h5preserve Documentation

Release 0.19.0+0.gae96e87.dirty

James Tocknell

Jan 08, 2024

CONTENTS

h5preserve is a wrapper around h5py and hdf5, providing easier serialisation of native python types. Its design is inspired by Camel, and follows its philosophy.

CHAPTER

ONE

WHY USE H5PRESERVE?

The purpose of `h5preserve` is to provide a simple serialisation library to hdf5 files. Hence `h5preserve` has support for complex numerical data, multidimensional arrays etc., which other serialisation formats may not represent as effectively. `h5preserve` makes it easy to split out the interaction with hdf5 files from the main logic of your code. Since `h5preserve` is designed to hide the underlying hdf5 file, large files where memory usage is a concern do not work well with `h5preserve`. In this case, `h5preserve` provides easy access to the underlying h5py objects, or you may want to look at using pytables, which provides a more database-like interface to hdf5 files.

CHAPTER TWO

WHY THE NAME?

The name comes from the “h5” label associated with hdf5, and the idea of preserving or pickling data.

CHAPTER

THREE

CITING H5PRESERVE

If you find h5preserve useful in your research, please cite the JOSS paper, as well as h5preserve's dependencies, h5py, numpy and hdf5. Further information about citing h5preserve, including specific releases, can be found at *Citing h5preserve*.

Contents:

3.1 Quickstart

Assume you have a class which represents some experiment you've run:

```
class Experiment:
    def __init__(self, data, time_started):
        self.data = data
        self.time_started = time_started
```

where `data` is some numpy array containing the experimental data, and `time_started` is a string containing time and data when the experiment was started (we're using a string in this case, but it could be an `datetime` object from the python standard library, or some other representation of time).

To save an instance of `Experiment` to a group in a file, you could do:

```
experiment = Experiment(
    data=np.linspace(2e6, 5e6, 1000),
    time_started="2019-01-01T00:00:00.000000+00:00",
)

grp["MyExperiment"] = experiment.data
grp["MyExperiment"].attrs["time started"] = experiment.time_started
```

and read it back with:

```
experiment = Experiment(
    data=grp["MyExperiment"][:],
    time_started=grp["MyExperiment"].attrs["time started"]
)
```

which is fine but:

1. You could forget to slice/convert the dataset to a numpy array, and then try to use the dataset in a numerical expression. Also, if you are using a special subclass of numpy array, slicing will not return a instance of that class.

2. Similarly, hdf5 attributes can only represent certain types, and if you forget to convert back to the native python type…
3. Your experiment changes, and you need add or remove metadata, or your experiment becomes comprised of multiple datasets. You could keep track of versions, but it's another piece of metadata, and you will need to validate that the version in the file matches what is written.

This represents time spent coding up validation code, which has to be tested, and so forth. For short scripts, this can become come to dominate the code. Instead, using *h5preserve*, you can write a dump function, and a load function, and let *h5preserve* deal with the rest. For the above example, reading and writing become:

```
grp["MyExperiment"] = experiment
experiment = grp["MyExperiment"]
```

with our dump function being:

```
@registry.dumper(Experiment, "Experiment", version=1)
def _exp_dump(experiment):
    return DatasetContainer(
        data=experiment.data,
        attrs={
            "time started": experiment.time_started
        }
    )
```

and our load function being:

```
@registry.loader("Experiment", version=1)
def _exp_load(dataset):
    return Experiment(
        data=dataset["data"],
        time_started=dataset["attrs"]["time started"]
    )
```

3.2 Installing h5preserve

h5preserve is distributed via PyPI, and behaves like a normal python package; you can install it with pip by running:

```
pip install h5preserve
```

Further information about how to use pip to install Python packages can be found at https://packaging.python.org/tutorials/installing-packages/.

Note that h5preserve uses h5py, which may require a C compiler and the hdf5 library to install. On common systems (Windows, MacOS, most Linux), there are pre-built wheels for h5py, which will automatically be installed when installing h5preserve if they are available. Other systems should see the [h5py installation instructions](http://docs.h5py.org/en/latest/build.html) for more information about how to install h5py.

If you have any problems installing h5preserve, please file a issue at https://github.com/h5preserve/h5preserve/issues.

3.3 Usage

To understand how *h5preserve* works, you need to remember the following concepts:

dumper
A function which converts your object to a representation ready to be written to an HDF5 file. Has an associated class, version and class label.

loader
A function which converts a representation of a HDF5 object (group, dataset etc.) to an instance of a specified class. Has an associated version and class label.

registry
A collection of dumpers and loaders, providing a common namespace. *h5preserve* comes with a few which convert common Python types.

registry collection
A collection of registries. Deals with choosing the correct registry, dumper and loader to use, including version locking.

So a complete example based on the *Quickstart* example is:

```
import numpy as np
from h5preserve import (
    open as h5open, Registry, new_registry_list, DatasetContainer
)

registry = Registry("experiment")

class Experiment:
    def __init__(self, data, time_started):
        self.data = data
        self.time_started = time_started

@registry.dumper(Experiment, "Experiment", version=1)
def _exp_dump(experiment):
    return DatasetContainer(
        data=experiment.data,
        attrs={
            "time started": experiment.time_started
        }
    )

@registry.loader("Experiment", version=1)
def _exp_load(dataset):
    return Experiment(
        data=dataset["data"],
        time_started=dataset["attrs"]["time started"]
    )

my_cool_experiment = Experiment(np.array([1,2,3,4,5]), 10)

with h5open("my_data_file.hdf5", new_registry_list(registry), mode='w') as f:
    f["cool experiment"] = my_cool_experiment
```

(continues on next page)

(continued from previous page)

```
with h5open("my_data_file.hdf5", new_registry_list(registry), mode='r') as f:
    my_cool_experiment_loaded = f["cool experiment"]

print(
    my_cool_experiment_loaded.time_started ==
    my_cool_experiment.time_started
)
```

Whilst for this simple case it's probably overkill to use *h5preserve*, *h5preserve* deals quite easily changing requirements, such as adding additional properties to `Experiment` via versioning, splitting `Experiment` into multiple classes via recursively converting python objects, or even more complex requirements via being able to only read and convert when needed, or to dump subsets of a class before dumping the whole class.

The rest of this guide provides information about how to deal with specific topics (versioning, advanced loading and dumping), but these topics are not required to use *h5preserve*.

3.3.1 How Versioning Works

Valid versions for dumpers are either integers or `None`. Valid versions for loaders are integers, `None`, `any` or `all`. The order in which loaders are used are:

1. `None` if available
2. `all` if available
3. The version which is stored in the file (if available)
4. `any` if available

Dumpers are similar:

1. If a version of a dumper is locked, use that one
2. `None` if available
3. The latest version of the dumper available

Using `None` should not be done lightly, as it forces that the dumper and loader not change in any way, as there is no way of overriding which loader *h5preserve* uses when `None` is available. It may be better to have a dumper with an integer version and use a loader with a version of `all`, which can be modified at the python level, and not require modification of the existing file.

A versioning example

Imagine a class like `Experiment` above; you have some data, and some metadata (to keep the example simple, we're only going to have one piece of metadata, and no data):

```
class ModelOutput:
    def __init__(self, a):
        self.a = a
```

`a` represents some input parameter to our model. We also write the associated dumper and loader:

```
@registry.dumper(ModelOutput, "ModelOutput", version=1)
def _exp_dump(modeloutput):
    return DatasetContainer(
```

(continues on next page)

(continued from previous page)

```python
        attrs={
            "a": modeloutput.a
        }
    )

@registry.loader("ModelOutput", version=1)
def _exp_load(dataset):
    return ModelOutput(
        a=dataset["attrs"]["a"]
    )
```

However, later on we realise we should have used b instead of a. This could be because we want to radians instead of degrees, using b is more meaningful in the model, or some other reason we have, something which motivates a change to the class. We change our class:

```python
class ModelOutput:
    def __init__(self, b):
        self.b = b
```

and create a new dumper and loader for version 2 of this class:

```python
@registry.dumper(ModelOutput, "ModelOutput", version=2)
def _exp_dump(modeloutput):
    return DatasetContainer(
        attrs={
            "b": modeloutput.b
        }
    )

@registry.loader("ModelOutput", version=2)
def _exp_load(dataset):
    return ModelOutput(
        b=dataset["attrs"]["b"]
    )
```

But then, how do we load our old data? Let's assume that $b = 2a$. So we'd write a loader for version 1 which converts a to b:

```python
@registry.loader("ModelOutput", version=1)
def _exp_load(dataset):
    return ModelOutput(
        b = 2 * dataset["attrs"]["a"]
    )
```

What about a dumper? We can write one also, but it may be that we add additional metadata instead of changing its representation, so we can't store all our metadata in the version 1 format, so we can't write a dumper for version 1.

One thing *h5preserve* cannot do is check that your code is forward or backwards compatible between different versions, that has to be managed by the user (there's some code on providing some tools to help with automated testing of loaders and dumpers being written, but that will still require having something to test against).

Locking Dumper Version

It is possible to force which dumper version is going to used, via `RegistryContainer.lock_version()`. An example how to do this, given `Experiment` is a class you want to dump version 1 of, and `registries` is a instance of *`RegistryContainer`* which contains a *`Registry`* that can dump `Experiment` is:

```
registries = new_registry_list(registry)

registries.lock_version(Experiment, 1)
```

3.3.2 Controlling how Classes are Dumped

`h5preserve` will recursively dump arguments passed to *`GroupContainer`* or *`DatasetContainer`* (as well as any variations on those classes), as long as the arguments are supported by `h5py` for writing (e.g. numpy arrays), or there exists a dumper for each of the arguments. Hence, dumpers should only need to worry about name which each attribute of the class is saved to, and whether they should be saved as group/dataset attributes or as groups/datasets (currently there is no support for loaders/dumpers that only write group/dataset attributes without creating a new group/dataset).

Using `DatasetContainer` and `GroupContainer`

The *Quickstart* example above used *`DatasetContainer`*; *`DatasetContainer`* takes keyword arguments which are passed on to `h5py.Group.create_dataset()`, as well as an `attrs` keyword argument which is used to set attributes on the associated HDF5 dataset.

`GroupContainer` behaves similar to *`DatasetContainer`*; it also takes keyword arguments, as well as an additional `attrs` keyword argument. However, these keywords names are used as the name for the subgroup or dataset created from the keyword arguments. Modifying the *Quickstart* example to have it use a group instead of a dataset is simple, we just change the loader as shown below:

```
@registry.dumper(Experiment, "Experiment", version=1)
def _exp_dump(experiment):
    return GroupContainer(
        experiment_data=experiment.data,
        attrs={
            "time started": experiment.time_started
        }
    )
```

The start time is now written to an attribute on the HDF5 group, and `experiment.data` is written to either a dataset or group, depending on what type it is. If it was as above a numpy array, then it would be written as a dataset (but it would not have `"time started"` as an attribute). Loading from a group is the same as loading from a dataset:

```
@registry.loader("Experiment", version=1)
def _exp_load(group):
    return Experiment(
        data=group["experiment_data"],
        time_started=group["attrs"]["time started"]
    )
```

3.3.3 Using On-Demand Loading

The purpose of on-demand loading is to deal with cases where recursively loading a group takes up too much memory. On-demand loading requires modifications to the class which contains the objects which are to be loaded on-demand. The modifications are:

- Wrapping attributes and other objects which should be loaded on-demand with the *wrap_on_demand()* function when set, and unwrapping the objects when needed.
- Adding `cls._h5preserve_update()` as a callback function to be called when the class is dumped. This callback must wrap any of the above objects which are to be loaded on-demand with *wrap_on_demand()* as above.

wrap_on_demand() returns an instance of *OnDemandWrapper*, which can be called with no arguments to return the original object (similar to a weakref).

An example of the necessary code for class which subclasses `collections.abc.MutableMapping` and which stores its members in `_mapping` is:

```python
def __getitem__(self, key):
    value = self._mapping[key]
    if isinstance(value, OnDemandWrapper):
        value = value()
        self._mapping[key] = value # acting as cache, this can be skipped if desired
    return value

def __setitem__(self, key, val):
    self._mapping[key] = wrap_on_demand(self, key, val)

def _h5preserve_update(self):
    for key, val in self.items():
        self._mapping[key] = wrap_on_demand(self, key, val)
```

A workaround where a group/dataset takes up too much memory but on-demand loading is not set up is to open the file via `h5py` or use the *h5py_file* or *h5py_group* attribute to access the underlying `h5py.Group`. Using this group you can then access a subset of the groups that would be loaded, which you can pass to *H5PreserveGroup* to use your loaders.

3.3.4 Using Delayed Dumping

Delayed dumping is similar to on-demand loading, however it needs less changes to the containing class. Assigning an instance of *DelayedContainer* in the necessary location in the class is sufficient in preparing *h5preserve* for delayed dumping of the object. When the data is ready to be dumped, calling *write_container()* dumps the data to the file as if it has been dumped when the containing class had been dumped. In a class where it is an attribute which is to be dumped later, the following is sufficient:

```python
class ContainerClass:
    def __init__(self, data=None):
        if data is None:
            data = DelayedContainer()
        self._data = data

    @property
    def data(self):
        return self._data

```

(continues on next page)

(continued from previous page)

```
    @data.setter
    def data(self, data):
        if isinstance(self._data, DelayedContainer):
            self._data.write_container(soln)
            self._data = data
        else:
            raise RuntimeError("Cannot change data")
```

3.3.5 Built-in Loaders, Dumpers and Registries

h5preserve comes with a number of predefined loader/dumper pairs for built-in python types. The defaults for *new_registry_list()* automatically include these registries. If you do not wish to use the predefined registries, you should instead instantiate *RegistryContainer* manually.

The following table outlines the supported types, and how they are encoded in the HDF5 file.

Type	Encoding	Included by default
`None`	`h5py.Empty`	True
`int`	a dataset	True
`float`	a dataset	True
`bool`	a dataset	True
`bytes` (py2 `str`)	a dataset	True
`unicode` (py3 `str`)	a dataset	True
`tuple`	a dataset	True
`list`	a dataset	True

Manually Creating the Registry Container

To create the Registry Container manually, replace all calls to *new_registry_list()* with *RegistryContainer*. This will allow you to select which built-in registries (if any) you which to use. For example, if you only want to convert `None` to `h5py.Empty`, you would do:

```
from h5preserve import Registry, RegistryContainer
from h5preserve.additional_registries import none_python_registry

registry = Registry("my cool registry")

registries = RegistryContainer(registry, none_python_registry)
```

You could then pass `registries` to *h5preserve.open*, or lock to a specific version, or anything else you'd do after calling *new_registry_list()*.

3.4 Reference

h5preserve is a thin wrapper around h5py, providing easier serialisation of native python types.

copyright

(c) 2016 James Tocknell

license
3-clause BSD

`class h5preserve.`**`DatasetContainer`**`(`*attrs=None*, ***kwargs*`)`
Representation of an hdf5 dataset for use in h5preserve.

Parameters

- **`attrs`** (*`Mapping`*) – mapping containing the attributes of the group
- **`**kwargs`** – properties of the group, which get passed to create group

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

`get``(`*k*[, *d*]`)` → D[k] if k in D, else d. d defaults to None.

`items``()` → a set-like object providing a view on D's items

`keys``()` → a set-like object providing a view on D's keys

`pop``(`*k*[, *d*]`)` → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

`popitem``()` → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

`setdefault``(`*k*[, *d*]`)` → D.get(k,d), also set D[k]=d if k not in D

`update``(`[*E*,]***F*`)` → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

`values``()` → an object providing a view on D's values

`class h5preserve.`**`DelayedContainer`**
Helper class for allowing delayed writing of containers to hdf5 files.

`write_container``(`*data*`)`
Write *data* to hdf5 file with the associated located of the *DelayedContainer*.

`class h5preserve.`**`GroupContainer`**`(`*attrs=None*, ***kwargs*`)`
Representation of an hdf5 group for use in h5preserve.

Parameters

- **`attrs`** (*`Mapping`*) – mapping containing the attributes of the group
- **`**kwargs`** – datasets or subgroups to add to the group

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

`get``(`*k*[, *d*]`)` → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class h5preserve.**H5PreserveFile**(*h5py_file*, *registries*)

Thin wrapper around `h5py.File` to automatically use h5preserve when accessing the file contents.

Acts like `h5preserve.H5PreserveGroup`, but allows access to the associated `h5py.File` instance via `h5py_file`.

Parameters

- **h5py_file** (a `h5py.File`) – the hdf5 file to wrap
- **registries** (`RegistryContainer`) – the collection of registries that you want to use to read from the hdf5 file

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

create_group(*name*)

Creates a new group in the associated hdf5 file

Parameters
name (*string, or other identifier accepted by h5py*) – name of the new group

Returns
The new group wrapped by H5PreserveGroup

Return type
H5PreserveGroup

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

property h5py_file

the instance of `h5py.File` which `H5PreserveFile` wraps

Type
h5py.File

property h5py_group

the instance of `h5py.Group` which `H5PreserveGroup` wraps

Type
h5py.Group

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

require_group(*name*)

Returns the group associated with `name`, creating it if necessary.

Parameters
name (*string, or other identifier accepted by h5py*) – name of the desired group

Returns
The group wrapped by H5PreserveGroup

Return type
H5PreserveGroup

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class h5preserve.**H5PreserveGroup**(*h5py_group*, *registries*)

Thin wrapper around `h5py.Group` to automatically use h5preserve when accessing the group contents.

Parameters

- **h5py_group** (h5py.Group) –
- **registries** (`RegistryContainer`) – the collection of registries that you want to use to read from the hdf5 file

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

create_group(*name*)

Creates a new group in the associated hdf5 file

Parameters
name (*string, or other identifier accepted by h5py*) – name of the new group

Returns
The new group wrapped by H5PreserveGroup

Return type
H5PreserveGroup

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

property h5py_group

the instance of `h5py.Group` which `H5PreserveGroup` wraps

Type
h5py.Group

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

require_group(*name*)

Returns the group associated with `name`, creating it if necessary.

Parameters
name (*string, or other identifier accepted by h5py*) – name of the desired group

Returns
The group wrapped by H5PreserveGroup

Return type
H5PreserveGroup

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class h5preserve.**HardLink**(*obj*)

Represent a h5py hard link to be created via h5preserve.

Parameters
obj (*string, h5py.Group or h5py.Dataset*) – the h5py object that the hard link points to, can either be an h5py object, or a string with the absolute path of the object

property h5py_obj

the object which the hard link will point to

Type
h5py.Group or h5py.Dataset

property path

The path this object points to

class h5preserve.**OnDemandDatasetContainer**(*attrs=None*, ***kwargs*)

Subclass of *DatasetContainer* which supports accessing dataset data on demand, rather that loading immediately.

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

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class h5preserve.**OnDemandGroupContainer**(*attrs=None*, ***kwargs*)

Subclass of *GroupContainer* which supports accessing group members on demand, rather that loading immediately.

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

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*,]***F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

class h5preserve.**OnDemandWrapper**(*func*)

Wrapper which represents a container which can be accessed on demand.

class h5preserve.**Registry**(*name*)

Register of functions for converting between hdf5 and python.

This is the core of h5preserve, containing the information about how to convert to and from hdf5 files, what version to use, and the namespace of created data.

Parameters

name (*string*) – name of registry for identification purposes

dumper(*cls*, *label*, *version*)

Decorator function to create a dumper function.

Parameters

- **cls** (*any class*) – the class which this dumper operates on
- **label** (*string*) – the label or tag associated with this class

- **version** (*integer, None*) – The version of the output that this function returns.

freeze()

Freeze the registry, preventing further changes to the registry.

loader(*label*, *version*)

Decorator function to create a loader function.

Parameters

- **label** (*string*) – the label or tag associated with this class
- **version** (*integer, any, all, None*) – The version of the output that this function reads.

property name

name of the registry

Type

str

class h5preserve.RegistryContainer(**registries*)

Ordered container of registries which manages interaction with the hdf5 file.

Parameters

***registries** (*list of Registry*) – the list of registries to be associated with this container

append(*value*)

S.append(value) – append value to the end of the sequence

clear() → None -- remove all items from S

count(*value*) → integer -- return number of occurrences of value

dump(*obj*)

Dump native python object to h5preserve representation

Parameters

obj – the object to dump

extend(*values*)

S.extend(iterable) – extend sequence by appending elements from the iterable

from_file(*h5py_obj*)

Return an representation of a hdf5 object from a hdf5 file

Parameters

h5py_obj (a h5py object, e.g. group, dataset) –

index(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*index*, *value*)

S.insert(index, value) – insert value before index

load(*obj*)

Load native python object from h5preserve representation

Parameters

obj – the object to load

`lock_version`(*cls*, *version*)

Lock output version for a specific class

Parameters

- **cls** (*any class*) – the class to lock the version of
- **version** (*integer, any, all, None*) – the version which will always be used

`pop`([*index*]) → item -- remove and return item at index (default last).

Raise IndexError if list is empty or index is out of range.

`property registries`

Iterator over the registries contained in the order they were added.

`remove`(*value*)

S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

`reverse`()

S.reverse() – reverse *IN PLACE*

`to_file`(*h5py_group*, *key*, *val*)

Dump h5preserve object to hdf5 file

Parameters

- **h5py_group** (`h5py.Group`) – the group to add the object to
- **key** (*string*) – the name for the object
- **val** – the object to add

`h5preserve.new_registry_list`(**registries*)

Create a new list of registries which includes builtin registries.

Parameters

***registries** (*list of Registry*) – the list of registries to be associated with this container

`h5preserve.open`(*filename*, *registries*, ***, *mode*, ***kwargs*)

Open a hdf5 file wrapped with h5preserve.

Parameters

- **filename** (string, or other identifier accepted by `h5py.File`) –
- **registries** (`RegistryContainer`) – the collection of registries that you want to use to read from the hdf5 file
- ****kwargs** – additional keyword arguments to pass to `h5py.File`

`h5preserve.wrap_on_demand`(*obj*, *key*, *val*)

Wrap *val* such that it can be used on demand.

wrap_on_demand returns either the original *val* if *obj* has not yet been dumped, or a wrapped version of *val* if *obj* has been dumped.

wrap_on_demand automatically deals with wrapping/unwrapping if needed, so it is save to repeatedly call on the same object.

Parameters

- **obj** (*any dumpable object*) – the object which *val* is a member of or an attribute of
- **key** (*string*) – the key to be used when writing out *val*

- **val** (*any dumpable object*) – the object to be wrapped

3.5 Contributing to h5preserve

We welcome contributions to h5preserve, subject to our code of conduct whether it is improvements to the documentation or examples, bug reports or code improvements.

3.5.1 Reporting Bugs

Bugs should be reported to https://github.com/h5preserve/h5preserve. Please include what version of Python this occurs on, as well as which operating system. Information about your h5py and HDF5 configuration is also helpful.

3.5.2 Patches and Pull Requests

The main repository is https://github.com/h5preserve/h5preserve, please make pull requests against that repository, and the branch that pull requests should be made on is master (backporting fixes will be done separately if necessary).

3.5.3 Running the tests

h5preserve uses tox to run its tests. See https://tox.readthedocs.io/en/latest/ for more information about tox, but the simplest method is to run:

```
tox
```

in the top level of the git repository.

3.5.4 Making a release

Current minimal working method (this doesn't produce a release commit, deal with DOIs needing to be preregistered, not automated, not signed etc.):

1. Checkout the latest commit on the `master` branch on the main repository locally. Ensure the work directory is clean (`git purge`/`git clean -xfd`).
2. Tag this commit with an annotated tag, with the format being `v*.*.*` (`git tag -a v*.*.*`; I should sign these…). The tag should mention the changes in this release.
3. Push tag to github.
4. Create a release on github using the web interface, copying the content of the tag.
5. Build sdist and wheel (`python setup.py sdist bdist_wheel`), and upload to PyPI (`twine upload dist/*`).

3.6 Citing h5preserve

The preferred method of citing h5preserve is to cite the JOSS Paper, for which the bibtex entry is below:

```
@article{h5preserve,
  doi = {10.21105/joss.00581},
  url = {https://doi.org/10.21105/joss.00581},
  year  = {2018},
  month = {feb},
  publisher = {The Open Journal},
  volume = {3},
  number = {22},
  pages = {581},
  author = {James Tocknell},
  title = {h5preserve: Thin wrapper around h5py, inspired by camel},
  journal = {The Journal of Open Source Software}
}
```

DOIs of individual releases can be found on Zenodo, which can be cited in addition to the JOSS paper.

We also strongly encourage citing the dependencies of h5preserve, numpy and h5py.

CHAPTER FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX