (*hdmf.common.table.DynamicTable property*), 80  
 columns (*hdmf.container.Table property*), 92  
 comm (*hdmf.backends.hdf5.h5tools.HDF5IO property*), 135  
 compute\_final\_result() (*hdmf.monitor.DataChunkProcessor method*), 164  
 compute\_final\_result() (*hdmf.monitor.NumSampleCounter method*), 165  
 construct() (*hdmf.build.manager.BuildManager method*), 100  
 construct() (*hdmf.build.manager.TypeMap method*), 104  
 construct() (*hdmf.build.objectmapper.ObjectMapper method*), 108  
 construct\_helper() (*hdmf.common.io.resources.HERDMap method*), 62  
 ConstructableDict (*class in hdmf.spec.spec*), 116  
 ConstructError, 99  
 constructor\_arg() (*hdmf.build.objectmapper.ObjectMapper static method*), 105  
 constructor\_args (*hdmf.build.objectmapper.ObjectMapper attribute*), 108  
 constructor\_args (*hdmf.common.io.alignedtable.AlignedDynamicTable attribute*), 61  
 constructor\_args (*hdmf.common.io.multi.SimpleMultiContainerMap attribute*), 61  
 constructor\_args (*hdmf.common.io.resources.HERDMap attribute*), 62  
 constructor\_args (*hdmf.common.io.table.DynamicTableMap attribute*), 63  
 contact (*hdmf.spec.namespace.SpecNamespace property*), 112  
 Container (*class in hdmf.container*), 89  
 container\_source (*hdmf.container.AbstractContainer*

property), 89  
 container\_types (*hdmf.build.manager.TypeMap* property), 102  
 ContainerConfigurationError, 99  
 ContainerH5ReferenceDataset (class in *hdmf.backends.hdf5.h5\_utils*), 132  
 ContainerH5RegionDataset (class in *hdmf.backends.hdf5.h5\_utils*), 132  
 ContainerH5TableDataset (class in *hdmf.backends.hdf5.h5\_utils*), 131  
 ContainerResolver (class in *hdmf.query*), 166  
 ContainerResolverMixin (class in *hdmf.backends.hdf5.h5\_utils*), 130  
 containers (*hdmf.common.multi.SimpleMultiContainer* property), 69  
 containers\_attr() (*hdmf.common.io.multi.SimpleMultiContainerMap* method), 61  
 containers\_carg() (*hdmf.common.io.multi.SimpleMultiContainerMap* method), 61  
 convert\_dt\_name() (*hdmf.build.objectmapper.ObjectMapper* class method), 106  
 convert\_dtype() (*hdmf.build.objectmapper.ObjectMapper* class method), 105  
 convert\_namespace() (*hdmf.backends.utils.NamespaceToBuilderHelper* class method), 143  
 copy() (*hdmf.common.table.DynamicTable* method), 83  
 copy\_file() (*hdmf.backends.hdf5.h5tools.HDF5IO* class method), 136  
 copy\_mappers() (*hdmf.build.manager.TypeMap* method), 102  
 create\_region() (*hdmf.common.table.DynamicTable* method), 81  
 CSRMatrix (class in *hdmf.common.sparse*), 77  
 css\_style (*hdmf.container.Container* property), 90  
 custom\_generators (*hdmf.build.classgenerator.ClassGenerator* property), 97  
 CustomClassGenerator (class in *hdmf.build.classgenerator*), 97

## D

Data (class in *hdmf.container*), 90  
 data (*hdmf.array.Array* property), 163  
 data (*hdmf.build.builders.DatasetBuilder* property), 96  
 data (*hdmf.container.Data* property), 90  
 data (*hdmf.container.DataRegion* property), 91  
 data (*hdmf.data\_utils.DataIO* property), 150  
 data (*hdmf.region.RegionSlicer* property), 166  
 data\_type (*hdmf.build.manager.Proxy* property), 99  
 data\_type (*hdmf.build.manager.TypeSource* property), 102  
 data\_type (*hdmf.common.alignedtable.AlignedDynamicTable* attribute), 68  
 data\_type (*hdmf.common.multi.SimpleMultiContainer* attribute), 70  
 data\_type (*hdmf.common.resources.HERD* attribute), 77  
 data\_type (*hdmf.common.sparse.CSRMatrix* attribute), 78  
 data\_type (*hdmf.common.table.DynamicTable* attribute), 83  
 data\_type (*hdmf.common.table.DynamicTableRegion* attribute), 84  
 data\_type (*hdmf.common.table.ElementIdentifiers* attribute), 80  
 data\_type (*hdmf.common.table.EnumData* attribute), 85  
 data\_type (*hdmf.common.table.VectorData* attribute), 85  
 data\_type (*hdmf.common.table.VectorIndex* attribute), 85  
 data\_type (*hdmf.container.AbstractContainer* property), 88  
 data\_type (*hdmf.container.Container* attribute), 90  
 data\_type (*hdmf.container.Data* attribute), 91  
 data\_type (*hdmf.spec.spec.BaseStorageSpec* property), 120  
 data\_type (*hdmf.validate.errors.MissingDataType* property), 157  
 data\_type\_def (*hdmf.spec.spec.BaseStorageSpec* property), 120  
 data\_type\_inc (*hdmf.spec.spec.BaseStorageSpec* property), 120  
 data\_type\_inc (*hdmf.spec.spec.LinkSpec* property), 122  
 DataChunk (class in *hdmf.data\_utils*), 148  
 DataChunkIterator (class in *hdmf.data\_utils*), 146  
 DataChunkProcessor (class in *hdmf.monitor*), 164  
 DataIO (class in *hdmf.data\_utils*), 149  
 DataRegion (class in *hdmf.container*), 91  
 datas\_attr() (*hdmf.common.io.multi.SimpleMultiContainerMap* method), 61  
 dataset (*hdmf.backends.hdf5.h5\_utils.H5DataIO* property), 134  
 dataset (*hdmf.query.HDMFDataset* property), 165  
 dataset\_spec\_cls (*hdmf.spec.namespace.NamespaceCatalog* property), 114  
 dataset\_spec\_cls() (*hdmf.spec.spec.GroupSpec* class method), 127  
 DatasetBuilder (class in *hdmf.build.builders*), 95  
 DatasetOfReferences (class in *hdmf.backends.hdf5.h5\_utils*), 130  
 datasets (*hdmf.build.builders.GroupBuilder* property), 94  
 datasets (*hdmf.spec.spec.GroupSpec* property), 125  
 DatasetSpec (class in *hdmf.spec.spec*), 121  
 DatasetValidator (class in *hdmf.validate.validator*),



160  
 date (*hdmf.spec.namespace.SpecNamespace* property), 112  
 def\_key() (*hdmf.spec.spec.BaseStorageSpec* class method), 120  
 default\_name (*hdmf.spec.spec.BaseStorageSpec* property), 118  
 default\_value (*hdmf.spec.spec.AttributeSpec* property), 118  
 default\_value (*hdmf.spec.spec.DatasetSpec* property), 122  
 description (*hdmf.common.table.DynamicTable* property), 80  
 description (*hdmf.common.table.VectorData* property), 78  
 dims (*hdmf.spec.spec.AttributeSpec* property), 118  
 dims (*hdmf.spec.spec.DatasetSpec* property), 122  
 doc (*hdmf.spec.namespace.SpecNamespace* property), 112  
 doc (*hdmf.spec.spec.DtypeSpec* property), 121  
 doc (*hdmf.spec.spec.Spec* property), 116  
 docval() (in module *hdmf.utils*), 151  
 docval\_macro() (in module *hdmf.utils*), 150  
 driver (*hdmf.backends.hdf5.h5tools.HDF5IO* property), 135  
 drop\_id\_columns() (in module *hdmf.common.hierarchicalltable*), 68  
 dtype (*hdmf.backends.hdf5.h5\_utils.AbstractH5ReferenceDataset* property), 131  
 dtype (*hdmf.backends.hdf5.h5\_utils.AbstractH5RegionDataset* property), 131  
 dtype (*hdmf.backends.hdf5.h5\_utils.AbstractH5TableDataset* property), 130  
 dtype (*hdmf.build.builders.DatasetBuilder* property), 96  
 dtype (*hdmf.data\_utils.AbstractDataChunkIterator* property), 145  
 dtype (*hdmf.data\_utils.DataChunk* property), 148  
 dtype (*hdmf.data\_utils.DataChunkIterator* property), 148  
 dtype (*hdmf.data\_utils.DataIO* property), 150  
 dtype (*hdmf.data\_utils.GenericDataChunkIterator* property), 146  
 dtype (*hdmf.query.HDMFDataset* property), 165  
 dtype (*hdmf.spec.spec.AttributeSpec* property), 117  
 dtype (*hdmf.spec.spec.DatasetSpec* property), 122  
 dtype (*hdmf.spec.spec.DtypeSpec* property), 121  
 dtype (*hdmf.term\_set.TermSetWrapper* property), 167  
 dtype\_spec\_cls() (*hdmf.spec.spec.DatasetSpec* class method), 122  
 DtypeConversionWarning, 109  
 DtypeError (class in *hdmf.validate.errors*), 156  
 DtypeHelper (class in *hdmf.spec.spec*), 116  
 DtypeSpec (class in *hdmf.spec.spec*), 120  
 DynamicTable (class in *hdmf.common.table*), 80  
 DynamicTableGenerator (class in *hdmf.common.io.table*), 63  
 DynamicTableMap (class in *hdmf.common.io.table*), 62  
 DynamicTableRegion (class in *hdmf.common.table*), 83  
**E**  
 ElementIdentifiers (class in *hdmf.common.table*), 79  
 elements (*hdmf.common.table.EnumData* property), 85  
 EmptyArrayError, 158  
 entities (*hdmf.common.resources.HERD* property), 74  
 entities() (*hdmf.common.io.resources.HERDMap* method), 62  
 Entity (class in *hdmf.common.resources*), 71  
 entity\_keys (*hdmf.common.resources.HERD* property), 74  
 entity\_keys() (*hdmf.common.io.resources.HERDMap* method), 62  
 EntityKey (class in *hdmf.common.resources*), 73  
 EntityKeyTable (class in *hdmf.common.resources*), 73  
 EntityTable (class in *hdmf.common.resources*), 70  
 EnumData (class in *hdmf.common.table*), 85  
 Error (class in *hdmf.validate.errors*), 156  
 ERROR (*hdmf.utils.AllowPositional* attribute), 150  
 evaluate() (*hdmf.query.Query* method), 165  
 exhaust\_queue() (*hdmf.backends.hdf5.h5\_utils.HDF5IODataChunkIterator* method), 129  
 ExpectedArrayError (class in *hdmf.validate.errors*), 156  
 export() (*hdmf.backends.hdf5.h5tools.HDF5IO* method), 137  
 export() (*hdmf.backends.io.HDMFIO* method), 142  
 export() (*hdmf.spec.write.NamespaceBuilder* method), 128  
 export\_io() (*hdmf.backends.hdf5.h5tools.HDF5IO* class method), 137  
 export\_spec() (in module *hdmf.spec.write*), 129  
 extend() (*hdmf.common.table.VectorData* method), 78  
 extend() (*hdmf.container.Data* method), 91  
 extend() (*hdmf.data\_utils.DataIO* method), 150  
 extend() (*hdmf.term\_set.TermSetWrapper* method), 168  
 extend\_data() (in module *hdmf.data\_utils*), 144  
 ExtenderMeta (class in *hdmf.utils*), 153  
**F**  
 field (*hdmf.term\_set.TermSetWrapper* property), 167  
 fields (*hdmf.container.AbstractContainer* property), 89  
 File (class in *hdmf.common.resources*), 71  
 files (*hdmf.common.resources.HERD* property), 74  
 files() (*hdmf.common.io.resources.HERDMap* method), 62  
 FileTable (class in *hdmf.common.resources*), 71  
 filter\_available() (*hdmf.backends.hdf5.h5\_utils.H5DataIO* static method), 134

`find_point()` (*hdmf.array.AbstractSortedArray method*), 163  
`find_point()` (*hdmf.array.LinSpace method*), 164  
`find_point()` (*hdmf.array.SortedArray method*), 163  
`flatten_column_index()` (*in module hdmf.common.hierarchicalltable*), 69  
`fmt_docval_args()` (*in module hdmf.utils*), 151  
`from_dataframe()` (*hdmf.common.table.DynamicTable class method*), 83  
`from_dataframe()` (*hdmf.container.Table class method*), 93  
`from_iterable()` (*hdmf.data\_utils.DataChunkIterator class method*), 147  
`from_zip()` (*hdmf.common.resources.HERD class method*), 77  
`full_name` (*hdmf.spec.namespace.SpecNamespace property*), 112

## G

`generate_class()` (*hdmf.build.classgenerator.ClassGenerator method*), 97  
`generate_html_repr()` (*hdmf.common.table.DynamicTable method*), 83  
`generate_new_id()` (*hdmf.container.AbstractContainer method*), 89  
`GenericDataChunkIterator` (*class in hdmf.data\_utils*), 145  
`get()` (*hdmf.build.builders.GroupBuilder method*), 95  
`get()` (*hdmf.common.alignedtable.AlignedDynamicTable method*), 66  
`get()` (*hdmf.common.table.DynamicTable method*), 82  
`get()` (*hdmf.common.table.DynamicTableRegion method*), 84  
`get()` (*hdmf.common.table.EnumData method*), 85  
`get()` (*hdmf.common.table.VectorData method*), 78  
`get()` (*hdmf.common.table.VectorIndex method*), 79  
`get()` (*hdmf.container.Data method*), 91  
`get_ancestor()` (*hdmf.container.AbstractContainer method*), 88  
`get_ancestors()` (*hdmf.container.AbstractContainer method*), 89  
`get_attr_names()` (*hdmf.build.objectmapper.ObjectMapper class method*), 106  
`get_attr_spec()` (*hdmf.build.objectmapper.ObjectMapper method*), 106  
`get_attr_value()` (*hdmf.build.objectmapper.ObjectMapper method*), 107  
`get_attr_value()` (*hdmf.common.io.table.DynamicTableMap method*), 62  
`get_attribute()` (*hdmf.build.objectmapper.ObjectMapper method*), 107  
`get_attribute()` (*hdmf.spec.spec.BaseStorageSpec method*), 120  
`get_builder()` (*hdmf.backends.hdf5.h5tools.HDF5IO method*), 138  
`get_builder()` (*hdmf.build.manager.BuildManager method*), 100  
`get_builder_dt()` (*hdmf.build.manager.BuildManager method*), 101  
`get_builder_dt()` (*hdmf.build.manager.TypeMap method*), 103  
`get_builder_loc()` (*hdmf.validate.validator.Validator class method*), 159  
`get_builder_name()` (*hdmf.build.manager.BuildManager method*), 101  
`get_builder_name()` (*hdmf.build.manager.TypeMap method*), 105  
`get_builder_name()` (*hdmf.build.objectmapper.ObjectMapper method*), 108  
`get_builder_ns()` (*hdmf.build.manager.BuildManager method*), 101  
`get_builder_ns()` (*hdmf.build.manager.TypeMap method*), 103  
`get_carg_spec()` (*hdmf.build.objectmapper.ObjectMapper method*), 106  
`get_category()` (*hdmf.common.alignedtable.AlignedDynamicTable method*), 64  
`get_class()` (*in module hdmf.common*), 86  
`get_cls()` (*hdmf.build.manager.BuildManager method*), 100  
`get_cls()` (*hdmf.build.manager.TypeMap method*), 103  
`get_colnames()` (*hdmf.common.alignedtable.AlignedDynamicTable method*), 65  
`get_config()` (*hdmf.term\_set.TypeConfigurator method*), 168  
`get_const_arg()` (*hdmf.build.objectmapper.ObjectMapper method*), 107  
`get_container()` (*hdmf.backends.hdf5.h5tools.HDF5IO method*), 138  
`get_container()` (*hdmf.common.multi.SimpleMultiContainer method*), 70  
`get_container_classes()` (*hdmf.build.manager.TypeMap method*), 104  
`get_container_cls()` (*hdmf.build.manager.TypeMap method*), 102  
`get_container_cls_dt()` (*hdmf.build.manager.TypeMap method*), 104  
`get_container_name()` (*hdmf.build.objectmapper.ObjectMapper method*), 106  
`get_container_ns_dt()` (*hdmf.build.manager.TypeMap method*), 104  
`get_data()` (*hdmf.array.AbstractSortedArray method*), 163

[get\\_data\(\)](#) (*hdmf.array.Array* method), 163  
[get\\_data\\_shape\(\)](#) (in module *hdmf.utils*), 153  
[get\\_data\\_type\(\)](#) (*hdmf.spec.spec.GroupSpec* method), 124  
[get\\_data\\_type\\_spec\(\)](#) (*hdmf.spec.spec.BaseStorageSpec* class method), 119  
[get\\_dataset\(\)](#) (*hdmf.spec.spec.GroupSpec* method), 126  
[get\\_docval\(\)](#) (in module *hdmf.utils*), 151  
[get\\_docval\\_macro\(\)](#) (in module *hdmf.utils*), 150  
[get\\_dt\\_container\\_cls\(\)](#) (*hdmf.build.manager.TypeMap* method), 103  
[get\\_entity\(\)](#) (*hdmf.common.resources.HERD* method), 76  
[get\\_fields\\_conf\(\)](#) (*hdmf.container.AbstractContainer* class method), 88  
[get\\_final\\_result\(\)](#) (*hdmf.monitor.DataChunkProcessor* method), 164  
[get\\_foreign\\_columns\(\)](#) (*hdmf.common.alignedtable.AlignedDynamicTable* method), 67  
[get\\_foreign\\_columns\(\)](#) (*hdmf.common.table.DynamicTable* method), 82  
[get\\_full\\_hierarchy\(\)](#) (*hdmf.spec.catalog.SpecCatalog* method), 110  
[get\\_group\(\)](#) (*hdmf.spec.spec.GroupSpec* method), 125  
[get\\_hdf5io\(\)](#) (in module *hdmf.common*), 87  
[get\\_hierarchy\(\)](#) (*hdmf.spec.catalog.SpecCatalog* method), 110  
[get\\_hierarchy\(\)](#) (*hdmf.spec.namespace.NamespaceCatalog* method), 114  
[get\\_hierarchy\(\)](#) (*hdmf.spec.namespace.SpecNamespace* method), 113  
[get\\_inverse\\_class\(\)](#) (*hdmf.backends.hdf5.h5\_utils.BuilderH5ReferenceDataset* class method), 132  
[get\\_inverse\\_class\(\)](#) (*hdmf.backends.hdf5.h5\_utils.BuilderH5RegionDataset* class method), 133  
[get\\_inverse\\_class\(\)](#) (*hdmf.backends.hdf5.h5\_utils.BuilderH5TableDataset* class method), 132  
[get\\_inverse\\_class\(\)](#) (*hdmf.backends.hdf5.h5\_utils.ContainerH5ReferenceDataset* class method), 132  
[get\\_inverse\\_class\(\)](#) (*hdmf.backends.hdf5.h5\_utils.ContainerH5RegionDataset* class method), 132  
[get\\_inverse\\_class\(\)](#) (*hdmf.backends.hdf5.h5\_utils.ContainerH5TableDataset* class method), 131  
[get\\_inverse\\_class\(\)](#) (*hdmf.query.ReferenceResolver* class method), 165  
[get\\_io\\_params\(\)](#) (*hdmf.backends.hdf5.h5\_utils.H5DataIO* method), 134  
[get\\_io\\_params\(\)](#) (*hdmf.data\_utils.DataIO* method), 150  
[get\\_iso8601\\_regex\(\)](#) (in module *hdmf.validate.validator*), 158  
[get\\_key\(\)](#) (*hdmf.common.resources.HERD* method), 76  
[get\\_link\(\)](#) (*hdmf.spec.spec.GroupSpec* method), 127  
[get\\_linked\\_resources\(\)](#) (*hdmf.container.HERDManager* method), 88  
[get\\_linked\\_tables\(\)](#) (*hdmf.common.alignedtable.AlignedDynamicTable* method), 67  
[get\\_linked\\_tables\(\)](#) (*hdmf.common.table.DynamicTable* method), 82  
[get\\_loaded\\_type\\_config\(\)](#) (in module *hdmf.common*), 86  
[get\\_manager\(\)](#) (in module *hdmf.common*), 87  
[get\\_map\(\)](#) (*hdmf.build.manager.TypeMap* method), 104  
[get\\_min\\_bounds\(\)](#) (*hdmf.data\_utils.DataChunk* method), 148  
[get\\_namespace\(\)](#) (*hdmf.spec.namespace.NamespaceCatalog* method), 114  
[get\\_namespace\\_sources\(\)](#) (*hdmf.spec.namespace.NamespaceCatalog* method), 115  
[get\\_namespace\\_spec\(\)](#) (*hdmf.spec.spec.BaseStorageSpec* class method), 119  
[get\\_namespaces\(\)](#) (*hdmf.backends.hdf5.h5tools.HDF5IO* class method), 136  
[get\\_object\(\)](#) (*hdmf.backends.hdf5.h5\_utils.BuilderResolverMixin* method), 130  
[get\\_object\(\)](#) (*hdmf.backends.hdf5.h5\_utils.ContainerResolverMixin* method), 130  
[get\\_object\(\)](#) (*hdmf.backends.hdf5.h5\_utils.DatasetOfReferences* method), 130  
[get\\_object\\_entities\(\)](#) (*hdmf.common.resources.HERD* method), 76  
[get\\_object\\_type\(\)](#) (*hdmf.common.resources.HERD* method), 76  
[get\\_proxy\(\)](#) (*hdmf.build.manager.BuildManager* method), 100  
[get\\_read\\_io\(\)](#) (*hdmf.container.AbstractContainer* method), 88  
[get\\_region\\_slicer\(\)](#) (in module *hdmf*), 168  
[get\\_registered\\_types\(\)](#) (*hdmf.spec.catalog.SpecCatalog* method),

[110](#)  
[get\\_registered\\_types\(\)](#) (*hdmf.spec.namespace.SpecNamespace* method), [113](#)  
[get\\_source\\_description\(\)](#) (*hdmf.spec.namespace.SpecNamespace* method), [112](#)  
[get\\_source\\_files\(\)](#) (*hdmf.spec.namespace.SpecNamespace* method), [112](#)  
[get\\_source\\_name\(\)](#) (*hdmf.backends.utils.NamespaceToBuilderHelper* class method), [143](#)  
[get\\_sources\(\)](#) (*hdmf.spec.namespace.NamespaceCatalog* method), [115](#)  
[get\\_spec\(\)](#) (*hdmf.spec.catalog.SpecCatalog* method), [110](#)  
[get\\_spec\(\)](#) (*hdmf.spec.namespace.NamespaceCatalog* method), [114](#)  
[get\\_spec\(\)](#) (*hdmf.spec.namespace.SpecNamespace* method), [112](#)  
[get\\_spec\\_loc\(\)](#) (*hdmf.validate.validator.Validator* class method), [159](#)  
[get\\_spec\\_source\\_file\(\)](#) (*hdmf.spec.catalog.SpecCatalog* method), [110](#)  
[get\\_string\\_format\(\)](#) (*hdmf.validate.validator* module), [158](#)  
[get\\_subspec\(\)](#) (*hdmf.build.manager.BuildManager* method), [101](#)  
[get\\_subspec\(\)](#) (*hdmf.build.manager.TypeMap* method), [103](#)  
[get\\_subtypes\(\)](#) (*hdmf.spec.catalog.SpecCatalog* method), [111](#)  
[get\\_target\\_type\(\)](#) (*hdmf.spec.spec.GroupSpec* method), [124](#)  
[get\\_type\(\)](#) (*hdmf.backends.hdf5.h5tools.HDF5IO* class method), [139](#)  
[get\\_type\(\)](#) (*hdmf.validate.validator* module), [158](#)  
[get\\_type\\_map\(\)](#) (*hdmf.common* module), [87](#)  
[get\\_types\(\)](#) (*hdmf.spec.namespace.NamespaceCatalog* method), [115](#)  
[get\\_validator\(\)](#) (*hdmf.validate.validator.ValidatorMap* method), [158](#)  
[get\\_written\(\)](#) (*hdmf.backends.hdf5.h5tools.HDF5IO* method), [138](#)  
[get\\_written\(\)](#) (*hdmf.backends.utils.WriteStatusTracker* method), [143](#)  
[get\\_zip\\_directory\(\)](#) (*hdmf.common.resources.HERD* class method), [77](#)  
[getargs\(\)](#) (*hdmf.utils* module), [152](#)  
[group\\_spec\\_cls](#) (*hdmf.spec.namespace.NamespaceCatalog* property), [114](#)  
[GroupBuilder](#) (class in *hdmf.build.builders*), [94](#)  
[groups](#) (*hdmf.build.builders.GroupBuilder* property), [94](#)  
[groups](#) (*hdmf.spec.spec.GroupSpec* property), [125](#)  
[GroupSpec](#) (class in *hdmf.spec.spec*), [122](#)  
[GroupValidator](#) (class in *hdmf.validate.validator*), [160](#)

## H

[H5DataIO](#) (class in *hdmf.backends.hdf5.h5\_utils*), [133](#)  
[H5Dataset](#) (class in *hdmf.backends.hdf5.h5\_utils*), [129](#)  
[H5RegionSlicer](#) (class in *hdmf.backends.hdf5.h5\_utils*), [133](#)  
[H5RoundtripMixin](#) (class in *hdmf.testing.testcase*), [162](#)  
[H5SpecReader](#) (class in *hdmf.backends.hdf5.h5\_utils*), [133](#)  
[H5SpecWriter](#) (class in *hdmf.backends.hdf5.h5\_utils*), [133](#)  
[has\\_foreign\\_columns\(\)](#) (*hdmf.common.alignedtable.AlignedDynamicTable* method), [67](#)  
[has\\_foreign\\_columns\(\)](#) (*hdmf.common.table.DynamicTable* method), [82](#)  
[HDF5IO](#) (class in *hdmf.backends.hdf5.h5tools*), [135](#)  
[HDF5IODataChunkIteratorQueue](#) (class in *hdmf.backends.hdf5.h5\_utils*), [129](#)  
[hdmf](#) module, [168](#)  
[hdmf.array](#) module, [163](#)  
[hdmf.backends](#) module, [144](#)  
[hdmf.backends.errors](#) module, [141](#)  
[hdmf.backends.hdf5](#) module, [141](#)  
[hdmf.backends.hdf5.h5\\_utils](#) module, [129](#)  
[hdmf.backends.hdf5.h5tools](#) module, [135](#)  
[hdmf.backends.io](#) module, [141](#)  
[hdmf.backends.utils](#) module, [143](#)  
[hdmf.backends.warnings](#) module, [144](#)  
[hdmf.build](#) module, [109](#)  
[hdmf.build.builders](#) module, [93](#)  
[hdmf.build.classgenerator](#) module, [97](#)  
[hdmf.build.errors](#) module, [98](#)  
[hdmf.build.manager](#) module, [99](#)  
[hdmf.build.map](#)



- module, 105
- hdmf.build.objectmapper
  - module, 105
- hdmf.build.warnings
  - module, 108
- hdmf.common
  - module, 86
- hdmf.common.alignedtable
  - module, 63
- hdmf.common.hierarchicaltable
  - module, 68
- hdmf.common.io
  - module, 63
- hdmf.common.io.alignedtable
  - module, 61
- hdmf.common.io.multi
  - module, 61
- hdmf.common.io.resources
  - module, 62
- hdmf.common.io.table
  - module, 62
- hdmf.common.multi
  - module, 69
- hdmf.common.resources
  - module, 70
- hdmf.common.sparse
  - module, 77
- hdmf.common.table
  - module, 78
- hdmf.container
  - module, 88
- hdmf.data\_utils
  - module, 144
- hdmf.monitor
  - module, 164
- hdmf.query
  - module, 165
- hdmf.region
  - module, 166
- hdmf.spec
  - module, 129
- hdmf.spec.catalog
  - module, 109
- hdmf.spec.namespace
  - module, 111
- hdmf.spec.spec
  - module, 116
- hdmf.spec.write
  - module, 127
- hdmf.term\_set
  - module, 167
- hdmf.testing
  - module, 163
- hdmf.testing.testcase

- module, 161
- hdmf.testing.utils
  - module, 163
- hdmf.testing.validate\_spec
  - module, 163
- hdmf.utils
  - module, 150
- hdmf.validate
  - module, 161
- hdmf.validate.errors
  - module, 156
- hdmf.validate.validator
  - module, 158
- HDMFDataset (*class in hdmf.query*), 165
- HDMFIO (*class in hdmf.backends.io*), 141
- HERD (*class in hdmf.common.resources*), 74
- HERDManager (*class in hdmf.container*), 88
- HERDMap (*class in hdmf.common.io.resources*), 62
- |
- id (*hdmf.common.table.DynamicTable property*), 80
- id\_key() (*hdmf.spec.spec.BaseStorageSpec class method*), 119
- idx (*hdmf.container.Row property*), 92
- IllegalLinkError (*class in hdmf.validate.errors*), 157
- inc\_key() (*hdmf.spec.spec.BaseStorageSpec class method*), 119
- include\_namespace()
  - (*hdmf.spec.write.NamespaceBuilder method*), 128
- include\_type() (*hdmf.spec.write.NamespaceBuilder method*), 128
- IncorrectDataType (*class in hdmf.validate.errors*), 157
- IncorrectQuantityBuildWarning, 108
- IncorrectQuantityError (*class in hdmf.validate.errors*), 157
- InvalidDataIOError, 150
- invert() (*hdmf.backends.hdf5.h5\_utils.DatasetOfReferences method*), 130
- invert() (*hdmf.query.ReferenceResolver method*), 165
- io (*hdmf.backends.hdf5.h5\_utils.H5Dataset property*), 130
- io\_settings (*hdmf.backends.hdf5.h5\_utils.H5DataIO property*), 134
- is\_empty() (*hdmf.build.builders.GroupBuilder method*), 95
- is\_inherited\_attribute()
  - (*hdmf.spec.spec.BaseStorageSpec method*), 119
- is\_inherited\_dataset() (*hdmf.spec.spec.GroupSpec method*), 123
- is\_inherited\_group() (*hdmf.spec.spec.GroupSpec method*), 123

- is\_inherited\_link()** (*hdmf.spec.spec.GroupSpec method*), 123  
**is\_inherited\_spec()** (*hdmf.spec.spec.BaseStorageSpec method*), 118  
**is\_inherited\_spec()** (*hdmf.spec.spec.GroupSpec method*), 124  
**is\_inherited\_target\_type()** (*hdmf.spec.spec.GroupSpec method*), 124  
**is\_inherited\_type()** (*hdmf.spec.spec.GroupSpec method*), 124  
**is\_many()** (*hdmf.spec.spec.BaseStorageSpec method*), 119  
**is\_many()** (*hdmf.spec.spec.LinkSpec method*), 122  
**is\_overridden\_attribute()** (*hdmf.spec.spec.BaseStorageSpec method*), 119  
**is\_overridden\_dataset()** (*hdmf.spec.spec.GroupSpec method*), 123  
**is\_overridden\_group()** (*hdmf.spec.spec.GroupSpec method*), 123  
**is\_overridden\_link()** (*hdmf.spec.spec.GroupSpec method*), 123  
**is\_overridden\_spec()** (*hdmf.spec.spec.BaseStorageSpec method*), 119  
**is\_overridden\_spec()** (*hdmf.spec.spec.GroupSpec method*), 124  
**is\_overridden\_target\_type()** (*hdmf.spec.spec.GroupSpec method*), 124  
**is\_overridden\_type()** (*hdmf.spec.spec.GroupSpec method*), 124  
**is\_ragged()** (*in module hdmf.utils*), 154  
**is\_ref()** (*hdmf.spec.spec.DtypeSpec static method*), 121  
**is\_region()** (*hdmf.spec.spec.RefSpec method*), 117  
**is\_sub\_data\_type()** (*hdmf.build.manager.BuildManager method*), 101  
**is\_sub\_data\_type()** (*hdmf.spec.namespace.NamespaceCatalog method*), 115  
**items()** (*hdmf.build.builders.GroupBuilder method*), 95
- J**
- js\_script** (*hdmf.container.Container property*), 90
- K**
- Key** (*class in hdmf.common.resources*), 70  
**key\_attr** (*hdmf.utils.LabelledDict property*), 155  
**keys** (*hdmf.common.resources.HERD property*), 74  
**keys()** (*hdmf.build.builders.GroupBuilder method*), 95  
**keys()** (*hdmf.common.io.resources.HERDMap method*), 62  
**KeyTable** (*class in hdmf.common.resources*), 70
- L**
- label** (*hdmf.utils.LabelledDict property*), 154  
**LabelledDict** (*class in hdmf.utils*), 154  
**link\_data** (*hdmf.backends.hdf5.h5\_utils.H5DataIO property*), 134  
**link\_resources()** (*hdmf.container.HERDManager method*), 88  
**link\_spec\_cls()** (*hdmf.spec.spec.GroupSpec class method*), 127  
**linkable** (*hdmf.spec.spec.BaseStorageSpec property*), 119  
**LinkBuilder** (*class in hdmf.build.builders*), 96  
**links** (*hdmf.build.builders.GroupBuilder property*), 94  
**links** (*hdmf.spec.spec.GroupSpec property*), 125  
**LinkSpec** (*class in hdmf.spec.spec*), 122  
**LinSpace** (*class in hdmf.array*), 164  
**ListSlicer** (*class in hdmf.region*), 166  
**load\_namespaces()** (*hdmf.backends.hdf5.h5tools.HDF5IO class method*), 135  
**load\_namespaces()** (*hdmf.build.manager.TypeMap method*), 102  
**load\_namespaces()** (*hdmf.spec.namespace.NamespaceCatalog method*), 115  
**load\_namespaces()** (*in module hdmf.common*), 86  
**load\_type\_config()** (*hdmf.term\_set.TypeConfigurator method*), 168  
**load\_type\_config()** (*in module hdmf.common*), 86  
**location** (*hdmf.build.builders.BaseBuilder property*), 94  
**location** (*hdmf.build.manager.Proxy property*), 99  
**location** (*hdmf.validate.errors.Error property*), 156
- M**
- main()** (*in module hdmf.testing.validate\_spec*), 163  
**manager** (*hdmf.backends.io.HDMFIO property*), 141  
**map\_attr()** (*hdmf.build.objectmapper.ObjectMapper method*), 106  
**map\_const\_arg()** (*hdmf.build.objectmapper.ObjectMapper method*), 107  
**map\_spec()** (*hdmf.build.objectmapper.ObjectMapper method*), 107  
**matches()** (*hdmf.build.manager.Proxy method*), 99  
**maxshape** (*hdmf.build.builders.DatasetBuilder property*), 96  
**maxshape** (*hdmf.data\_utils.AbstractDataChunkIterator property*), 145  
**maxshape** (*hdmf.data\_utils.DataChunkIterator property*), 148  
**maxshape** (*hdmf.data\_utils.GenericDataChunkIterator property*), 146  
**MCIClassGenerator** (*class in hdmf.build.classgenerator*), 98  
**merge()** (*hdmf.build.manager.TypeMap method*), 102

merge() (*hdmf.spec.namespace.NamespaceCatalog method*), 114

MissingDataType (*class in hdmf.validate.errors*), 157

MissingError (*class in hdmf.validate.errors*), 156

MissingRequiredBuildWarning, 109

MissingRequiredWarning, 109

mode (*hdmf.backends.hdf5.h5tools.HDF5IO property*), 140

modified (*hdmf.container.AbstractContainer property*), 89

module

- hdmf, 168
- hdmf.array, 163
- hdmf.backends, 144
- hdmf.backends.errors, 141
- hdmf.backends.hdf5, 141
- hdmf.backends.hdf5.h5\_utils, 129
- hdmf.backends.hdf5.h5tools, 135
- hdmf.backends.io, 141
- hdmf.backends.utils, 143
- hdmf.backends.warnings, 144
- hdmf.build, 109
- hdmf.build.builders, 93
- hdmf.build.classgenerator, 97
- hdmf.build.errors, 98
- hdmf.build.manager, 99
- hdmf.build.map, 105
- hdmf.build.objectmapper, 105
- hdmf.build.warnings, 108
- hdmf.common, 86
- hdmf.common.alignedtable, 63
- hdmf.common.hierarchicaltable, 68
- hdmf.common.io, 63
- hdmf.common.io.alignedtable, 61
- hdmf.common.io.multi, 61
- hdmf.common.io.resources, 62
- hdmf.common.io.table, 62
- hdmf.common.multi, 69
- hdmf.common.resources, 70
- hdmf.common.sparse, 77
- hdmf.common.table, 78
- hdmf.container, 88
- hdmf.data\_utils, 144
- hdmf.monitor, 164
- hdmf.query, 165
- hdmf.region, 166
- hdmf.spec, 129
- hdmf.spec.catalog, 109
- hdmf.spec.namespace, 111
- hdmf.spec.spec, 116
- hdmf.spec.write, 127
- hdmf.term\_set, 167
- hdmf.testing, 163
- hdmf.testing.testcase, 161

- hdmf.testing.utils, 163
- hdmf.testing.validate\_spec, 163
- hdmf.utils, 150
- hdmf.validate, 161
- hdmf.validate.errors, 156
- hdmf.validate.validator, 158
- MultiContainerInterface (*class in hdmf.container*), 91

## N

- name (*hdmf.build.builders.Builder property*), 93
- name (*hdmf.container.AbstractContainer property*), 88
- name (*hdmf.spec.namespace.SpecNamespace property*), 112
- name (*hdmf.spec.spec.DtypeSpec property*), 121
- name (*hdmf.spec.spec.Spec property*), 116
- name (*hdmf.spec.write.NamespaceBuilder property*), 129
- name (*hdmf.validate.errors.Error property*), 156
- namespace (*hdmf.build.manager.Proxy property*), 99
- namespace (*hdmf.build.manager.TypeSource property*), 102
- namespace (*hdmf.common.alignedtable.AlignedDynamicTable attribute*), 68
- namespace (*hdmf.common.multi.SimpleMultiContainer attribute*), 70
- namespace (*hdmf.common.resources.HERD attribute*), 77
- namespace (*hdmf.common.sparse.CSRMatrix attribute*), 78
- namespace (*hdmf.common.table.DynamicTable attribute*), 83
- namespace (*hdmf.common.table.DynamicTableRegion attribute*), 85
- namespace (*hdmf.common.table.ElementIdentifiers attribute*), 80
- namespace (*hdmf.common.table.EnumData attribute*), 85
- namespace (*hdmf.common.table.VectorData attribute*), 78
- namespace (*hdmf.common.table.VectorIndex attribute*), 79
- namespace (*hdmf.container.Container attribute*), 90
- namespace (*hdmf.container.Data attribute*), 91
- namespace (*hdmf.validate.validator.ValidatorMap property*), 158
- namespace\_catalog (*hdmf.build.manager.BuildManager property*), 100
- namespace\_catalog (*hdmf.build.manager.TypeMap property*), 102
- NamespaceBuilder (*class in hdmf.spec.write*), 127
- NamespaceCatalog (*class in hdmf.spec.namespace*), 113
- namespaces (*hdmf.spec.namespace.NamespaceCatalog property*), 114

- NamespaceToBuilderHelper (class in *hdmf.backends.utils*), 143
- next() (*hdmf.data\_utils.DataChunkIterator* method), 147
- next() (*hdmf.query.HDMFDataset* method), 165
- no\_convert() (*hdmf.build.objectmapper.ObjectMapper* class method), 105
- NotYetExhausted, 164
- NumSampleCounter (class in *hdmf.monitor*), 164
- ## O
- obj\_attrs (*hdmf.build.objectmapper.ObjectMapper* attribute), 108
- obj\_attrs (*hdmf.common.io.alignedtable.AlignedDynamicTableMap* attribute), 61
- obj\_attrs (*hdmf.common.io.multi.SimpleMultiContainerMap* attribute), 62
- obj\_attrs (*hdmf.common.io.resources.HERDMap* attribute), 62
- obj\_attrs (*hdmf.common.io.table.DynamicTableMap* attribute), 63
- Object (class in *hdmf.common.resources*), 72
- object\_attr() (*hdmf.build.objectmapper.ObjectMapper* static method), 106
- object\_id (*hdmf.container.AbstractContainer* property), 89
- object\_keys (*hdmf.common.resources.HERD* property), 74
- object\_keys() (*hdmf.common.io.resources.HERDMap* method), 62
- OBJECT\_REF\_TYPE (*hdmf.build.builders.DatasetBuilder* attribute), 96
- ObjectKey (class in *hdmf.common.resources*), 73
- ObjectKeyTable (class in *hdmf.common.resources*), 72
- ObjectMapper (class in *hdmf.build.objectmapper*), 105
- objects (*hdmf.common.resources.HERD* property), 74
- objects() (*hdmf.common.io.resources.HERDMap* method), 62
- ObjectTable (class in *hdmf.common.resources*), 72
- open() (*hdmf.backends.hdf5.h5tools.HDF5IO* method), 138
- open() (*hdmf.backends.io.HDMFIO* method), 142
- OrphanContainerBuildError, 98
- OrphanContainerWarning, 109
- ## P
- parent (*hdmf.build.builders.Builder* property), 93
- parent (*hdmf.container.AbstractContainer* property), 89
- parent (*hdmf.spec.spec.Spec* property), 117
- path (*hdmf.build.builders.Builder* property), 93
- path (*hdmf.spec.spec.Spec* property), 117
- pop() (*hdmf.utils.LabelledDict* method), 155
- popargs() (in module *hdmf.utils*), 152
- popargs\_to\_dict() (in module *hdmf.utils*), 152
- popitem() (*hdmf.utils.LabelledDict* method), 155
- post\_init() (*hdmf.utils.ExtenderMeta* class method), 153
- post\_process() (*hdmf.build.classgenerator.CustomClassGenerator* class method), 98
- post\_process() (*hdmf.build.classgenerator.MCIClassGenerator* class method), 98
- post\_process() (*hdmf.common.io.table.DynamicTableGenerator* class method), 63
- pre\_init() (*hdmf.utils.ExtenderMeta* class method), 153
- prebuilt() (*hdmf.build.manager.BuildManager* method), 100
- primary\_dtype\_synonyms (*hdmf.spec.spec.DtypeHelper* attribute), 116
- process\_data\_chunk() (*hdmf.monitor.DataChunkProcessor* method), 164
- process\_data\_chunk() (*hdmf.monitor.NumSampleCounter* method), 165
- process\_field\_spec() (*hdmf.build.classgenerator.CustomClassGenerator* class method), 97
- process\_field\_spec() (*hdmf.build.classgenerator.MCIClassGenerator* class method), 98
- process\_field\_spec() (*hdmf.common.io.table.DynamicTableGenerator* class method), 63
- Proxy (class in *hdmf.build.manager*), 99
- purge\_outdated() (*hdmf.build.manager.BuildManager* method), 100
- pystr() (in module *hdmf.utils*), 153
- ## Q
- quantity (*hdmf.spec.spec.BaseStorageSpec* property), 120
- quantity (*hdmf.spec.spec.LinkSpec* property), 122
- Query (class in *hdmf.query*), 165
- queue\_ref() (*hdmf.build.manager.BuildManager* method), 100
- ## R
- read() (*hdmf.backends.hdf5.h5tools.HDF5IO* method), 137
- read() (*hdmf.backends.io.HDMFIO* method), 141
- read\_builder() (*hdmf.backends.hdf5.h5tools.HDF5IO* method), 138
- read\_builder() (*hdmf.backends.io.HDMFIO* method), 142
- read\_io (*hdmf.container.AbstractContainer* property), 88



`read_namespace()` (*hdmf.backends.hdf5.h5\_utils.H5SpecReader* property), 130  
     *method*), 133  
`read_namespace()` (*hdmf.spec.namespace.SpecReader* property), 166  
     *method*), 113  
`read_namespace()` (*hdmf.spec.namespace.YAMLSpecReader* property), 166  
     *method*), 113  
`read_spec()` (*hdmf.backends.hdf5.h5\_utils.H5SpecReader* property), 166  
     *method*), 133  
`read_spec()` (*hdmf.spec.namespace.SpecReader* property), 166  
     *method*), 113  
`read_spec()` (*hdmf.spec.namespace.YAMLSpecReader* property), 166  
     *method*), 113  
`reason` (*hdmf.validate.errors.Error* property), 156  
`recommended_chunk_shape()` (*hdmf.data\_utils.AbstractDataChunkIterator* property), 144  
     *method*), 144  
`recommended_chunk_shape()` (*hdmf.data\_utils.DataChunkIterator* property), 147  
     *method*), 147  
`recommended_chunk_shape()` (*hdmf.data\_utils.GenericDataChunkIterator* property), 146  
     *method*), 146  
`recommended_chunk_shape()` (*hdmf.monitor.DataChunkProcessor* property), 164  
     *method*), 164  
`recommended_data_shape()` (*hdmf.data\_utils.AbstractDataChunkIterator* property), 144  
     *method*), 144  
`recommended_data_shape()` (*hdmf.data\_utils.DataChunkIterator* property), 147  
     *method*), 147  
`recommended_data_shape()` (*hdmf.data\_utils.GenericDataChunkIterator* property), 146  
     *method*), 146  
`recommended_data_shape()` (*hdmf.monitor.DataChunkProcessor* property), 164  
     *method*), 164  
`recommended_primary_dtypes` (*hdmf.spec.spec.DtypeHelper* property), 116  
`ref` (*hdmf.backends.hdf5.h5\_utils.H5Dataset* property), 130  
`ReferenceBuilder` (*class in hdmf.build.builders*), 96  
`ReferenceResolver` (*class in hdmf.query*), 165  
`ReferenceTargetNotBuiltError`, 99  
`RefSpec` (*class in hdmf.spec.spec*), 117  
`reftype` (*hdmf.spec.spec.RefSpec* property), 117  
`region` (*hdmf.build.builders.RegionBuilder* property), 97  
`region` (*hdmf.container.DataRegion* property), 91  
`region` (*hdmf.region.RegionSlicer* property), 166  
`REGION_REF_TYPE` (*hdmf.build.builders.DatasetBuilder* attribute), 96  
`RegionBuilder` (*class in hdmf.build.builders*), 96  
`regionref` (*hdmf.backends.hdf5.h5\_utils.H5Dataset* property), 130  
`RegionSlicer` (*class in hdmf.region*), 166  
`register_class()` (*in module hdmf.common*), 86  
`register_container_type()` (*hdmf.build.manager.TypeMap* property), 104  
     *method*), 104  
`register_generator()` (*hdmf.build.classgenerator.ClassGenerator* property), 97  
     *method*), 97  
`register_generator()` (*hdmf.build.manager.TypeMap* property), 102  
     *method*), 102  
`register_map()` (*hdmf.build.manager.TypeMap* property), 104  
     *method*), 104  
`register_map()` (*in module hdmf.common*), 86  
`register_spec()` (*hdmf.spec.catalog.SpecCatalog* property), 109  
     *method*), 109  
`remove_test_file()` (*in module hdmf.testing.utils*), 163  
`reorder_yaml()` (*hdmf.spec.write.YAMLSpecWriter* property), 127  
     *method*), 127  
`required` (*hdmf.spec.spec.AttributeSpec* property), 118  
`required` (*hdmf.spec.spec.BaseStorageSpec* property), 118  
`required` (*hdmf.spec.spec.LinkSpec* property), 122  
`reset_parent()` (*hdmf.container.AbstractContainer* property), 89  
     *method*), 89  
`resolve()` (*hdmf.backends.hdf5.h5\_utils.AbstractH5TableDataset* property), 131  
     *method*), 131  
`resolve()` (*hdmf.build.manager.Proxy* property), 99  
`resolve_spec()` (*hdmf.spec.spec.BaseStorageSpec* property), 118  
     *method*), 118  
`resolve_spec()` (*hdmf.spec.spec.DatasetSpec* property), 122  
     *method*), 122  
`resolve_spec()` (*hdmf.spec.spec.GroupSpec* property), 123  
     *method*), 123  
`resolved` (*hdmf.spec.spec.BaseStorageSpec* property), 118  
`roundtripContainer()` (*hdmf.testing.testcase.H5RoundTripMixin* property), 162  
     *method*), 162  
`roundtripExportContainer()` (*hdmf.testing.testcase.H5RoundTripMixin* property), 162  
     *method*), 162  
`Row` (*class in hdmf.container*), 92  
`RowGetter` (*class in hdmf.container*), 92

## S

`schema` (*hdmf.spec.namespace.SpecNamespace* property), 112  
`set_attribute()` (*hdmf.build.builders.BaseBuilder* property), 94  
     *method*), 94  
`set_attribute()` (*hdmf.build.builders.GroupBuilder* property), 94  
     *method*), 94

set\_attribute() (*hdmf.spec.spec.BaseStorageSpec method*), 120  
 set\_attributes() (*hdmf.backends.hdf5.h5tools.HDF5IO method*), 139  
 set\_data\_io() (*hdmf.container.Container method*), 90  
 set\_data\_io() (*hdmf.container.Data method*), 90  
 set\_dataio() (*hdmf.backends.hdf5.h5tools.HDF5IO class method*), 140  
 set\_dataio() (*hdmf.container.Data method*), 90  
 set\_dataset() (*hdmf.build.builders.GroupBuilder method*), 95  
 set\_dataset() (*hdmf.spec.spec.GroupSpec method*), 126  
 set\_group() (*hdmf.build.builders.GroupBuilder method*), 95  
 set\_group() (*hdmf.spec.spec.GroupSpec method*), 125  
 set\_init() (*hdmf.build.classgenerator.CustomClassGenerator class method*), 98  
 set\_init() (*hdmf.build.classgenerator.MCIClassGenerator class method*), 98  
 set\_link() (*hdmf.build.builders.GroupBuilder method*), 95  
 set\_link() (*hdmf.spec.spec.GroupSpec method*), 126  
 set\_modified() (*hdmf.container.AbstractContainer method*), 89  
 set\_written() (*hdmf.backends.utils.WriteStatusTracker method*), 143  
 setdefault() (*hdmf.utils.LabelledDict method*), 155  
 setUp() (*hdmf.testing.testcase.H5RoundTripMixin method*), 162  
 setUpContainer() (*hdmf.testing.testcase.H5RoundTripMixin method*), 162  
 shape (*hdmf.backends.hdf5.h5\_utils.H5Dataset property*), 130  
 shape (*hdmf.common.table.DynamicTableRegion property*), 84  
 shape (*hdmf.container.Data property*), 90  
 shape (*hdmf.data\_utils.DataIO property*), 150  
 shape (*hdmf.spec.spec.AttributeSpec property*), 118  
 shape (*hdmf.spec.spec.DatasetSpec property*), 122  
 SHAPE\_ERROR (*hdmf.data\_utils.ShapeValidatorResult attribute*), 149  
 ShapeError (*class in hdmf.validate.errors*), 156  
 ShapeValidatorResult (*class in hdmf.data\_utils*), 149  
 SimpleMultiContainer (*class in hdmf.common.multi*), 69  
 SimpleMultiContainerMap (*class in hdmf.common.io.multi*), 61  
 simplify\_cpd\_type() (*hdmf.spec.spec.DtypeHelper static method*), 116  
 slice (*hdmf.region.RegionSlicer property*), 166  
 sort\_keys() (*hdmf.spec.write.YAMLSpecWriter method*), 127  
 SortedArray (*class in hdmf.array*), 163  
 source (*hdmf.backends.io.HDMFIO property*), 141  
 source (*hdmf.build.builders.Builder property*), 93  
 source (*hdmf.build.builders.GroupBuilder property*), 94  
 source (*hdmf.build.manager.Proxy property*), 99  
 source (*hdmf.spec.namespace.SpecReader property*), 113  
 Spec (*class in hdmf.spec.spec*), 116  
 spec (*hdmf.build.objectmapper.ObjectMapper property*), 106  
 spec (*hdmf.validate.validator.Validator property*), 159  
 spec\_matches (*hdmf.validate.validator.SpecMatcher property*), 161  
 spec\_namespace\_cls (*hdmf.spec.namespace.NamespaceCatalog property*), 114  
 SpecCatalog (*class in hdmf.spec.catalog*), 109  
 SpecFileBuilder (*class in hdmf.spec.write*), 129  
 SpecMatcher (*class in hdmf.validate.validator*), 160  
 SpecMatches (*class in hdmf.validate.validator*), 160  
 SpecNamespace (*class in hdmf.spec.namespace*), 111  
 SpecReader (*class in hdmf.spec.namespace*), 113  
 SpecWriter (*class in hdmf.spec.write*), 127  
 StrDataset (*class in hdmf.utils*), 155  
 stringify() (*hdmf.backends.hdf5.h5\_utils.H5SpecWriter static method*), 133

## T

Table (*class in hdmf.container*), 92  
 table (*hdmf.common.table.DynamicTableRegion property*), 84  
 table (*hdmf.container.Row property*), 92  
 target (*hdmf.common.table.VectorIndex property*), 79  
 target (*hdmf.region.RegionSlicer property*), 166  
 target\_type (*hdmf.spec.spec.LinkSpec property*), 122  
 target\_type (*hdmf.spec.spec.RefSpec property*), 117  
 tearDown() (*hdmf.testing.testcase.H5RoundTripMixin method*), 162  
 TermSet (*class in hdmf.term\_set*), 167  
 termset (*hdmf.term\_set.TermSetWrapper property*), 167  
 TermSetWrapper (*class in hdmf.term\_set*), 167  
 test\_roundtrip() (*hdmf.testing.testcase.H5RoundTripMixin method*), 162  
 test\_roundtrip\_export() (*hdmf.testing.testcase.H5RoundTripMixin method*), 162  
 TestCase (*class in hdmf.testing.testcase*), 161  
 to\_dataframe() (*hdmf.common.alignedtable.AlignedDynamicTable method*), 66  
 to\_dataframe() (*hdmf.common.resources.HERD method*), 76  
 to\_dataframe() (*hdmf.common.table.DynamicTable method*), 83  
 to\_dataframe() (*hdmf.common.table.DynamicTableRegion method*), 84  
 to\_dataframe() (*hdmf.container.Table method*), 93

- [to\\_hierarchical\\_dataframe\(\)](#) (in module [hdmf.common.hierarchicaltable](#)), 68  
[to\\_spmat\(\)](#) ([hdmf.common.sparse.CSRMatrix](#) method), 78  
[to\\_uint\\_array\(\)](#) (in module [hdmf.utils](#)), 153  
[to\\_zip\(\)](#) ([hdmf.common.resources.HERD](#) method), 77  
[todict\(\)](#) ([hdmf.common.resources.Entity](#) method), 71  
[todict\(\)](#) ([hdmf.common.resources.EntityKey](#) method), 73  
[todict\(\)](#) ([hdmf.common.resources.File](#) method), 71  
[todict\(\)](#) ([hdmf.common.resources.Key](#) method), 70  
[todict\(\)](#) ([hdmf.common.resources.Object](#) method), 72  
[todict\(\)](#) ([hdmf.common.resources.ObjectKey](#) method), 74  
[transform\(\)](#) ([hdmf.container.Data](#) method), 91  
[type\\_hierarchy\(\)](#) ([hdmf.container.AbstractContainer](#) class method), 89  
[type\\_key\(\)](#) ([hdmf.spec.spec.BaseStorageSpec](#) class method), 119  
[type\\_map](#) ([hdmf.build.manager.BuildManager](#) property), 100  
[TypeConfigurator](#) (class in [hdmf.term\\_set](#)), 168  
[TypeDoesNotExistError](#), 97  
[TypeMap](#) (class in [hdmf.build.manager](#)), 102  
[types](#) ([hdmf.backends.hdf5.h5\\_utils.AbstractH5TableData](#) property), 130  
[types\\_key\(\)](#) ([hdmf.spec.namespace.SpecNamespace](#) class method), 112  
[TypeSource](#) (class in [hdmf.build.manager](#)), 101
- ## U
- [unload\\_type\\_config\(\)](#) ([hdmf.term\\_set.TypeConfigurator](#) method), 168  
[unload\\_type\\_config\(\)](#) (in module [hdmf.common](#)), 86  
[unmap\(\)](#) ([hdmf.build.objectmapper.ObjectMapper](#) method), 107  
[unmatched\\_builders](#) ([hdmf.validate.validator.SpecMatcher](#) property), 161  
[UnsupportedOperation](#), 141  
[UNVERSIONED](#) ([hdmf.spec.namespace.SpecNamespace](#) attribute), 112  
[update\(\)](#) ([hdmf.utils.LabelledDict](#) method), 155
- ## V
- [valid](#) ([hdmf.backends.hdf5.h5\\_utils.H5DataIO](#) property), 134  
[valid](#) ([hdmf.data\\_utils.DataIO](#) property), 150  
[valid\\_primary\\_dtypes](#) ([hdmf.spec.spec.DtypeHelper](#) attribute), 116  
[valid\\_types\(\)](#) ([hdmf.validate.validator.ValidatorMap](#) method), 158  
[validate\(\)](#) ([hdmf.term\\_set.TermSet](#) method), 167  
[validate\(\)](#) ([hdmf.testing.testcase.H5RoundTripMixin](#) method), 162  
[validate\(\)](#) ([hdmf.validate.validator.AttributeValidator](#) method), 159  
[validate\(\)](#) ([hdmf.validate.validator.BaseStorageValidator](#) method), 159  
[validate\(\)](#) ([hdmf.validate.validator.DatasetValidator](#) method), 160  
[validate\(\)](#) ([hdmf.validate.validator.GroupValidator](#) method), 160  
[validate\(\)](#) ([hdmf.validate.validator.Validator](#) method), 159  
[validate\(\)](#) ([hdmf.validate.validator.ValidatorMap](#) method), 158  
[validate\(\)](#) (in module [hdmf.common](#)), 87  
[validate\\_spec\(\)](#) (in module [hdmf.testing.validate\\_spec](#)), 163  
[Validator](#) (class in [hdmf.validate.validator](#)), 158  
[ValidatorMap](#) (class in [hdmf.validate.validator](#)), 158  
[value](#) ([hdmf.spec.spec.AttributeSpec](#) property), 117  
[value](#) ([hdmf.term\\_set.TermSetWrapper](#) property), 167  
[values\(\)](#) ([hdmf.build.builders.GroupBuilder](#) method), 95  
[VectorData](#) (class in [hdmf.common.table](#)), 78  
[VectorIndex](#) (class in [hdmf.common.table](#)), 79  
[version](#) ([hdmf.spec.namespace.SpecNamespace](#) property), 112  
[view\\_set](#) ([hdmf.term\\_set.TermSet](#) property), 167  
[vmap](#) ([hdmf.validate.validator.Validator](#) property), 159
- ## W
- [WARNING](#) ([hdmf.utils.AllowPositional](#) attribute), 150  
[which\(\)](#) ([hdmf.container.Table](#) method), 93  
[write\(\)](#) ([hdmf.backends.hdf5.h5tools.HDF5IO](#) method), 136  
[write\(\)](#) ([hdmf.backends.io.HDMFIO](#) method), 141  
[write\\_builder\(\)](#) ([hdmf.backends.hdf5.h5tools.HDF5IO](#) method), 139  
[write\\_builder\(\)](#) ([hdmf.backends.io.HDMFIO](#) method), 142  
[write\\_dataset\(\)](#) ([hdmf.backends.hdf5.h5tools.HDF5IO](#) method), 139  
[write\\_group\(\)](#) ([hdmf.backends.hdf5.h5tools.HDF5IO](#) method), 139  
[write\\_link\(\)](#) ([hdmf.backends.hdf5.h5tools.HDF5IO](#) method), 139  
[write\\_namespace\(\)](#) ([hdmf.backends.hdf5.h5\\_utils.H5SpecWriter](#) method), 133  
[write\\_namespace\(\)](#) ([hdmf.spec.write.SpecWriter](#) method), 127  
[write\\_namespace\(\)](#) ([hdmf.spec.write.YAMLSpecWriter](#) method), 127  
[write\\_spec\(\)](#) ([hdmf.backends.hdf5.h5\\_utils.H5SpecWriter](#) method), 133

`write_spec()` (*hdmf.spec.write.SpecWriter* method),  
127  
`write_spec()` (*hdmf.spec.write.YAMLSpecWriter*  
method), 127  
`WriteStatusTracker` (class in *hdmf.backends.utils*),  
143

## Y

`YAMLSpecReader` (class in *hdmf.spec.namespace*), 113  
`YAMLSpecWriter` (class in *hdmf.spec.write*), 127