itsh5py

Release 0.7.2

Max Elfner

Jan 28, 2023

CONTENTS

While there are many ways to store different data types, many of them have their drawbacks. hdf is a common way to store large arrays. Sometimes it can be practical to store arrays with additional (pythonic) data in a single file. While *hdf attributes* can support some types, many exception exists especially with python types.

This is a small implementation of recursive dict support for python to write and read *hdf-files* with many different pythonic data types. Almost all types implemented in default python and *numpy* should be supported, even in nested structures. The resulting files work in *hdfview* and *panoply* with some small drawbacks.

A major convenience is the ability to store iterables like lists and tuples, even in nested form. Mixed types are also supported.

Conversion, obscuration or changes to the saved types are kept at the bare minimum. So if, for any reasons, the files have to be used without *itsh5py*, all the data will be accessible with just a little added inconvenience.

CHAPTER

ONE

INSTALLATION

itsh5py is available on PyPI and can be readily installed via

```
pip install itsh5py
```

Run `pip uninstall itsh5py` in order to remove the package from your system.

To work, this requires some additional packages. Obviously, `h5py` is used for data storage. `numpy` is used for array handling. DataFrames are also supported using `pandas`. Finally, for serialization of difficult data types, *yaml* is used via `pyyaml`.

All the source packages above are available on PyPI for all common OS.

1.1 Limitations and warning

Some limitation still exist:

- While most of the core data types should be implemented, there is arbitrary complexity especially with nested iterables. Most likely there are still some cases and types which are not supported and may fail with different levels of grace. Since this package will most likely be used for data storage please always consider checking if *your* type is saved **and loaded** correctly. If in doubt, always open the file with `h5py.File()` and check. Feel free to report missing or buggy data types and they will be implemented if possible.
- *numpy* object arrays are not supported.
- Keys of the dictionary which will be saved should only be strings to avoid any ambiguity. Any other types are not tested and most likely will fail.
- Lazy slicing of arrays is not supported (yet).
- Long tuples and mixed type lists will be saved element-wise and thus be slow. This is recognizable starting at approx. 100 elements.
- Path object are supported as single datasets or as *list* or *tuple* iterables - however only non nested type.
- Closing a *LazyHdfDict* will close the file reference - even if another *LazyHdfDict* accesses the same file (which should not happen too often).

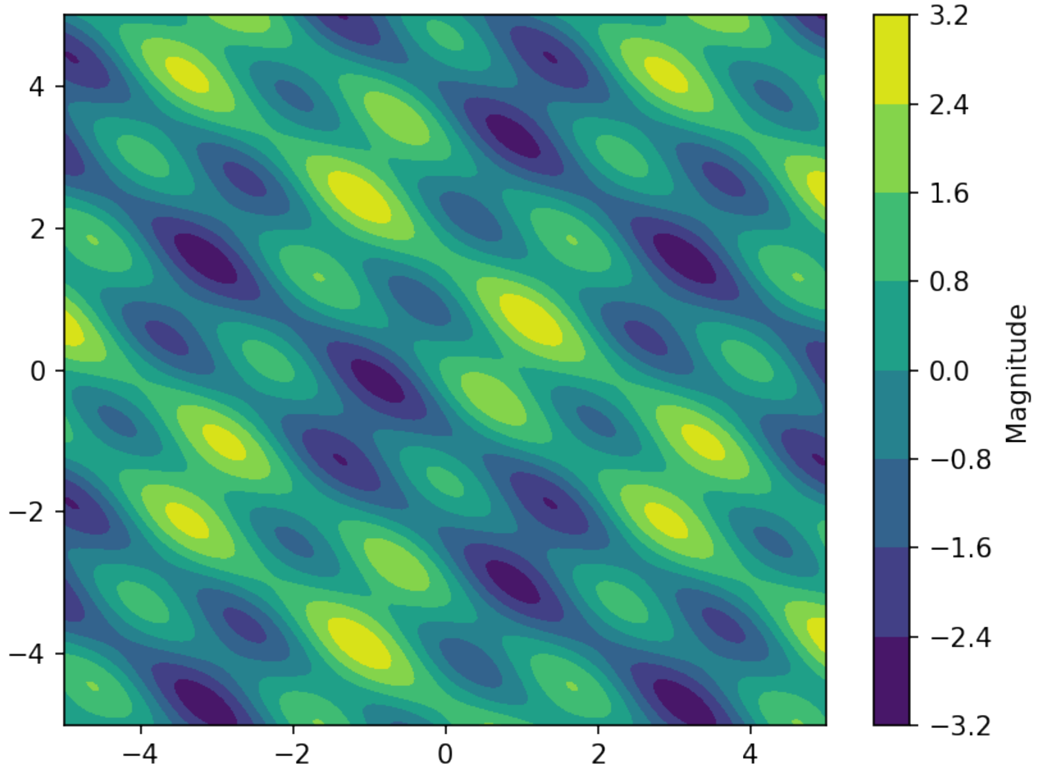
CHAPTER

TWO

TUTORIAL

Let's start with some sample data: Some coordinate arrays and a derived data field:

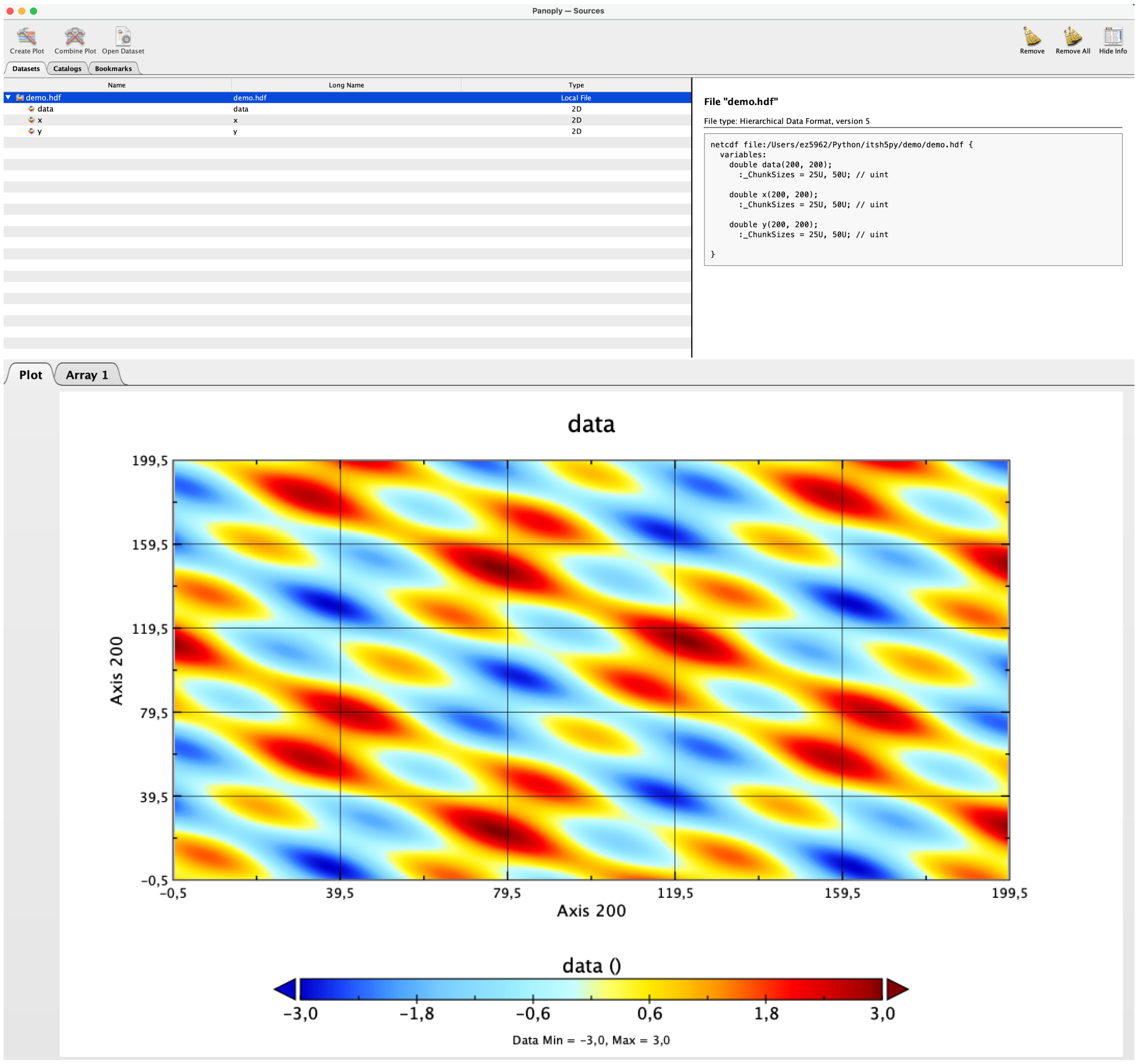
```
import numpy as np
import itsh5py

# Taken from mayavi examples!
x, y = np.mgrid[-5.:5.:200j, -5.:5:200j]
z = np.sin(x + y) + np.sin(2 * x - y) + np.cos(3 * x + 4 * y)
```



Saving this to hdf is obviously easy and possible with h5py. But it would mean creating a file, datasets and filling this manually. Using `itsh5py`, this is as easy as

```
itsh5py.save('demo', {'x': x, 'y': y, 'data': z})
```

Still, this is a default *hdf* file which can be opened and inspected in tools like *hdfview* or Panoply:



Name	Long Name	Type
demo.hdf	demo.hdf	Local File
data	data	2D
x	x	2D
y	y	2D

File "demo.hdf"

File type: Hierarchical Data Format, version 5

```
netcdf file:/Users/ez5962/Python/itsh5py/demo/demo.hdf {
  variables:
    double data(200, 200);
      :_ChunkSizes = 25U, 50U; // uint

    double x(200, 200);
      :_ChunkSizes = 25U, 50U; // uint

    double y(200, 200);
      :_ChunkSizes = 25U, 50U; // uint

}
```

However, sometimes you just need to store some metadata with your file and attributes just won't do it. Most of the types used in python are supported, thus

```
itsh5py.save('demo2', {'x': x, 'y': y, 'data': z, 'meta': ['type1', 2.]})
```

This can be inspected too:

File "demo2.hdf"

File type: Hierarchical Data Format, version 5

```
netcdf file:/Users/ez5962/Python/itsh5py/demo/demo2.hdf {
  variables:
    double data(200, 200);
      :_ChunkSizes = 25U, 50U; // uint

    double x(200, 200);
      :_ChunkSizes = 25U, 50U; // uint

    double y(200, 200);
      :_ChunkSizes = 25U, 50U; // uint

  group: meta {
    variables:
      String i_0;

      double i_1;

    // group attributes:
    :_TYPE_ = "list";
  }

}
```

As you can see, a mixed list is split into its elements since this type is not supported by *hdf*. For other types, other conversions exist. They will be visible when opening the files using h5py or similar but when loading them using itsh5py, they are converted back.

Loading can be done in two ways: *Lazy*, which keeps everything possible with weak references (default) or just loading all data. If lazy is active, the result is a *LazyHdfDict*:

```
lazy_demo = itsh5py.load('demo')
lazy_demo_2 = itsh5py.load('demo2')
itsh5py.config.use_lazy = False
basic_demo = itsh5py.load('demo')
basic_demo_2 = itsh5py.load('demo2')
```

Inspecting the results shows the following:

```
basic_demo
{
    'data': array([[ 0.59925318,  0.47366702,  0.38353246, ..., -0.93771155,
        -0.65135851, -0.36662565],
       [ 0.52598534,  0.43316361,  0.37683558, ..., -0.80275349,
        -0.52186765, -0.24853893],
       [ 0.46326134,  0.40428732,  0.38196713, ..., -0.66053162,
        -0.39007068, -0.13290833],
       ...,
       [ 1.24416509,  1.14695653,  1.03256892, ..., -2.31421298,
        -2.40068459, -2.44342076],
       [ 1.09745781,  0.99214212,  0.87544695, ..., -2.36468556,
        -2.42493877, -2.44148253],
```

(continues on next page)

(continued from previous page)

```
       [ 0.93395003,  0.82435405,  0.70941222, ..., -2.3818948 ,
        -2.41563926, -2.40663759]]),
    'x': array([[-5.        , -5.        , -5.        , ..., -5.        ,
        -5.        , -5.        ],
       [-4.94974874, -4.94974874, -4.94974874, ..., -4.94974874,
        -4.94974874, -4.94974874],
       [-4.89949749, -4.89949749, -4.89949749, ..., -4.89949749,
        -4.89949749, -4.89949749],
       ...,
       [ 4.89949749,  4.89949749,  4.89949749, ...,  4.89949749,
         4.89949749,  4.89949749],
       [ 4.94974874,  4.94974874,  4.94974874, ...,  4.94974874,
         4.94974874,  4.94974874],
       [ 5.        ,  5.        ,  5.        , ...,  5.        ,
         5.        ,  5.        ]]),
    'y': array([[-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
         4.94974874,  5.        ],
       [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
         4.94974874,  5.        ],
       [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
         4.94974874,  5.        ],
       ...,
       [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
         4.94974874,  5.        ],
       [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
         4.94974874,  5.        ],
       [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
         4.94974874,  5.        ]])
}

basic_demo_2
{
    'data': array([[ 0.59925318,  0.47366702,  0.38353246, ..., -0.93771155,
        -0.65135851, -0.36662565],
       [ 0.52598534,  0.43316361,  0.37683558, ..., -0.80275349,
        -0.52186765, -0.24853893],
       [ 0.46326134,  0.40428732,  0.38196713, ..., -0.66053162,
        -0.39007068, -0.13290833],
       ...,
       [ 1.24416509,  1.14695653,  1.03256892, ..., -2.31421298,
        -2.40068459, -2.44342076],
       [ 1.09745781,  0.99214212,  0.87544695, ..., -2.36468556,
        -2.42493877, -2.44148253],
       [ 0.93395003,  0.82435405,  0.70941222, ..., -2.3818948 ,
        -2.41563926, -2.40663759]]),
    'meta': ['type1', 2.0],
    'x': array([[-5.        , -5.        , -5.        , ..., -5.        ,
        -5.        , -5.        ],
       [-4.94974874, -4.94974874, -4.94974874, ..., -4.94974874,
        -4.94974874, -4.94974874],
       [-4.89949749, -4.89949749, -4.89949749, ..., -4.89949749,
        -4.89949749, -4.89949749],
```

(continues on next page)

(continued from previous page)

```
        ...,
        [ 4.89949749,  4.89949749,  4.89949749, ...,  4.89949749,
          4.89949749,  4.89949749],
        [ 4.94974874,  4.94974874,  4.94974874, ...,  4.94974874,
          4.94974874,  4.94974874],
        [ 5.        ,  5.        ,  5.        , ...,  5.        ,
          5.        ,  5.        ]]),
     'y': array([[-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
          4.94974874,  5.        ],
        [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
          4.94974874,  5.        ],
        [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
          4.94974874,  5.        ],
        ...,
        [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
          4.94974874,  5.        ],
        [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
          4.94974874,  5.        ],
        [-5.        , -4.94974874, -4.89949749, ...,  4.89949749,
          4.94974874,  5.        ]])
}
```

Which is, while not pretty, what was expected since it's the same as the input.

Taking a look at the *LazyHdfDict*, this is structured better:

```
demo.hdf
├─ /data::(200, 200)
├─ /x::(200, 200)
└─ /y::(200, 200)

demo2.hdf
├─ /data::(200, 200)
├─ Group /meta
│  ├─ /meta/i_0::b'type1'
│  └─ /meta/i_1::2.0
├─ /x::(200, 200)
└─ /y::(200, 200)
```

You can always *unlazy* a *LazyHdfDict* by either calling `dict()` or using the `.unlazy()` method. The latter is a wrapper that takes care of closing the then unused reference.

2.1 Attributes

Attributes can be used to add (scalar) quantities to *hdf* types (*Files*, *Groups*, *Datasets*). They can be loaded using the `unpack_attrs` option to `itsh5py.load()` which will place them in a `dict` called `attrs`. This is off by default. Otherwise, they can be accessed via the `h5py` backend, see below.

To quickly store some attributes with your data, you can use the same `attrs` key:

```
file = itsh5py.save('demo_att',
                    {'x': x, 'y': y, 'data': z,
```

(continues on next page)

(continued from previous page)

```
                    'attrs': {'additional_str': 'meta_string',
                              'addition_float': 100.,
                              },
                    })
reloaded = itsh5py.load(file)
```

```
reloaded:

demo_att.hdf
├─ /data::(200, 200)
├─ /x::(200, 200)
└─ /y::(200, 200)
```

While the attribute were added to the file, they are not loaded by default. Access them via either of the two methods:

```
[f'{k}: {v} (Type {type(v)})' for k, v in reloaded.h5file.attrs.items()]
["addition_float: 100.0 (Type <class 'numpy.float64'>)",
 "additional_str: meta_string (Type <class 'str'>)"]

reloaded = itsh5py.load(file, unpack_attrs=True)
reloaded['attrs']
{'addition_float': 100.0, 'additional_str': 'meta_string'}
```

2.2 h5py Backend

After loading lazy (by default), the underlying *hdf* can be accessed via the `LazyHdfDict.h5file` property. This allows the creation, extraction, slicing and so on with all basic `h5py` methods on the file.

2.3 Queue System

Open files (at least *lazy* ones) are stored in a queue. The handling Functions are mostly hidden and do not need to be accessed. However there are two things to not here: The amount of files open at once can be controlled via the `itsh5py.max_open_files` attribute. Currently open files can be shown using

```
itsh5py.open_filenames()
['demo2.hdf', 'demo.hdf']
```

There might be situations where large amounts of open files can be present, e.g. in list comprehensions. This can be handled in two ways:

1. Setting `itsh5py.max_open_files` to a large number. Be aware that this, combined with unlazy files, can be difficult for RAM and slow down the process considerably.
2. Using `itsh5.config.allow_fallback_open = True` (defaults to `False`). Since closing a *LazyHdfDict* does not remove the *python* instance, this allows to reopen a file on the fly to access unwrapped data from a previously open file. This will only open the file to get the data and subsequently close it again, preventing memory issues but also slowing down the process.

CHAPTER

THREE

USAGE

Overview documentation of the public API functions.

save	Adds keys of given dict as groups and values as datasets to the given hdf-file (by string or object) or group object.
load	Returns a dictionary containing the groups as keys and the datasets as values from given hdf file.
LazyHdfDict	Helps loading data only if values from the dict are requested.
queue_handler	Base module to handle the queue of open (in memory) files.
config	Package-wide config options

3.1 itsh5py.save

itsh5py.**save**(*hdf*, *data*, *compress=(True, 5)*, *packer=<function pack_dataset>*, **args*, ***kwargs*)

Adds keys of given dict as groups and values as datasets to the given hdf-file (by string or object) or group object. Iterative dicts are supported.

The dict can have the *attrs* key containing a dict of key, value pairs which are added as root level attributes to the hdf file. Those must be scalar, else exceptions will occur.

**args* and ***kwargs* will be passed to the *h5py.File* constructor.

Parameters

- **hdf** (*string*, *Path*) – Path to File
- **data** (*dict*) – The dictionary containing *only string or tuple* keys and data values or dicts as above again.
- **packer** (*callable*) – Callable gets *hdfobject, key, value* as input. *hdfobject* is considered to be either a h5py.File or a h5py.Group. *key* is the name of the dataset. *value* is the dataset to be packed and accepted by h5py. Defaults to *pack_dataset()*
- **compress** (*tuple*) – Try to compress arrays, use carefully. If on, gzip mode is used in every case. Defaults to *(False, 0)*. When *(True,…)* the second element specifies the level from *0-9*, see h5py doc.

Returns

hdf – Path to new file

Return type

string

3.2 itsh5py.load

`itsh5py.load`(*hdf*, *unpack_attrs=False*, *unpacker=<function unpack_dataset>*)

Returns a dictionary containing the groups as keys and the datasets as values from given hdf file.

Parameters

- **hdf** (*string, Path*) – Path to hdf file.
- **unpack_attrs** (*bool*, optional) – If True attrs from h5 file will be unpacked and are available as dict key attrs, no matter if lazy or not. Defaults to False.
- **unpacker** (*callable*) – Unpack function gets *value* of type h5py.Dataset. Must return the data you would like to have it in the returned dict.

Returns
result – The dictionary containing all groupnames as keys and datasets as values. Can be lazy and thus not unwrapped.

Return type
dict, *LazyHdfDict*

3.3 itsh5py.LazyHdfDict

`class itsh5py.LazyHdfDict`(*_h5file=None*, *group='/'*, **args*, ***kwargs*)

Helps loading data only if values from the dict are requested. This is done by reimplementing the __getitem__ method from dict. Other convenience functions are added to work with the hdf files as backend.

Parameters

- **_h5file** (*'h5py.File', optional*) – h5py File object or None
- **group** (*str*, optional) – Group to anchor the LazyHdfDict into.
- **args** – Passed to the parent *UserDcit* implemented type.
- **kwargs** – Passed to the parent *UserDcit* implemented type.

`__init__`(*_h5file=None*, *group='/'*, **args*, ***kwargs*)

Methods

`__init__`([_h5file, group])	
`clear`()	
`close`()	Closes the h5file if provided at initialization.
`copy`()	
`fromkeys`(iterable[, value])	
`get`(k[,d])	
`items`()	
`keys`()	
`pop`(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
`popitem`()	as a 2-tuple; but raise KeyError if D is empty.
`setdefault`(k[,d])	
`unlazy`()	Unpacks all datasets and closes the Lazy reference
`update`([E,]**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`()	

Attributes

`group`	Root group of the *LazyHdfDict*.
`h5file`	File handle of the *h5py.File()* object behind the *LazyHdfDict*.

property h5file

File handle of the *h5py.File()* object behind the *LazyHdfDict*.

property group

Root group of the *LazyHdfDict*.

unlazy()

Unpacks all datasets and closes the Lazy reference

close()

Closes the h5file if provided at initialization.

Unpackig will keep on working using the fallback routine if enabled.

3.4 itsh5py.queue_handler

Base module to handle the queue of open (in memory) files. The main important settings of how many filse are allowed (*max_open_files*) and the currently open files are exposed in the main API.

Functions

`add_open_file`(lazy_dict)	Adds a file (or better a LazyDict reference) to the queue.
`cleanup`()	This will be run atexit and ensures that no references persist in memory and all hdf files are freed.
`close`(lazy_dict)	Closes a LazyDict.
`is_open`(filepath)	Checks if a file is in the queue and thus oenened in memory.
`open_filenames`()	Show file paths of open files
`remove_from_queue`(file)	Removes file from the queue and from memory.

`itsh5py.queue_handler.`**`add_open_file`**(*lazy_dict*)

Adds a file (or better a LazyDict reference) to the queue.

`itsh5py.queue_handler.`**`is_open`**(*filepath*)

Checks if a file is in the queue and thus oenened in memory.

`itsh5py.queue_handler.`**`remove_from_queue`**(*file*)

Removes file from the queue and from memory. Only if file exists.

This is more complicated than it should be. The issue is that in the queue the actual LazyHdfDict are stored and comparison of those on remove can fail. So comaprison is done on file name basis and removal via index.

`itsh5py.queue_handler.`**`close`**(*lazy_dict*)

Closes a LazyDict. This is a small wrapper to check if close will work.

`itsh5py.queue_handler.`**`open_filenames`**()

Show file paths of open files

`itsh5py.queue_handler.`**`cleanup`**()

This will be run atexit and ensures that no references persist in memory and all hdf files are freed.

3.5 itsh5py.config

Package-wide config options

default_suffix: *str*, defaults to *.hdf*

Default suffix to use for saveing hdf files.

use_lazy: *bool*, defaults to *True*

Default setting for lazyness on loading.

default_compression: *tuple*, defaults to *(True, 5)*

Default setting for gzip compression. First element is yes or no, second is level of compression. See *h5py* docs for more details.

allow_fallback_open: *bool*, defaults to *True*
: If an item is unwrapped from a closed file (e.g. when holding many files open in long list comprehension), this allows a quick reopen and getting of a specified item. This can substantially slow down data handling, increase memory load and lead to access errors on files open by other applications.

allow_overwrite: *bool*, defaults to *False*
: If set to True, files will be overwritten if existing without warning. On default value of *False* the file mode will be *a* which is safe but can lead to exceptions if datasets already exist.

squeeze_single: *bool*, defaults to *False*
: If set to True, unpacked data containing a single key will be unpacked. This can lead to issues with single key dicts containing sub dicts, thus the default is the safer version (False)

max_tree_children: *int*, defaults to 30
: Maximum number of children in a group for the tree view to keep recursing. This can help to reduce tree size with very large files.

CHAPTER

FOUR

RELEASES

4.1 0.7.2

This is just a maintenance release after some time with just a small QoL feature when working with large files.

- Published on 2023-01-28
- Added support for the new config `max_tree_children` to reduce too large tree views with a default of 30 elements per group.
- Fixed some minor issues in the documentation
- Set minimum versions for dependencies
- Minimum Python version is set at 3.7 still - this might change when `h5py` updates their requirement.
- Increased some backend versions to reflect some changes in the past

4.2 0.7.1

- Published on 2022-2-2
- Changed affiliations and contact for main author
- Some minor code style changes

4.3 0.7.0

- Published on 2021-08-09
- Initial Release on PyPI

PYTHON MODULE INDEX

INDEX