pyhdf5io

Release 1.0.3

unknown

Oct 04, 2021

CONTENTS

This module defines a common interface for classes that can be saved to HDF5 format. This allows saving objects from different projects to the same HDF5 file.

CHAPTER

ONE

TUTORIAL, EXAMPLES AND DOCUMENTATION

1.1 Tutorial

In this tutorial we'll see how to add HDF5 serialization to classes. Let's start with defining a simple class:

```
In [1]: class Snek:
   ...:     def __init__(self, length):
   ...:         self.length = length
   ...:     def __repr__(self):
   ...:         return ':' + '=' * self.length + '>...'
   ...:

In [2]: Snek(10)
Out[2]: :==========>...
```

To make this Snek HDF5 serializable, we need to answer these questions three:

1. How is the Snek serialized to HDF5?
2. How is the HDF5 converted back into a Snek?
3. What is your favourite colour the unique tag identifying the Snek class?

To define how the Snek is serialized to HDF5, we add a `to_hdf5` method. This method is passed a `hdf5_handle`, which is a `h5py.File` or `h5py.Group` defining the (current) root of the HDF5 file where the object should be added.

For de-serialization, the `from_hdf5` classmethod should be implemented. Again, this method is passed a `hdf5_handle`. It should return the deserialized object.

Finally, the *subscribe_hdf5()* class decorator is used to define a unique `type_tag` which identifies this class.

Note: The `type_tag` needs to be unique across all projects using `fsc.hdf5_io`. For this reason, you should always prepend it with the name of your module.

```
In [3]: from fsc.hdf5_io import subscribe_hdf5, HDF5Enabled

In [4]: @subscribe_hdf5('my_snek_module.snek')
   ...: class HDF5Snek(Snek, HDF5Enabled):
   ...:     def to_hdf5(self, hdf5_handle):
   ...:         hdf5_handle['length'] = self.length
   ...:     @classmethod
   ...:     def from_hdf5(cls, hdf5_handle):
```

(continues on next page)

(continued from previous page)

```
   ...:         return cls(hdf5_handle['length'][()])
   ...:

In [5]: HDF5Snek(12)
Out[5]: :============>...
```

Notice also that we inherit from *HDF5Enabled*. This abstract base class checks for the existence of the HDF5 (de-)serialization functions, and adds methods `to_hdf5_file` and `from_hdf5_file` to save and load directly to a file.

Now we can use the *save()* and *load()* methods to save and load Sneks in HDF5 format:

```
In [6]: from fsc.hdf5_io import save, load

In [7]: from tempfile import NamedTemporaryFile

In [8]: mysnek = HDF5Snek(12)

In [9]: with NamedTemporaryFile() as f:
   ...:     save(mysnek, f.name)
   ...:     snek_clone = load(f.name)
   ...:

In [10]: snek_clone
Out[10]: :============>...
```

You can also save and load lists or dictionaries containing Sneks:

```
In [11]: with NamedTemporaryFile() as f:
   ....:     save([HDF5Snek(2), HDF5Snek(4)], f.name)
   ....:     snek_2, snek_4 = load(f.name)
   ....:

In [12]: print(snek_2, snek_4)
:==>... :====>...
```

A common use case is to serialize all the attributes of an object, a base class *SimpleHDF5Mapping* exists for this case. A subclass needs to define a lists `HDF5_ATTRIBUTES` of attributes that should be serialized. The attribute names must be the same as the arguments accepted by the constructor.

We can re-write the `Snek` as

```
In [13]: from fsc.hdf5_io import SimpleHDF5Mapping

In [14]: @subscribe_hdf5('my_snek_module.simplified_snek')
   ....: class SimplifiedHDF5Snek(Snek, SimpleHDF5Mapping):
   ....:     HDF5_ATTRIBUTES = ['length']
   ....:

In [15]: new_snek = SimplifiedHDF5Snek(9)

In [16]: with NamedTemporaryFile() as f:
   ....:     save(new_snek, f.name)
   ....:     new_snek_clone = load(f.name)
   ....:
```

(continues on next page)

(continued from previous page)

```
In [17]: new_snek_clone
Out[17]: :==========>···
```

We can extend the Snek functionality by adding a list of friends:

```
In [18]: @subscribe_hdf5('my_snek_module.snek_with_friends')
   ....: class SnekWithFriends(SimplifiedHDF5Snek):
   ....:     HDF5_ATTRIBUTES = SimplifiedHDF5Snek.HDF5_ATTRIBUTES + ['friends']
   ....:     def __init__(self, length, friends):
   ....:         super().__init__(length)
   ....:         self.friends = friends
   ....:

In [19]: snek_with_friends = SnekWithFriends(3, friends=[mysnek, new_snek])

In [20]: snek_with_friends
Out[20]: :===>···

In [21]: snek_with_friends.friends
Out[21]: [:============>···, :==========>···]

In [22]: with NamedTemporaryFile() as f:
   ....:     save(snek_with_friends, f.name)
   ....:     snek_with_friends_clone = load(f.name)
   ....:

In [23]: snek_with_friends_clone
Out[23]: :===>···

In [24]: snek_with_friends_clone.friends
Out[24]: [:============>···, :==========>···]
```

1.2 Entry points

The (de-)serialization methods registered with `fsc.hdf5-io` are only available once the Python module defining them has been loaded. To avoid having to explicitly `import` all necessary modules before loading a module, `fsc.hdf5-io` defines two *entry point groups*:

1.2.1 Serialization: `fsc.hdf5_io.save`

When a Python object whose serialization is not defined is encountered, `fsc.hdf5-io` will load (if it exists) the entry point corresponding to the full name of the object's class in the `fsc.hdf5_io.save` entrypoint. If there is no entry point for the *exact* Python name, it will also try its module name(s).

For example, if a `scipy.sparse.csr.csr_matrix` should be serialized, it will first check for an entry point named `scipy.sparse.csr.csr_matrix` in `fsc.hdf5_io.save`. If this entry point does not exist, it will try `scipy.sparse.csr`, `scipy.sparse`, and `scipy`, stopping at the first entry point that exists.

If you were to define serialization for all `scipy.sparse` objects in a module called `scipy_helpers.sparse.hdf5_io`, you could define the following entry point in the module `setup.py` (as an argument to the `setup` function):

```
entry_points={
    'fsc.hdf5_io.save': ['scipy.sparse = scipy_helpers.sparse.hdf5_io']
}
```

1.2.2 Deserialization: `fsc.hdf5_io.load`

The same principle applies for deserializing HDF5 objects, but the entry point names go by `type_tag` instead. For example, if you define `your_module` with type tags `your_module.some_object` and `your_module.another_object`, you have two choices:

If the top-level import of `your_module` loads all the submodules needed to deserialize both classes, the following configuration enables autoloading:

```
entry_points={
    'fsc.hdf5_io.load': ['your_module = your_module']
}
```

If instead they are in two separate submodules `some_object_submodule` and `another_object_submodule` that are *not* loaded when simply importing `your_module`, you need to define two entry points:

```
entry_points={
    'fsc.hdf5_io.load': [
        'your_module.some_object = your_module.some_object_submodule',
        'your_module.another_object = your_module.another_object_submodule',
    ]
}
```

As a real-world example, `fsc.hdf5-io` itself uses entry points to define the (de-)serialization of `sympy` objects, without always having to import `sympy`.

1.3 Documentation

This module contains functions to save and load objects, using the HDF5 format.

class fsc.hdf5_io.**HDF5Enabled**
Base class for data which can be serialized to and deserialized from HDF5.

class fsc.hdf5_io.**SimpleHDF5Mapping**
Base class for data classes which simply map their member to HDF5 values / groups.

The child class needs to define a list `HDF5_ATTRIBUTES` of attributes which should be serialized. The name of the attributes must correspond to the name accepted by the constructor.

For attributes which *can* be serialized but are not required, it can also define a list `HDF5_OPTIONAL`. The same logic as for the `HDF5_ATTRIBUTES` applies, but no error is raised if an attribute does not exist.

to_hdf5(*hdf5_handle*)
Serializes the object to HDF5 format, attaching it to the given HDF5 handle (might be a HDF5 File or Dataset).

fsc.hdf5_io.**from_hdf5**(*hdf5_handle*)
Deserializes the given HDF5 handle into an object.

Parameters **hdf5_handle** (`h5py.File` or `h5py.Group`.) – HDF5 location where the serialized object is stored.

fsc.hdf5_io.**from_hdf5_file**(*hdf5_file*)
Alias for *from_hdf5_file()*.

fsc.hdf5_io.**load**(*hdf5_file*)
Alias for *from_hdf5_file()*.

fsc.hdf5_io.**save**(*obj*, *hdf5_file*)
Alias for *to_hdf5_file()*.

fsc.hdf5_io.**subscribe_hdf5**(*type_tag*, *extra_tags=()*, *check_on_load=True*)
Class decorator that subscribes the class with the given type_tag for serialization.

Parameters

- **type_tag** (*str*) – Unique identifier of the class, which is injected into the HDF5 data to identify the class.
- **extra_tags** (*tuple(str)*) – Additional tags which should be deserialized to the given class.
- **check_on_load** (*bool*) – Flag that determines whether the 'type_tag' is checked when deserializing the object.

fsc.hdf5_io.**to_hdf5**(*obj*, *hdf5_handle*)
Serializes a given object to HDF5 format.

Parameters

- **obj** – Object to serialize.
- **hdf5_handle** (`h5py.File` or `h5py.Group`.) – HDF5 location where the serialized object gets stored.

fsc.hdf5_io.**to_hdf5_file**(*obj*, *hdf5_file*)
Alias for *to_hdf5_file()*.

fsc.hdf5_io.**to_hdf5_singledispatch**(*obj*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: collections.abc.Iterable*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: tuple*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: collections.abc.Mapping*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: numbers.Complex*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: numpy.str_*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: str*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: bytes*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: None*, *hdf5_handle*)
fsc.hdf5_io.**to_hdf5_singledispatch**(*obj: numpy.ndarray*, *hdf5_handle*)
Singledispatch function which is called to serialize and object when it does not have a `to_hdf5` method.

Parameters

- **obj** – Object to serialize.
- **hdf5_handle** (`h5py.File` or `h5py.Group`.) – HDF5 location where the serialized object gets stored.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

f

INDEX