Hierarchical Data Modeling Framework

HDMF

Release 2.5.7.post.dev1

Jun 09, 2021

Getting Started

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: Neurophysiology (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.

CHAPTER 1

Installing HDMF

HDMF requires having Python 3.6, 3.7, or 3.8 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
```

CHAPTER 2

Tutorials

2.1 ExternalResources

This is a user guide to interacting with the `ExternalResources` class. The ExternalResources 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.

2.1.1 Introduction

The *`ExternalResources`* class provides a way to organize and map user terms (keys) to multiple resources and entities from the 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 *`ExternalResources`* 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 `ExternalResources` 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` (e.g., a term) of an online `resource` (e.g., an ontology). That is, `(object, key)` refer to parts inside a file and `(resource, entity)` refer to an external resource outside of the file, and `ExternalResources` allows us to link the two. To reduce data redundancy and improve data integrity, `ExternalResources` stores this data internally in a collection of interlinked tables.

- *`KeyTable`* where each row describes a *`Key`*
- *`ResourceTable`* where each row describes a *`Resource`*
- *`EntityTable`* where each row describes an *`Entity`*
- *`ObjectTable`* where each row descibes an *`Object`*
- *`ObjectKeyTable`* where each row describes an *`ObjectKey`* pair identifying which keys are used by which objects.

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

Creating an instance of the ExternalResources class

```
from hdmf.common import ExternalResources
from hdmf.common import DynamicTable
from hdmf import Data

er = ExternalResources(name='example')
```

2.1.2 Using the add_ref method

add_ref is a wrapper function provided by the `ExternalResources` 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'])
er.add_ref(container=data, field='', key='Homo sapiens', resource_name='NCBI_Taxonomy
↪',
           resource_uri='https://www.ncbi.nlm.nih.gov/taxonomy', entity_id=
↪'NCBI:txid9606',
           entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?
↪id=9606')

er.add_ref(container=data, field='', key='Mus musculus', resource_name='NCBI_Taxonomy
↪',
           resource_uri='https://www.ncbi.nlm.nih.gov/taxonomy', entity_id=
↪'NCBI:txid10090',
           entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?
↪id=10090')
```

2.1.3 Using the add_ref method with get_resource

When adding references to resources, you may want to refer to multiple entities within the same resource. Resource names are unique, so if you call `add_ref` with the name of an existing resource, then that resource will be reused. You can also use the *get_resource* method to get the `Resource` object and pass that in to `add_ref` to reuse an existing resource.

```
# Let's create a new instance of ExternalResources.
er = ExternalResources(name='example')

data = Data(name="species", data=['Homo sapiens', 'Mus musculus'])
er.add_ref(container=data, field='', key='Homo sapiens', resource_name='NCBI_Taxonomy
↪',
           resource_uri='https://www.ncbi.nlm.nih.gov/taxonomy', entity_id=
↪'NCBI:txid9606',
           entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?
↪id=9606')

# Using get_resource
existing_resource = er.get_resource('NCBI_Taxonomy')
er.add_ref(container=data, field='', key='Mus musculus', resources_idx=existing_
↪resource,
           entity_id='NCBI:txid10090',
           entity_uri='https://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?
↪id=10090')
```

2.1.4 Using the add_ref method with a field

In the above example, the `field` keyword argument was empty because the data of the *Data* object passed in for the `container` argument was being associated with a resource. However, you may want to associate an attribute of a *Data* object with a resource or a dataset or attribute of a *Container* object with a resource. To disambiguate between these different fields, you can set the 'field' keyword.

```
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')
er.add_ref(container=genotypes, field='genotype_name', key='Rorb', resource_name='MGI␣
↪Ontology',
           resource_uri='http://www.informatics.jax.org/', entity_id='MGI:1346434',
           entity_uri="http://www.informatics.jax.org/probe/key/804614")
```

2.1.5 Using the get_keys method

This method returns a DataFrame of key_name, resource_table_idx, entity_id, and entity_uri. You can either have a single key object, a list of key objects, or leave the input paramters empty to return all.

```
# All Keys
er.get_keys()

# Single Key
er.get_keys(keys=er.get_key('Homo sapiens'))

# List of Specific Keys
er.get_keys(keys=[er.get_key('Homo sapiens'), er.get_key('Mus musculus')])
```

2.1.6 Using the get_key method

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

```
# The get_key method will return the key object of the unique (key, container, field).
key_object = er.get_key(key_name='Rorb', container=genotypes, field='genotype_name')
```

2.1.7 Using the add_ref method with a key_object

Sometimes you want to reference a specific key that already exists when adding new ontology data into `ExternalResources`.

```
er.add_ref(container=genotypes, field='genotype_name', key=key_object, resource_name=
↪'Ensembl',
           resource_uri='https://uswest.ensembl.org/index.html', entity_id=
↪'ENSG00000198963',
           entity_uri='https://uswest.ensembl.org/Homo_sapiens/Gene/Summary?db=core;
↪g=ENSG00000198963')

# Let's use get_keys to visualize
er.get_keys()
```

2.2 MultiContainerInterface

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

2.2.1 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)

2.2.2 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}
```

2.2.3 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')
```

2.2.4 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)
```

2.2.5 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]
```

2.2.6 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()
```

2.2.7 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')
```

2.2.8 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',
    }
```

2.2.9 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


class MultiCollectionHolder(MultiContainerInterface):

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

2.2.10 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
```

2.2.11 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
```

2.3 AlignedDynamicTable

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

2.3.1 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

2.3.2 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 initalizing *AlignedDynamicTable* is the same as in *DynamicTable*. See the *DynamicTable tutorial* 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(
    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()
```

2.3.3 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.

2.3.4 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*

```
# 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]
```

Out:

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

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

Out:

```
'Emma'
```

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

Out:

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

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

Out:

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

2.4 DynamicTable

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

2.4.1 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 *VectorData*, *VectorIndex*, and *DynamicTableRegion* objects.

2.4.2 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',
)
```

2.4.3 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(0, 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 100 and 200
table_set_ids = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col1, col2],
    id=[100, 200],
)
```

If a list of integers in 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(
```

(continues on next page)

(continued from previous page)

```
    name='my table',
    description='an example table',
    columns=[col1, col2],
    id=ElementIdentifiers(name='id', data=[100, 200]),
)
```

2.4.4 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.

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

You can supply an optional row ID to *DynamicTable.add_row*. If no ID is supplied, the automatic row IDs count up from 0.

```
table.add_row(
    col1=4,
    col2='d',
    id=10,
)
```

Note: Row IDs are not required to be unique. However, if `enforce_unique_id=True` is passed, then adding a row with an ID that already exists in the table will raise an error.

2.4.5 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 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
)
```

2.4.6 Enumerated 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 types, and has the advantage of communicating to downstream tools that the data is categorical in nature.

```
from hdmf.common.table import EnumData

# this column has a length of 5, not 3
enum_col = EnumData(
```

(continues on next page)

(continued from previous page)

```
    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],
)
```

2.4.7 Ragged array columns

A table column with a different number of elements for each row is called a ragged array. To initialize a *DynamicTable* with a ragged array column, pass both the *VectorIndex* and its target *VectorData* object 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'],
)
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],
)
```

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(
    name='col4',
    description='column #4',
    data=[1, 0, -1, 0, -1, 1, 1, -1],
    index=[3, 4, 6, 8],  # specify the end indices of data for each row
)
```

2.4.8 Referencing rows of other tables

You can create a column that references rows of another table using 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],
    table=table
)

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

table2 = DynamicTable(
    name='my table',
    description='an example table',
    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
)

dtr_idx = VectorIndex(
    name='table1_ref2_index',
    target=indexed_dtr_col,
    data=[2, 3, 5, 7],
)

table3 = DynamicTable(
```

(continues on next page)

(continued from previous page)

```
    name='my table',
    description='an example table',
    columns=[dtr_idx, indexed_dtr_col],
)
```

2.4.9 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*.

```
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='table that can be expanded after being saved to file',
    description='an example table',
    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*, *ElementIdentifiers*, and *DynamicTableRegion* 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` arg of *VectorIndex* will also need to be wrapped in *H5DataIO*.

2.4.10 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, and is recommended. 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.

2.4.11 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,
)
```

2.4.12 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
```

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]
```

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'
table[0:10:2, 'col1']  # get a list of elements from rows 0 to 10 (exclusive) in␣
↪steps of 2 at column 'col1'
table[10::-1, 'col1']  # get a list of elements from rows 10 to 0 in reverse order at␣
↪column 'col1'
table[slice(0, 10, 2), 'col1']  # equivalent to table[0:4:2, 'col1']
table[np.s_[0:10:2], 'col1']  # equivalent to table[0:10:2, 'col1']
```

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

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

2.4.13 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]]
```

2.4.14 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[0:10:2]  # get rows 0 to 10 (exclusive) in steps of 2
table[10::-1]  # get rows 10 to 0 in reverse order

# the following are equivalent to table[0:10:2]
table[slice(0, 10, 2)]
table[np.s_[0:10:2]]

# you can also index a DynamicTable with a list or 1-dimensional numpy array of
# integer values. This will raise an IndexError if any of the index values is
# out of bounds of the table.
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*.

Note: Do not access a set of rows by supplying a list/tuple of row indices. This syntax will instead return the table element at the row index corresponding to the first element of the list/tuple and the column index corresponding to the second element.

2.4.15 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

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

2.4.16 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
```

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
```

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.

2.4.17 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'][0:10:2]  # get a list of the 0th to 10th (exclusive) elements in steps␣
↪of 2
table['col1'][10::-1]  # get a list of the 10th to 0th elements in reverse order

# the following are equivalent to table['col1'][0:10:2]
table['col1'][slice(0, 10, 2)]
table['col1'][np.s_[0:10:2]]

# you can also index a column with a list or 1-dimensional numpy array of
# integer values. This will raise an IndexError if any of the index values is
# out of bounds of the table.
table['col1'][[0, 2]]
```

(continues on next page)

(continued from previous page)

```
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
```

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

Note: It is also possible to access columns by column index using `table[:, j]` and elements by row index and column index using `table[i, j]`. These are equivalent to `table.columns[j][:]` and `table.columns[j][i]` and are not recommended because they interact with the internal list of columns.

2.4.18 Nested ragged array columns

Each element within a column can be an n-dimensional array, and this is true for ragged array columns as well.

```
col5 = VectorData(
    name='col5',
    description='column #5',
    data=[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']],
)
col5_ind = VectorIndex(
    name='col5_index',
    target=col5,
    data=[2, 3],
)
```

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.

```
col6 = VectorData(
    name='col6',
    description='column #6',
    data=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm'],
)
col6_ind = VectorIndex(
    name='col6_index',
    target=col6,
    data=[3, 6, 8, 10, 12, 13],
)
col6_ind_ind = VectorIndex(
    name='col6_index_index',
    target=col6_ind,
    data=[2, 5, 6],
)
```

(continues on next page)

(continued from previous page)

```
# All indices must be added to the table
table_double_ragged_col = DynamicTable(
    name='my table',
    description='an example table',
    columns=[col6, col6_ind, col6_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, 'col6']  # returns [['a', 'b', 'c'], ['d', 'e', 'f']]
table_double_ragged_col['col6'][0]  # same as line above
table_double_ragged_col['col6'][0][1]  # returns ['d', 'e', 'f']
```

Accessing the column named 'col6' using square bracket notation will return the top-level *VectorIndex* for the column. Accessing the column named 'col6' using dot notation will return the *VectorData* object

```
table_double_ragged_col['col6']  # returns col6_ind_ind
table_double_ragged_col.col6  # returns col6
```

2.4.19 Creating custom DynamicTable subclasses

TODO

Defining `__columns__`

TODO

CHAPTER 3

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.

CHAPTER 4

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.

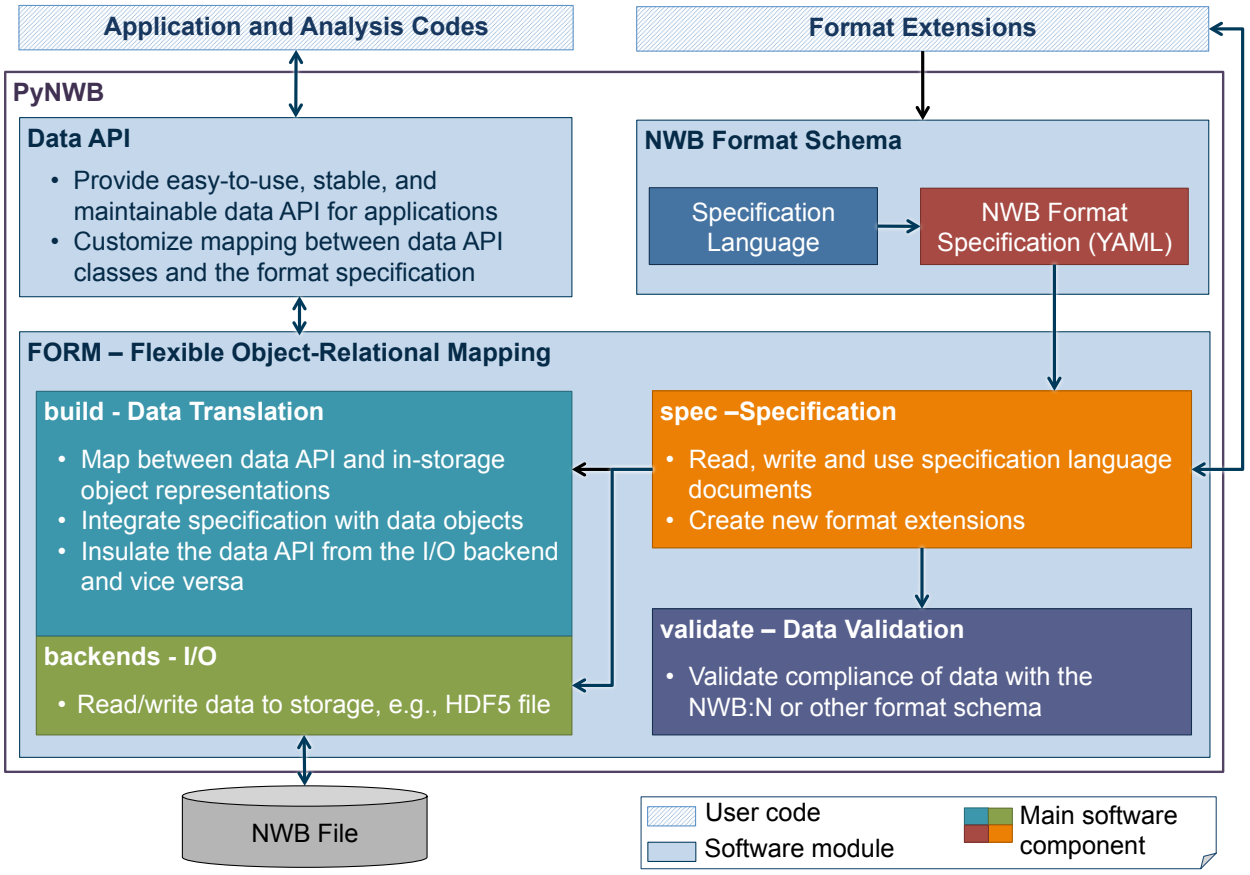

Fig. 1: Overview of the high-level software architecture of HDMF (click to enlarge).

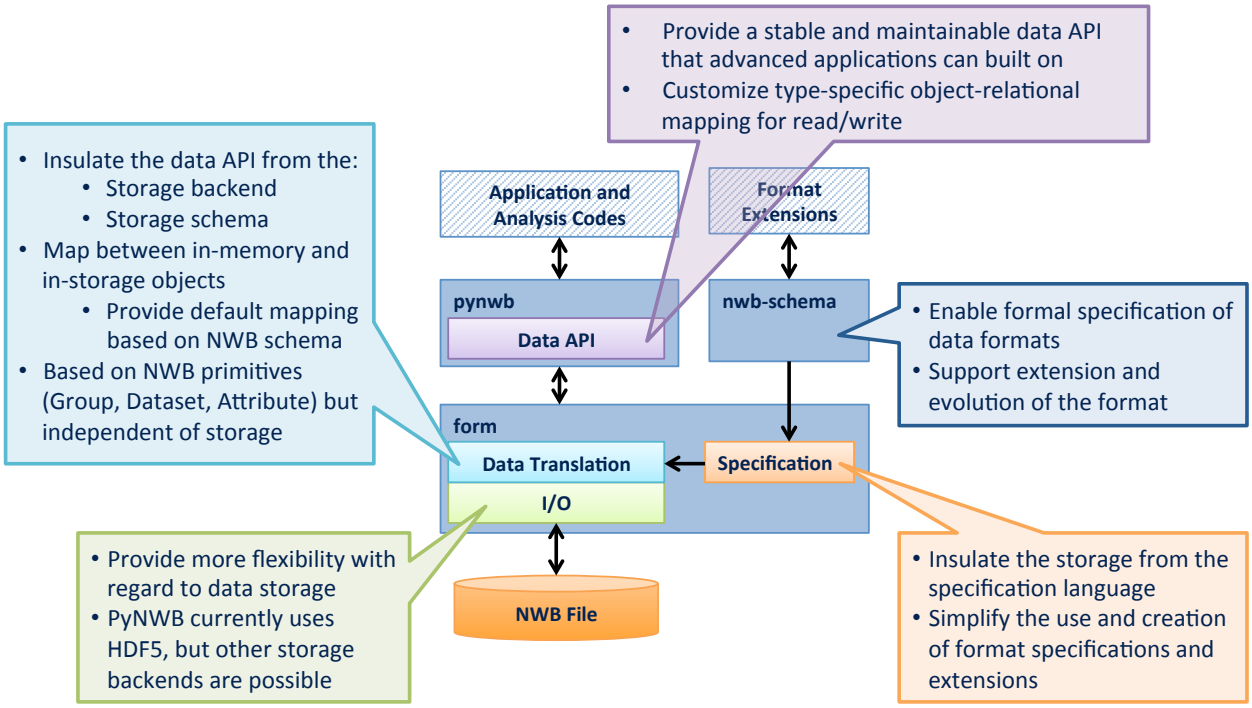

Fig. 2: We choose a modular design for HDMF to enable flexibility and separate the various levels of standardizing hierarchical data (click to enlarge).

4.1 Main Concepts

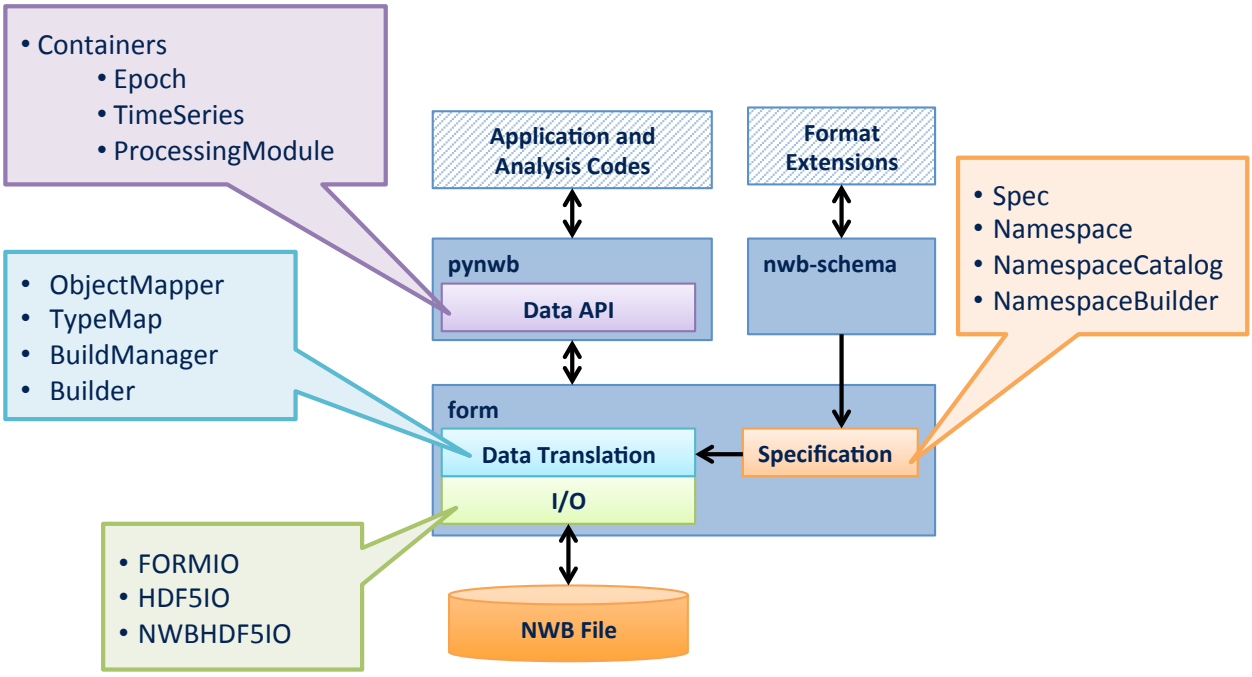

Fig. 3: Overview of the main concepts/classes in HDMF and their location in the overall software architecture (click to enlarge).

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*

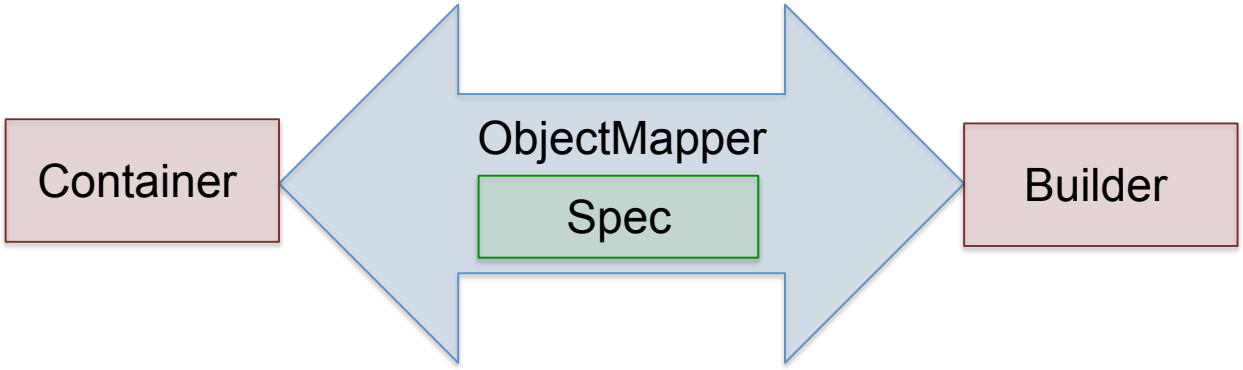

Fig. 4: Relationship between *Container*, *Builder*, *ObjectMapper*, and *Spec*

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`

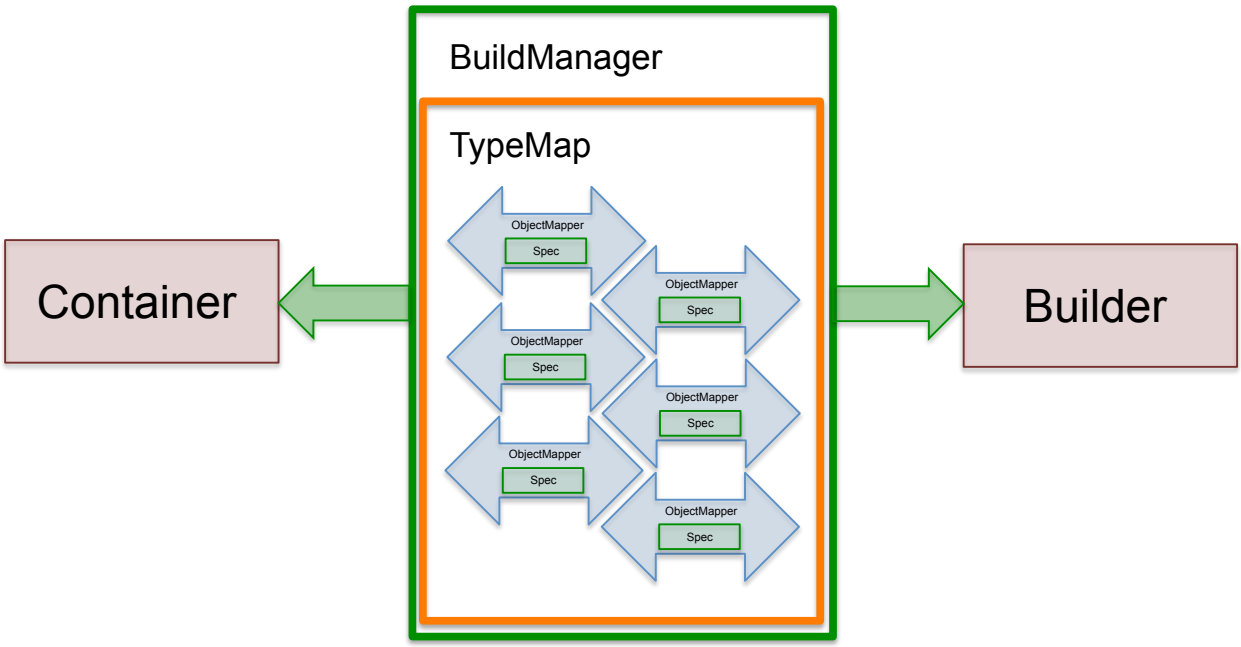

Fig. 5: Overview of *BuildManager* (and *TypeMap*) (click to enlarge).

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

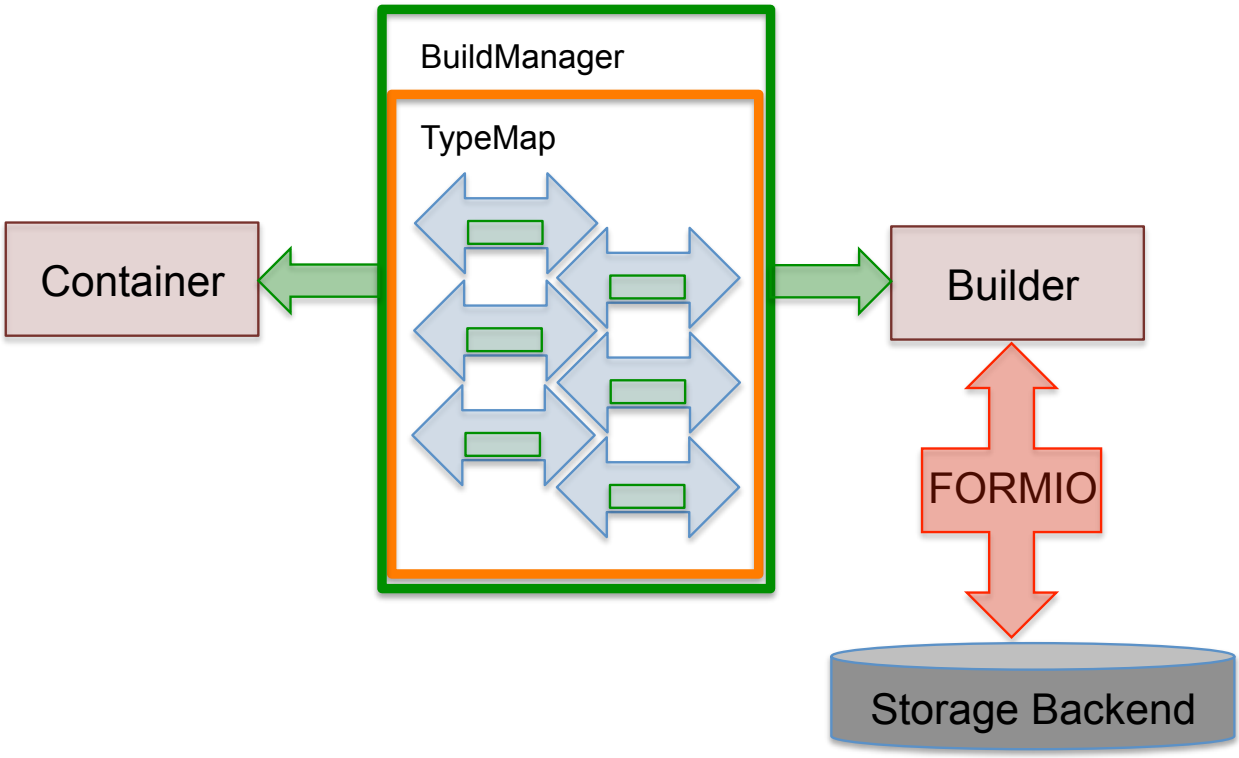

Fig. 6: Overview of *HDMFIO* (click to enlarge).

CHAPTER 5

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. F. {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 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 yourscript.py
```

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

```
DUECREDIT-ENABLE=yes python yourscript.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.

CHAPTER 6

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: *hdmf.common.io.table.DynamicTableMap*

Customize the mapping for AlignedDynamicTable

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: *hdmf.build.objectmapper.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>,

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

hdmf.common.io.resources module

class hdmf.common.io.resources.**ExternalResourcesMap**(*spec*)
Bases: *hdmf.build.objectmapper.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*)

resources (*builder*, *manager*)

entities (*builder*, *manager*)

objects (*builder*, *manager*)

object_keys (*builder*, *manager*)

constructor_args = {'entities': <function ExternalResourcesMap.entities>, 'keys': <f

obj_attrs = {}

hdmf.common.io.table module

class hdmf.common.io.table.**DynamicTableMap**(*spec*)
Bases: *hdmf.build.objectmapper.ObjectMapper*

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**
Bases: *hdmf.build.classgenerator.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.

classmethod set_init(*classdict*, *bases*, *docval_args*, *not_inherited_fields*, *name*)

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*, *category_tables=None*, *categories=None*)

Bases: *hdmf.common.table.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.

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 *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 *HDMFDataset* or *AbstractDataChunkIterator*) – the ordered names of the columns in this table. columns must also be provided.
- **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 *HDMFDataset* or *AbstractDataChunkIterator*) – List of names with the ordering of category tables

category_tables
Only regular DynamicTables are allowed. Using AlignedDynamicTable as a category for AlignedDynamicTable is currently not supported.

Type List of DynamicTables to be added to the container. NOTE

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'>*, *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 *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 *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 *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. 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

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*)

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

- **int, list, array, slice** : Return one or multiple row of the table as a DataFrame
- **string** : Return a single category table as a DynamicTable or a single column of the primary table as a
- **tuple**: Get a column, row, or cell from a particular category. The tuple is expected to consist of (category, selection) where category may be a string with the name of the sub-category or None (or the name of this AlignedDynamicTable) if we want to slice into the main table.

Returns DataFrame when retrieving a row or category. Returns scalar when selecting a cell. Returns a VectorData/VectorIndex when retrieving a single column.

data_type = 'AlignedDynamicTable'

namespace = 'hdmf-common'

hdmf.common.multi module

class hdmf.common.multi.**SimpleMultiContainer**(*name*, *containers=None*)
Bases: *hdmf.container.MultiContainerInterface*

Parameters

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

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 (<class 'hdmf.container.Container'>, <class 'hdmf.container.Data'>)

add_container(*containers*)
Add a Container to this SimpleMultiContainer

Parameters **containers** (list or tuple or dict or *Container* or *Data*) – the Container or Data to add

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 (<class 'hdmf.container.Container'>, <class 'hdmf.container.Data'>)

namespace = 'hdmf-common'

hdmf.common.resources module

class hdmf.common.resources.**KeyTable**(*name='keys'*, *data=[]*)

Bases: *hdmf.container.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 *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: *hdmf.container.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.**ResourceTable**(*name='resources'*, *data=[]*)

Bases: *hdmf.container.Table*

A table for storing names of ontology sources and their uri

Parameters

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

add_row(*resource*, *resource_uri*)

Parameters

- **resource** (str) – The resource/registry that the term/symbol comes from.
- **resource_uri** (str) – The URI for the resource term / registry symbol.

class hdmf.common.resources.**Resource**(*resource*, *resource_uri*, *table=None*, *idx=None*)
Bases: *hdmf.container.Row*

A Row class for representing rows in the ResourceTable

Parameters

- **resource** (str) – The resource/registry that the term/symbol comes from.
- **resource_uri** (str) – The URI for the resource term / registry symbol.
- **table** (*Table*) – None
- **idx** (int) – None

todict()

class hdmf.common.resources.**EntityTable**(*name='entities'*, *data=[]*)
Bases: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the data in this table

add_row(*keys_idx*, *resources_idx*, *entity_id*, *entity_uri*)

Parameters

- **keys_idx** (int or *Key*) – The index into the keys table for the user key that maps to the resource term / registry symbol.
- **resources_idx** (int or *Resource*) – The index into the ResourceTable.
- **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**(*keys_idx*, *resources_idx*, *entity_id*, *entity_uri*, *table=None*, *idx=None*)
Bases: *hdmf.container.Row*

A Row class for representing rows in the EntityTable

Parameters

- **keys_idx** (int or *Key*) – The index into the keys table for the user key that maps to the resource term / registry symbol.
- **resources_idx** (int or *Resource*) – The index into the ResourceTable.
- **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.**ObjectTable**(*name='objects'*, *data=[]*)
Bases: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the data in this table

add_row(*object_id*, *field*)

Parameters

- **object_id** (str) – The object ID for the Container/Data
- **field** (str) – The field on the Container/Data that uses an external resource reference key

class hdmf.common.resources.**Object**(*object_id*, *field*, *table=None*, *idx=None*)

Bases: *hdmf.container.Row*

A Row class for representing rows in the ObjectTable

Parameters

- **object_id** (str) – The object ID for the Container/Data
- **field** (str) – The field on the Container/Data that uses an external resource reference key
- **table** (*Table*) – None
- **idx** (int) – None

todict()

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

Bases: *hdmf.container.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 *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 key table that is used to make an external resource reference

class hdmf.common.resources.**ObjectKey**(*objects_idx*, *keys_idx*, *table=None*, *idx=None*)

Bases: *hdmf.container.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 key table that is used to make an external resource reference

- **table** (*Table*) – None
- **idx** (int) – None

todict()

class hdmf.common.resources.**ExternalResources**(*name*, *keys=None*, *resources=None*, *entities=None*, *objects=None*, *object_keys=None*)

Bases: *hdmf.container.Container*

A table for mapping user terms (i.e. keys) to resource entities.

Parameters

- **name** (str) – the name of this ExternalResources container
- **keys** (*KeyTable*) – the table storing user keys for referencing resources
- **resources** (*ResourceTable*) – the table for storing names of resources and their uri
- **entities** (*EntityTable*) – the table storing entity information
- **objects** (*ObjectTable*) – the table storing object information
- **object_keys** (*ObjectKeyTable*) – the table storing object-resource relationships

keys
the table storing user keys for referencing resources

resources
the table for storing names of resources and their uri

entities
the table storing entity information

objects
the table storing object information

object_keys
the table storing object-resource relationships

get_key(*key_name*, *container=None*, *field=None*)
Return a Key or a list of Key objects that correspond to the given key.

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

Parameters

- **key_name** (str) – the name of the key to get
- **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
- **field** (str) – the field of the Container that uses the key

get_resource(*resource_name=None*)
Retrieve resource object with the given resource_name.

Parameters **resource_name** (str) – None

add_ref(*container=None*, *field=None*, *key=None*, *resources_idx=None*, *resource_name=None*, *resource_uri=None*, *entity_id=None*, *entity_uri=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 and field. This method does not support such functionality by default. The different keys must be added separately using _add_key and passed to the *key* argument in separate calls of this method. If a resource with the same name already exists, then it will be used and the resource_uri will be ignored.

Parameters

- **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
- **field** (str) – the field of the Container/Data that uses the key
- **key** (str or *Key*) – the name of the key or the Row object from the KeyTable for the key to add a resource for
- **resources_idx** (*Resource*) – the resourcetable id
- **resource_name** (str) – the name of the resource to be created
- **resource_uri** (str) – the uri of the resource to be created
- **entity_id** (str) – the identifier for the entity at the resource
- **entity_uri** (str) – the URI for the identifier at the resource

get_keys(*keys=None*)

Return a DataFrame with information about keys used to make references to external resources.

The DataFrame will contain the following columns:

- *key_name*: the key that will be used for referencing an external resource
- *resources_idx*: the index for the resourcetable
- *entity_id*: the index for the entity at the external resource
- *entity_uri*: the URI for the entity at the external resource

It is possible to use the same *key_name* to refer to different resources so long as the *key_name* is not used within the same object and field. This method does not support such functionality by default. To select specific keys, use the *keys* argument to pass in the Key object(s) representing the desired keys. Note, if the same *key_name* is used more than once, multiple calls to this method with different Key objects will be required to keep the different instances separate. If a single call is made, it is left up to the caller to distinguish the different instances.

Parameters **keys** (list or *Key*) – the Key(s) to get external resource data for

Returns a DataFrame with keys and external resource data

Return type DataFrame

data_type = 'ExternalResources'

namespace = 'hdmf-experimental'

hdmf.common.sparse module

class hdmf.common.sparse.**CSRMatrix**(*data*, *indices=None*, *indptr=None*, *shape=None*, *name='csr_matrix'*)

Bases: *hdmf.container.Container*

Parameters

- **data** (csr_matrix or ndarray or Dataset) – 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 Dataset) – CSR index array
- **indptr** (ndarray or Dataset) – CSR index pointer array
- **shape** (list or tuple or ndarray) – 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: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – a dataset where the first dimension is a concatenation of multiple vectors

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*)

extend(*ar*, ***kwargs*)

data_type = 'VectorData'

namespace = 'hdmf-common'

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

Bases: *hdmf.common.table.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 *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

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 corrsponding 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: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – a 1D dataset containing identifiers

data_type = 'ElementIdentifiers'

namespace = 'hdmf-common'

class hdmf.common.table.**DynamicTable**(*name*, *description*, *id=None*, *columns=None*, *colnames=None*)
Bases: *hdmf.container.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 *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 *HDMFDataset* or *AbstractDataChunkIterator*) – the ordered names of the columns in this table. columns must also be provided.

description
a description of what is in this table

id
the identifiers for this table

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

columns
the columns in this table

add_row(*data=None*, *id=None*, *enforce_unique_id=False*)
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

add_column(*name*, *description*, *data=[]*, *table=False*, *index=False*, *enum=False*, *col_cls=<class 'hdmf.common.table.VectorData'>*)
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.

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 *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 *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 *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.

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*, ***kwargs*)

Select a subset from the table

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 (str, int) where the string identifies the column to select by name and the int selects the row
3) int, list of ints, or slice selecting a set of full rows in the table

Returns

1) If key is a string, then return array with the data of the selected column
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 consisting of one or more rows

Raises KeyError

to_dataframe(*exclude=None*)

Produce a pandas DataFrame containing this table's data.

Parameters **exclude** (set) – Set of columns to exclude from the dataframe

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: *hdmf.common.table.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 *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

table

The DynamicTable this DynamicTableRegion is pointing to

__getitem__(*arg*)

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

Subset the DynamicTableRegion

Parameters

- **arg** – 1) tuple consisting of (str, int) where the string defines the column to select and the int selects the row, 2) int or slice to select a subset of rows

- **df** – Boolean indicating whether we want to return the result as a pandas dataframe

Returns Result from self.table[....] with the appropritate 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.

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: *hdmf.common.table.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 *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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO* or *VectorData*) – lookup values for each integer in data

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.**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 the namespaces loaded from the given file

Return type tuple

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*, *mode*, *manager=None*, *comm=None*, *file=None*, *driver=None*)

A convenience method for getting an HDF5IO object

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) – a pre-existing h5py.File object
- **driver** (str) – driver for h5py to use when opening HDF5 file

6.2 hdmf.container module

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

Bases: object

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

classmethod get_fields_conf()

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

fields

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

modified

set_modified(*modified=True*)

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

children

add_child(*child=None*)

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

classmethod type_hierarchy()

container_source
The source of this Container

parent
The parent Container of this Container

class hdmf.container.**Container**(*name*)
Bases: *hdmf.container.AbstractContainer*

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

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

data_type = 'Container'

namespace = 'hdmf-common'

class hdmf.container.**Data**(*name*, *data*)
Bases: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the source of the data

data

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

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*)

data_type = 'Data'

namespace = 'hdmf-common'

class hdmf.container.**DataRegion**(*name*, *data*)
Bases: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the source of the data

data
The target data that this region applies to

region
The region that indexes into data e.g. slice or list of indices

class hdmf.container.**MultiContainerInterface**(*name*)
Bases: *hdmf.container.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.

idx
The index of this row in its respective Table

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: *hdmf.container.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 *HDMFDataset* or *AbstractDataChunkIterator* or *DataIO*) – the source of the data

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

path
The path of this builder.

name
The name of this builder.

source
The source of this builder.

parent
The parent builder of this builder.

class hdmf.build.builders.**BaseBuilder**(*name*, *attributes={}*, *parent=None*, *source=None*)
Bases: *hdmf.build.builders.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

location
The location of this Builder in its source.

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: *hdmf.build.builders.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.

source
The source of this Builder

groups
The subgroups contained in this group.

datasets
The datasets contained in this group.

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: *hdmf.build.builders.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 *HDMFDataset* or *AbstractDataChunkIterator* or str or int or float or bytes or bool or *DataIO* or DatasetBuilder or RegionBuilder or Iterable or datetime) – 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'

data
The data stored in the dataset represented by this builder.

chunks
Whether or not this dataset is chunked.

maxshape
The max shape of this dataset.

dtype
The data type of this dataset.

class hdmf.build.builders.**LinkBuilder**(*builder*, *name=None*, *parent=None*, *source=None*)
Bases: *hdmf.build.builders.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

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.

builder
The target builder object.

class hdmf.build.builders.**RegionBuilder**(*region*, *builder*)
Bases: *hdmf.build.builders.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.

region
The selected region of the target dataset.

hdmf.build.classgenerator module

class hdmf.build.classgenerator.**ClassGenerator**
Bases: object

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**
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**
Bases: *hdmf.build.classgenerator.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.

hdmf.build.errors module

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: *hdmf.build.errors.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: *hdmf.build.errors.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.

source
The source of the object e.g. file source

location
The location of the object. This can be thought of as a unique path

namespace
The namespace from which the data_type of this Proxy came from

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

namespace_catalog

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()

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

namespace

data_type

class hdmf.build.manager.**TypeMap**(*namespaces=None*, *mapper_cls=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

namespace_catalog

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 AbstractContainer 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

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 AbstractContainer 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

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: *hdmf.build.warnings.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: *hdmf.build.warnings.BuildWarning*

Raised when a required field is missing.

exception hdmf.build.warnings.**MissingRequiredWarning**
Bases: *hdmf.build.warnings.MissingRequiredBuildWarning*

Raised when a required field is missing.

exception hdmf.build.warnings.**OrphanContainerWarning**
Bases: *hdmf.build.warnings.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 calcuation 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'

full_name
String with full name or None

contact
String or list of strings with the contacts or None

author
String or list of strings with the authors or None

version
String, list, or tuple with the version or SpecNamespace.UNVERSIONED if the version is missing or empty

date
Date last modified or released.

Returns datetime object, string, or None

name
String with short name or None

doc

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

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

source

read_spec()

read_namespace()

class hdmf.spec.namespace.**YAMLSpecReader**(*indir='.'*)
Bases: *hdmf.spec.namespace.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*)

namespaces
The namespaces in this NamespaceCatalog

Returns a tuple of the available namespaces

Return type tuple

dataset_spec_cls
The DatasetSpec class used in this NamespaceCatalog

group_spec_cls
The GroupSpec class used in this NamespaceCatalog

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':

recommended_primary_dtypes = ['float', 'double', 'short', 'int', 'long', 'utf', 'ascii

valid_primary_dtypes = {'ascii', 'bool', 'bytes', 'datetime', 'double', 'float', 'floa

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: *hdmf.spec.spec.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

doc
Documentation on what this Spec is specifying

name
The name of the object being specified

parent
The parent specification of this specification

classmethod build_const_args(*spec_dict*)
Build constructor arguments for this Spec class from a dictionary

path

class hdmf.spec.spec.**RefSpec**(*target_type*, *reftype*)
Bases: *hdmf.spec.spec.ConstructableDict*

Parameters

- **target_type** (str) – the target type GroupSpec or DatasetSpec
- **reftype** (str) – the type of references this is i.e. region or object

target_type
The data_type of the target of the reference

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: *hdmf.spec.spec.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

dtype
The data type of the attribute

value
The constant value of the attribute. "None" if this attribute is not constant

default_value
The default value of the attribute. "None" if this attribute has no default value

required
True if this attribute is required, False otherwise.

dims
The dimensions of this attribute's value

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: *hdmf.spec.spec.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

default_name
The default name for this spec

resolved

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()

attributes
Tuple of attribute specifications for this specification

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'.

data_type_inc
The data type this specification inherits

data_type_def
The data type this specification defines

data_type
The data type of this specification

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: *hdmf.spec.spec.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

doc
Documentation about this component

name
The name of this component

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: *hdmf.spec.spec.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

dims
The dimensions of this Dataset

dtype
The data type of the Dataset

shape
The shape of the dataset

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: *hdmf.spec.spec.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

target_type
The data type of target specification

data_type_inc
The data type of target specification

is_many()

quantity
The number of times the object being specified should be present

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: *hdmf.spec.spec.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 youspecify name, quantity cannot be '*', '+', or an integer greater that 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 was inherited from a parent type

Parameters **spec** (*Spec* or str) – the specification to check

is_inherited_type(*spec*)
Returns True if *spec* represents a spec that was inherited from an included data_type

Parameters **spec** (*BaseStorageSpec* or str) – the specification to check

is_overridden_type(*spec*)
Returns True if *spec* represents a spec that was overriden by the subtype

Parameters **spec** (*BaseStorageSpec* or str) – the specification to check

get_data_type(*data_type*)
Get a specification by "data_type"

Parameters **data_type** (str) – the data_type to retrieve

groups
The groups specificed in this GroupSpec

datasets
The datasets specificed in this GroupSpec

links
The links specificed 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 youspecify name, quantity cannot be '*', '+', or an integer greater that 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

write_spec(*spec_file_dict*, *path*)

write_namespace(*namespace*, *path*)

class hdmf.spec.write.**YAMLSpecWriter**(*outdir='.'*)
Bases: *hdmf.spec.write.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

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

- **– NamespaceBuilder instance used to build the** (*ns_builder*) – namespace and extension
- **– Iterable of specs that represent new data types** (*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

class hdmf.backends.hdf5.h5_utils.**H5Dataset**(*dataset*, *io*)
Bases: *hdmf.query.HDMFDataset*

Parameters

- **dataset** (Dataset or *Array*) – the HDF5 file lazily evaluate
- **io** (HDF5IO) – the IO object that was used to read the underlying dataset

io

regionref

ref

shape

class hdmf.backends.hdf5.h5_utils.**DatasetOfReferences**(*dataset*, *io*)
Bases: *hdmf.backends.hdf5.h5_utils.H5Dataset*, *hdmf.query.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

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: *hdmf.query.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: *hdmf.query.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: *hdmf.backends.hdf5.h5_utils.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

types

dtype

__getitem__(*arg*)

resolve(*manager*)

class hdmf.backends.hdf5.h5_utils.**AbstractH5ReferenceDataset**(*dataset*, *io*)
Bases: *hdmf.backends.hdf5.h5_utils.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*)

dtype

class hdmf.backends.hdf5.h5_utils.**AbstractH5RegionDataset**(*dataset*, *io*)
Bases: *hdmf.backends.hdf5.h5_utils.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*)

dtype

class hdmf.backends.hdf5.h5_utils.**ContainerH5TableDataset**(*dataset*, *io*, *types*)
Bases: *hdmf.backends.hdf5.h5_utils.ContainerResolverMixin*, *hdmf.backends.hdf5.h5_utils.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 refernces 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: *hdmf.backends.hdf5.h5_utils.BuilderResolverMixin*, *hdmf.backends.hdf5.h5_utils.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 the represents the ReferenceResolver that resolves refernces 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: *hdmf.backends.hdf5.h5_utils.ContainerResolverMixin*, *hdmf.backends.hdf5.h5_utils.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 the represents the ReferenceResolver that resolves refernces 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: *hdmf.backends.hdf5.h5_utils.BuilderResolverMixin*, *hdmf.backends.hdf5.h5_utils.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 the represents the ReferenceResolver that resolves refernces 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: *hdmf.backends.hdf5.h5_utils.ContainerResolverMixin*, *hdmf.backends.hdf5.h5_utils.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 the represents the ReferenceResolver that resolves refernces 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: *hdmf.backends.hdf5.h5_utils.BuilderResolverMixin*, *hdmf.backends.hdf5.h5_utils.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 the represents the ReferenceResolver that resolves refernces 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: *hdmf.spec.write.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: *hdmf.spec.namespace.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: *hdmf.region.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*)

Bases: *hdmf.data_utils.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

get_io_params()
Returns a dict with the I/O parameters specifiedin in this DataIO.

static filter_available(*filter*, *allow_plugin_filters*)
Check if a given I/O filter is available

Parameters

- **filter** (*String, 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 wether the given filter is available

link_data

io_settings

valid
bool indicating if the data object is valid

hdmf.backends.hdf5.h5tools module

class hdmf.backends.hdf5.h5tools.**HDF5IO**(*path*, *mode*, *manager=None*, *comm=None*, *file=None*, *driver=None*)

Bases: *hdmf.backends.io.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) – a pre-existing h5py.File object
- **driver** (str) – driver for h5py to use when opening HDF5 file

comm
The MPI communicator to use for parallel I/O.

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*)

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.

export(*src_io*, *container=None*, *write_args={}*, *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={}*, *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()

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 this HDMFIO object to further reading/writing

close_linked_files()
Close all opened, linked-to files.

MacOS and Linux automatically releases 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*)

Parameters

- **parent** (Group) – the parent HDF5 object
- **builder** (*LinkBuilder*) – the LinkBuilder to write

Returns the Link that was created

Return type Link

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

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*)

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

Module contents

6.5.2 Submodules

hdmf.backends.io module

class hdmf.backends.io.**HDMFIO**(*manager=None*, *source=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

manager

The BuildManager this instance is using

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*)

Write a container to the IO source.

Parameters **container** (*Container*) – the Container object to write

export(*src_io*, *container=None*, *write_args={}*)

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)
```

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*

read_builder()
Read data and return the GroupBuilder representing it

Returns a GroupBuilder representing the read data

Return type *GroupBuilder*

write_builder(*builder*)
Write a GroupBuilder representing an Container object

Parameters **builder** (*GroupBuilder*) – the GroupBuilder object representing the Container

open()
Open this HDMFIO object for writing of the builder

close()
Close this HDMFIO object to further reading/writing

exception hdmf.backends.io.**UnsupportedOperation**
Bases: ValueError

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*)

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.

__iter__()
Return the iterator object

__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*

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.

dtype
Define the data type of the array

Returns NumPy style dtype or otherwise compliant dtype string

maxshape
Property describing the maximum shape of the data array that is being iterated over

Returns NumPy-style shape tuple indicating the maxiumum dimensions up to which the dataset may be resized. Axes with None are unlimited.

class hdmf.data_utils.**DataChunkIterator**(*data=None*, *maxshape=None*, *dtype=None*, *buffer_size=1*, *iter_axis=0*)
Bases: *hdmf.data_utils.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.

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.

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*

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

maxshape

Get a shape tuple describing the maximum shape of the array described by this DataChunkIterator. 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.

Returns Shape tuple. None is used for dimenwions where the maximum shape is not known or unlimited.

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

dtype
Data type of the values in the chunk

Returns np.dtype of the values in the DataChunk

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, np.ndarray, DataChunkIterator etc.*) – The first input array
- **data2** (*List, Tuple, np.ndarray, DataChunkIterator etc.*) – The second input array
- **name1** – Optional string with the name of data1
- **name2** – Optional string with the name of data2
- **axes1** (*int, Tuple of ints, List of ints, or 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 validaton 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 = {None: 'All required axes matched', 'NUM_DIMS_ERROR': 'Unequal number o

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*)

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 *HDMFDataset* or *AbstractDataChunkIterator*) – the data to be written

get_io_params()

Returns a dict with the I/O parameters specified in this DataIO.

data

Get the wrapped data object

append(*arg*)

extend(*arg*)

__getitem__(*item*)

Delegate slicing to the data object

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**

Bases: enum.Enum

An enumeration.

ALLOWED = 1

ERROR = 3

WARNING = 2

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.**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*)

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.

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
- **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

class hdmf.utils.**ExtenderMeta**(*name*, *bases*, *classdict*)
Bases: abc.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*)

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, any object that support __len__ or shape.*) – Array for which we should determine the 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.

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 the 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

label
Return the label of this LabelledDict

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.

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

name

reason

location

class hdmf.validate.errors.**DtypeError**(*name*, *expected*, *received*, *location=None*)

Bases: *hdmf.validate.errors.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: *hdmf.validate.errors.Error*

Parameters

- **name** (str) – the name of the component that is erroneous
- **location** (str) – the location of the error

class hdmf.validate.errors.**MissingDataType**(*name*, *data_type*, *location=None*, *missing_dt_name=None*)

Bases: *hdmf.validate.errors.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

data_type

class hdmf.validate.errors.**IncorrectQuantityError**(*name*, *data_type*, *expected*, *received*, *location=None*)

Bases: *hdmf.validate.errors.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

class hdmf.validate.errors.**ExpectedArrayError**(*name*, *expected*, *received*, *location=None*)

Bases: *hdmf.validate.errors.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: *hdmf.validate.errors.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.**IllegalLinkError**(*name*, *location=None*)
Bases: *hdmf.validate.errors.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: *hdmf.validate.errors.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

hdmf.validate.validator module

hdmf.validate.validator.**check_type**(*expected*, *received*)
expected should come from the spec *received* should come from the data

hdmf.validate.validator.**get_iso8601_regex**()

exception hdmf.validate.validator.**EmptyArrayError**
Bases: Exception

hdmf.validate.validator.**get_type**(*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

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

spec

vmap

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: *hdmf.validate.validator.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: *hdmf.validate.validator.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: *hdmf.validate.validator.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: *hdmf.validate.validator.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.

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.

spec_matches
(spec, assigned builders)

Type Returns a list of tuples of

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: unittest.case.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*)
Asserts that the two AbstractContainers have equal contents. This applies to both Container and Data types.

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

assertBuilderEqual(*builder1*, *builder2*, *check_path=True*, *check_source=True*)
Test whether two builders are equal. Like assertDictEqual but also checks type, name, path, and source.

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()

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

data

get_data()

__getitem__(*arg*)

class hdmf.array.**AbstractSortedArray**(*data*)
Bases: *hdmf.array.Array*

An abstract class for representing sorted array

find_point(*val*)

get_data()

class hdmf.array.**SortedArray**(*array*)
Bases: *hdmf.array.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: *hdmf.array.SortedArray*

find_point(*val*)

hdmf.monitor module

exception hdmf.monitor.**NotYetExhausted**
Bases: Exception

class hdmf.monitor.**DataChunkProcessor**(*data*)
Bases: *hdmf.data_utils.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

process_data_chunk(*data_chunk*)

This method should take in a DataChunk, and process it.

Parameters **data_chunk** (*DataChunk*) – a chunk to process

compute_final_result()

Return the result of processing this stream Should raise NotYetExhaused exception

class hdmf.monitor.**NumSampleCounter**(***kwargs*)
Bases: *hdmf.monitor.DataChunkProcessor*

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*) – the HDF5 file lazily evaluate

__getitem__(*key*)

dataset

dtype

next()

class hdmf.query.**ReferenceResolver**
Bases: object

A base class for classes that resolve references

classmethod **get_inverse_class**()
Return the class the represents the ReferenceResolver that resolves refernces 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.

invert()
Return an object that defers reference resolution but in the opposite direction.

class hdmf.query.**BuilderResolver**
Bases: *hdmf.query.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: *hdmf.query.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: *hdmf.container.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

data
The target data. Same as self.target

region
The selected region. Same as self.slice

target
The target data

slice
The selected slice

__getitem__
Must be implemented by subclasses

__len__
Must be implemented by subclasses

class hdmf.region.**ListSlicer**(*dataset*, *region*)
Bases: *hdmf.region.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

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

CHAPTER 7

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',
                 data_type_def='MyExtendedSpikeEventSeries')

ext2 = GroupSpec('A custom EventDetection interface',
```

(continues on next page)

(continued from previous page)

```
                    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')
```

(continues on next page)

(continued from previous page)

```
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

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

CHAPTER 8

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')
```

(continues on next page)

(continued from previous page)

```
class MyContainer(Container):
    ...
```

register_class is used with *Data* the same way it is used with *Container*.

CHAPTER 9

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.

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, than 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.

CHAPTER 10

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.

CHAPTER 11

Installing HDMF for Developers

11.1 Set up a virtual environment

For development, we recommend installing HDMF in a virtual environment in editable mode. You can use the virtualenv tool to create a new virtual environment. Or you can use the conda package and environment management system for managing virtual environments.

11.1.1 Option 1: Using virtualenv

First, install the latest version of the `virtualenv` tool and use it to create a new virtual environment. This virtual environment will be stored in the `venv` directory in the current directory.

```
pip install -U virtualenv
virtualenv venv
```

On macOS or Linux, run the following to activate your new virtual environment:

```
source venv/bin/activate
```

On Windows, run the following to activate your new virtual environment:

```
venv\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.

11.1.2 Option 2: Using conda

First, install Anaconda to install the `conda` tool. Then create and activate a new virtual environment called "venv" with Python 3.8 installed.

```
conda create --name venv python=3.8
conda activate venv
```

Similar to a virtual environment created with `virtualenv`, 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.

11.2 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
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`, e.g.,

```
conda install --file=requirements.txt --file=requirements-dev.txt --file=requirements-
↪doc.txt
```

11.3 Run tests

You can run the full test suite with the following command:

```
python test.py
```

You could also run the full test suite by installing and running the `pytest` tool, a popular testing tool that provides more options for configuring test runs.

First, install `pytest`:

```
pip install pytest
```

Then run the full test suite:

```
pytest
```

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
```

Finally, 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 Python 3.6, 3.7, 3.8, and 3.9. This can take some time to run.

```
tox
```

11.4 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
```

CHAPTER 12

Contributing Guide

12.1 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.

12.2 Types of Contributions

12.2.1 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.

12.2.2 Did you write a patch that fixes a bug or implements a new feature?

See the *Contributing Patches and Changes* section below for details.

12.2.3 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.

12.2.4 Do you have questions about HDMF?

See our hdmf-dev.github.io website for details.

12.2.5 Informal discussions between developers and users?

The https://nwb-users.slack.com slack is currently used for informal discussions between developers and users.

12.3 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 `flake8` 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.

12.4 Styleguides

12.4.1 Python Code Styleguide

Before you create a Pull Request, make sure you are following the HDMF style guide (PEP8). To check whether your code conforms to the HDMF style guide, simply run the flake8 tool in the project's root directory.

```
$ flake8
```

12.4.2 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.

12.4.3 Documentation Styleguide

All documentations is written in reStructuredText (RST) using Sphinx.

12.5 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"."

12.6 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.

CHAPTER 13

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.

13.1 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.

13.1.1 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])
        indptr = np.array([0, 2, 3, 6])
        return CSRMatrix(data, indices, indptr, (3, 3))
```

CHAPTER 14

Software Process

14.1 Continuous Integration

HDMF is tested against Ubuntu, macOS, and Windows operating systems. The project has both unit and integration tests.

- Azure Pipelines runs all HDMF tests on Windows and macOS
- CircleCI runs all HDMF tests on Ubuntu

Each time a PR is published 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.

14.2 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.

14.3 Requirement Specifications

There are 5 kinds of requirements specification in HDMF.

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-doc.txt, which 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.

The final one is within setup.py, which contains a list of package dependencies and their version ranges allowed for running HDMF.

In order to check the status of the required packages, requires.io is used to create a badge on the project README. If all the required packages are up to date, a green badge appears.

If some of the packages are outdated, see *How to Update Requirements Files*.

14.4 Versioning and Releasing

HDMF uses versioneer for versioning source and wheel distributions. Versioneer creates a semi-unique release name for the wheels that are created. It requires a version control system (git in HDMF's case) to generate a release name. After all the tests pass, CircleCI creates both a wheel (*.whl) and source distribution (*.tar.gz) for Python 3 and uploads them back to GitHub as a release. Versioneer makes it possible to get the source distribution from GitHub and create wheels directly without having to use a version control system because it hardcodes versions in the source distribution.

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 CircleCI, typically named *hdmf-{version}.tar.gz*.

CHAPTER 15

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.

15.1 Prerequisites

- All CI tests are passing on CircleCI and Azure Pipelines.
- You have a GPG signing key.
- Dependency versions in requirements.txt, requirements-dev.txt, requirements-doc.txt, requirements-min.txt are up-to-date.
- Legal information and copyright dates are up-to-date in Legal.txt, license.txt, README.rst, docs/source/conf.py, and any other files.
- 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 ReadTheDocs on the 'latest' build.
- Release notes have been prepared.

- An appropriate new version number has been selected.

15.2 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.

15.3 Setting up environment

1. First, register for an account on PyPI.
2. If not already the case, ask to be added as a `Package Index Maintainer`.
3. Create a `~/.pypirc` file with your login credentials:

   ```
   [distutils]
   index-servers =
     pypi
     pypitest

   [pypi]
   username=<your-username>
   password=<your-password>

   [pypitest]
   repository=https://test.pypi.org/legacy/
   username=<your-username>
   password=<your-password>
   ```

   where `<your-username>` and `<your-password>` correspond to your PyPI account.

15.4 PyPI: Step-by-step

1. Make sure that all CI tests are passing on CircleCI and Azure Pipelines.
2. List all tags sorted by version

   ```
   $ git tag -l | sort -V
   ```

3. Choose the next release version number

   ```
   $ release=X.Y.Z
   ```

   Warning: To ensure the packages are uploaded on PyPI, tags must match this regular expression: `^[0-9]+(\.[0-9]+)*(\.post[0-9]+)?$`.

4. Download 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 builds on each CI services and automatically upload the wheels and source distribution on PyPI.

7. Check the status of the builds on CircleCI and Azure Pipelines.
8. Once the builds are completed, check that the distributions are available on PyPI and that a new GitHub release was created.
9. Create a clean testing environment to test the installation

```
$ mkvirtualenv hdmf-${release}-install-test && \
  pip install hdmf && \
  python -c "import hdmf; print(hdmf.__version__)"
```

Note: If the `mkvirtualenv` command is not available, this means you do not have virtualenvwrapper installed, in that case, you could either install it or directly use virtualenv or venv.

10. Cleanup

```
$ deactivate  && \
  rm -rf dist/* && \
  rmvirtualenv hdmf-${release}-install-test
```

15.5 Conda: Step-by-step

Warning: Publishing on conda requires you to have corresponding package version uploaded on PyPI. So you have to do the PyPI and Github release before you do the conda release.

In order to release a new version on conda-forge, follow the steps below:

1. Choose the next release version number (that matches with the pypi version that you already published)

```
$ release=X.Y.Z
```

2. Fork hdmf-feedstock

   First step is to fork hdmf-feedstock repository. This is the recommended best practice by conda.

3. Clone forked feedstock

   Fill the YOURGITHUBUSER part.

```
$ cd /tmp && git clone https://github.com/YOURGITHUBUSER/hdmf-feedstock.git
```

4. Download 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 and sha256.

   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 requirements-min.txt was changed, the changes should be reflected in the requirements/run list.

7. Push the changes

```
$ git push origin $release
```

8. Create a Pull Request

   Create a pull request against the main repository. If the tests pass, merge the PR, and a new release will be published on Anaconda cloud.

CHAPTER 16

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

16.1 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
```

16.2 requirements-(dev|doc).txt

Any of these requirements files can be updated using the following scripts:

```
cd hdmf

# Set the requirements file to update: requirements-dev.txt or requirements-doc.txt
```

(continues on next page)

(continued from previous page)

```
target_requirements=requirements-dev.txt

mkvirtualenv hdmf-requirements

# Install updated requirements
pip install -U -r $target_requirements

# 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
```

16.3 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.

CHAPTER 17

Copyright

CHAPTER 18

License

Python Module Index

h

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

T

U

V

W

Y