**Python API for Neurodata Without Borders (NWB) format**

Version 1.0 (August 4, 2015)

**1. Overview.**

The Python API for the NWB format is a write API that can be used to create NWB files (it does not provide functionality for reading). It is implemented using a specification language and API that are domain-independent. The API provides a small set of generic functions for storing data in the file (that is, creating HDF5 groups and datasets). The specialization of the API to create NWB files is achieved by having the format defined using the specification language.

**2. Files.**

The Python API is implemented using five files:

1. h5gate.py
2. nwb\_core.py
3. nwb\_file.py
4. nwb\_init.py
5. nwb\_utils.py

The first, **h5gate.py**, is the main script that implements the API. It is domain and format independent in that it could be used to implement other formats both for neuroscience data and for other domains.

File **nwb\_core.py** contains a formal specification of the NWB format. It is written in a “specification language”. The specification language itself is also domain and format independent and is constructed using a JSON-like syntax.

Files **nwb\_file.py** and **nwb\_init.py** are used to setup the API and initialize a newly created NWB file. File **nwb\_utils.py** contains utility functions that are useful when creating NWB files.

**3. Using the API.**

To use the Python API, all of the above files must be in the Python path.

*3.1 Initialization.*

The initialization of a NWB file is performed using code like the following:

# import nwb api

import nwb\_file

# import utility routines

import nwb\_utils as ut

# Create the NWB file

fname='sample\_01.nwb'

start\_time='2015-07-27 8:34AM'

f = nwb\_file.nwb\_file(fname, start\_time)

In the above code variable “fname” stores the name of the NWB file being created and start\_time is the starting time of the session. (Parameter start\_time is optional. If not specified, the current time is used).

The call to function nwb\_file returns a “h5gate File” object. This is a Python object that controls the creation of an hdf5 file using the format specified by the specification language file. In the above call to nwb\_file, since only the file name is provided, the software uses the nwb\_core.py file for the format specification. Function nwb\_file has additional optional parameters that make it possible to specify other format specification files or multiple format specifications. This functionality allows extending the format by writing a specification for new format features and providing those in the call to nwb\_file. The extensions can be shared between labs and also stored in a central repository to allow forming a collection of commonly used extensions, some of which could be incorporated into the NWB “core” format. These optional parameters are described in section 4 (Using Extensions).

*3.2 Referencing groups and datasets*

The NWB Python API works by allowing the user to sequentially create hdf5 groups and datasets that conform to the specification of the format. To reference a group or dataset in a call to the API, the name of the group or dataset as given in the file format specification is used. Some groups and datasets have a name which is variable, that is, which is specified in the call to the API rather than in the format specification. (For example, group “electrode\_X” in the general/intracellular\_ephys group.) In the API calls, groups or datasets that have a variable name are referenced by enclosing the identifier associated with them in angle brackets, e.g. “<electrode\_X>”. Another example are NWB “Modules” which are stored in hdf5 groups. To create a module, a call to make\_group is used, e.g. f.make\_group("<module>", "shank-1").

*3.3 h5gate File and Group objects.*

The call to function nwb\_file.nwb\_file returns a “h5gate File” object (stored in variable “f” in the above example). Methods of this object are used to create groups and datasets in the hdf5 file that are, figuratively speaking, at the “top-level” of the file format specification, that is, not located inside groups that are defined in the specification language. Calls to the API functions which creates groups, return an “h5gate Group” object. This object has methods that are used to create groups and datasets within the associated hdf5 group. There is also a method (set\_attrs) which can be used set attributes associated with groups or datasets.

The methods of the h5gate File and Group objects and the set\_attrs function are described in the following sections.

*3.4 h5gate File object methods.*

The h5gate File object has the following methods (functions):

1. make\_group
2. make\_custom\_group
3. set\_dataset
4. set\_custom\_dataset
5. get\_node
6. close

*3.4.1 h5gate File make\_group*

Method “make\_group” of object h5gate File creates a group in the hdf5 file. It has the following signature:

g = f.make\_group(qid, name, path, attrs, link, abort)

“f” signifies an h5gate File object. The returned object, stored in variable ‘g’ in the above, is a “h5gate Group” object—which is described in the section 3.5. In the make\_group function, only the first argument (qid) is always required. The second two arguments (name and path) are sometimes required. All arguments are described below:

*qid,* is the “qualified id” for the group. The qualified id is the name used to reference the group (with surrounding angle brackets if the name is variable) optionally prefixed with a “namespace.” The namespace provides a way to associate extensions to the format with an identifier, in a manner similar to how namespaces are used in XML. This is described in section 4 (Using extensions). For normal uses (without extensions) the qid will just be the group name as given in the format specification.

*name* is only used if the group name is variable (referenced using <angle brackets>). It contains the name to be used when creating the group.

*path* specifies the path to the parent group within the hdf5 file. It is only needed if the location of the group within the file is ambiguous. For many groups, the location is not ambiguous and for those groups, the location is automatically determined by the API, without requiring a specification by argument “path”.

*attrs* is a Python dictionary containing hdf5 attributes keys and values to assign to the created group. It is optional.

*link* is used to create a hdf5 link to an existing group. If present it contains either a previously created h5gate Group object, or a string of the form “link:/path/to/group” or a string of the form: “extlink:path/to/file, path/to/group”. The first two are used to make a hdf5 link to a group within the current file. The third method specifies a link to a group in an external hdf5 file.

*abort* is a logical variable, default is True. It controls the program behavior if the group being created already exists. If abort is True the program will abort. If False, the function will return the previously existing group (h5gate Group object).

*3.4.2 h5gate File make\_custom\_group*

Method “make\_custom\_group” of object h5gate File creates a custom group in the hdf5 file (that is, a group that is not part of the format specification). This method is provided because method “make\_group” can only be used to create groups that are specified in the file format. The function signature for “make\_custom\_group” is:

g = f.make\_custom\_group(qid, name, path, attrs)

Return type and parameters are the same as for h5gate File method “make\_group”. If parameter “path” is not specified or is a relative path, then the custom group will be created in the default custom location, which is inside group “/general”.

*3.4.3 h5gate File set\_dataset*

Method “set\_dataset” of object h5gate File is used to create and store data in an hdf5 dataset. It has the following signature:

d = f.set\_dataset(qid, value, name, path, attrs, dtype, compress)

The return value is an object of type h5gate Dataset. The arguments are described below:

*qid* - the “qualified id” for the dataset. The qualified id is the name used to reference the dataset (with surrounding angle brackets if the name is variable) optionally prefixed with a “namespace” as described in the qid parameter for method make\_group.

*value* - value to store in the dataset. To store numeric or string values in the dataset (what is normally done) the value can be a scalar, Python list or tuple, or a Numpy array. To have the created dataset be a link to another Dataset, the value is set to an h5gate Dataset object or a string matching the pattern: link:/path/to/dataset (to link to a dataset within the file) or “extlink:path/to/file, /path/to/dataset” to link to a dataset in an external file.

*name* - name of the dataset in case the name is unspecified (qid is in <angle brackets>).

*path* - specified path of where to create the dataset (path to parent group). Only needed if the location of where to create the dataset is ambiguous in the format specification.

*attrs* - a dictionary containing hdf5 attributes keys and values to assign to the created group. It is optional.

*dtype* – type of data. If provided, this is included in the call to the library routine which creates the dataset (h5py.create\_dataset).

*compress* - if True, compression is specified in the call to the library routine which creates the dataset (h5py.create\_dataset). The default value is False. It is recommended that this be set True when saving large datasets in order to reduce the size of the generated file.

*3.4.4 h5gate File set\_custom\_dataset*

Method “set\_custom\_dataset” of object h5gate File creates a custom dataset in the hdf5 file (that is, a dataset that is not part of the format specification). The function signature is:

d = f.set\_custom\_dataset(qid, value, name, path, attrs, dtype, compress)

Return type and parameters are the same as for method “set\_dataset”.

*3.4.5 h5gate File get\_node*

Method “get\_node” returns the h5gate Group or Dataset object located at the specified location (full path in the hdf5 file). It has the following signature:

n = f.get\_node(full\_path, abort)

Arguments are:

*full\_path* – absolute path to group or dataset.

*abort* – A logical value that specifies what to do if there is no node (group or dataset) at the specified path. Default is True, which causes the program to abort. A value of False, causes the function to return None.

*3.4.6 h5gate File close*

Method “close” of object h5gate File is used to close the created file. It must be called to complete the creation of the file. Function signature is:

f.close()

There are no arguments.

**3.5 h5gate Group methods**

An h5gate Group object is returned by h5gate File methods “make\_group” and “make\_custom\_group”. The h5gate Group object has the following methods:

1. make\_group
2. make\_custom\_group
3. set\_dataset
4. set\_custom\_dataset
5. set\_attr

The name of the first four of these methods are the same as the name of methods of the h5gate File object. The difference between the File and Group object methods is that the h5gate File methods are used to create groups and datasets that are not inside groups that are defined as part of the format specification whereas the Group methods are used to create groups and datasets inside the current group, that is, inside the Group object used to call the methods. The h5gate Group methods are described below:

*3.5.1 h5gate Group make\_group*

Method “make\_group” of object h5gate Group creates a group inside the current group. It has the following signature:

g = pg.make\_group(qid, name, attrs, link, abort)

In the above line, “pg” signifies a parent group, (object of type h5gate Group). The returned object, stored in variable ‘g’ in the above, is also an “h5gate Group” object. Parameters in the function have the same meaning as those in function h5gate File make\_group. There is no “path” parameter (which was in the File object make\_group) because the location of the created group is always known. (The created group will be located inside the parent group used to invoke the method).

*3.5.2 h5gate Group make\_custom\_group*

Method “make\_custom\_group” of object h5gate Group creates a custom group, usually within the parent group. The function signature for “make\_custom\_group” is:

g = pg.make\_custom\_group(qid, name, path, attrs)

Return type and parameters are the same as for h5gate Group method “make\_group”. If path is not specified or is a relative path, the group will be created inside the parent group. If path is an absolute path, the group will be created at the specified location which can be anywhere in the hdf5 file.

*3.5.3 h5gate Group set\_dataset*

Method “set\_dataset” of object h5gate Group is used to create a dataset within the parent group. It has the following signature:

d = pg.set\_dataset(qid, value, name, attrs, dtype, compress)

The return value is an object of type h5gate Dataset. The parameters have the same meaning as those in function h5gate File set\_dataset. There is no “path” parameter because the created dataset is always located inside the parent group used to invoke the method.

*3.5.4 h5gate Group set\_custom\_dataset*

Method set\_custom\_dataset of object h5gate Group is used to create a custom dataset, usually within the parent group. It has the following signature:

d = pg.set\_custom\_dataset(qid, value, name, path, attrs, dtype, compress)

The return value is an object of type h5gate Dataset. The parameters have the same meaning as those in function h5gate Group set\_dataset. If path is not specified or is a relative path, the group will be created inside the parent group. If path is an absolute path, the group will be created at the specified location.

*3.5.5 set\_attr method*

Both the h5gate Group and h5gate Dataset objects have a method “set\_attr” which is used to set the value of an hdf5 attribute of the group or dataset. It has the following signature:

n.set\_attr(aid, value, custom)

In the above, “n” is an h5gate Node object (which is a h5gate Group or Dataset).

Parameters are:

*aid* – attribute id (name of the attribute).

*value* – value to store in the attribute.

*custom* – a logical value (default “False”) which indicates whether or not the attribute is a custom attribute (that is, not part of the file format specification). Setting a value of “True” when setting a custom attribute prevents a warning message from being displayed for the attribute when closing the file.

**3.6. nwb\_utils.py functions**

File nwb\_utils.py provides the following utility functions that are useful when using the API. The function signatures and doc strings are given below.

def load\_file(filename):

""" Load content of a file. Useful for setting metadata to

content of a text file"""

def add\_epoch\_ts(e, start\_time, stop\_time, name, ts):

""" Add timeseries\_X group to nwb epoch.

e - h5gate.Group containing epoch

start\_time - start time of epoch

stop\_time - stop time of epoch

name - name of <timeseries> group to be added to epoch

ts - timeseries to be added, either h5gate.Group object or

path to timeseries """

def add\_roi\_mask\_pixels(seg\_iface, image\_plane, name, desc, pixel\_list, weights, width, height, start\_time=0):

""" Adds an ROI to the module, with the ROI defined using a list of pixels.

Args:

\*image\_plane\* (text) name of imaging plane

\*name\* (text) name of ROI

\*desc\* (text) description of RO

\*pixel\_list\* (2D int array) array of [x,y] pixel values

\*weights\* (float array) array of pixel weights

\*width\* (int) width of reference image, in pixels

\*height\* (int) height of reference image, in pixels

\*start\_time\* (double) <ignore for now>

Returns:

\*nothing\*

"""

def add\_roi\_mask\_img(seg\_iface, image\_plane, name, desc, img, start\_time=0):

""" Adds an ROI to the module, with the ROI defined within a 2D image.

Args:

\*seg\_iface\* (h5gate Group object) ImageSegmentation folder

\*image\_plane\* (text) name of imaging plane

\*name\* (text) name of ROI

\*desc\* (text) description of ROI

\*img\* (2D float array) description of ROI in a pixel map (float[y][x])

\*start\_time\* <ignore for now>

Returns:

\*nothing\*

"""

def add\_reference\_image(seg\_iface, plane, name, img):

""" Add a reference image to the segmentation interface

Args:

\*seg\_iface\* Group folder having the segmentation interface

\*plane\* (text) name of imaging plane

\*name\* (text) name of reference image

\*img\* (byte array) raw pixel map of image, 8-bit grayscale

Returns:

\*nothing\*

"""

**3.6. Example API calls**

To create an NWB file using the Python API, each component in the file (hdf5 group or dataset) must be created using a call to either “make\_group” or “set\_dataset”. For a given API call, the main decisions to be made are:

1. Whether or not to use the h5gate File object methods (e.g. “f.make\_group”, “f.set\_dataset”) or the methods associated with a previously created group (e.g. “g.make\_group” or “g.set\_dataset”).
2. What name to specify if the group or dataset has a variable name.
3. Whether or not a path must be specified for the parent group or dataset.
4. The group or dataset to create based on the NWB File format specification and the data to be stored in the file.

Some examples of calls made for the different types of data stored in NWB files are given below.

*3.6.1. Setting metadata in the top-level general group*

For metadata that is in the top level of group general, a call to the h5gate File object is used. Examples are:

f.set\_dataset("lab", "*content to be stored...*")

f.set\_dataset("surgery", "*content to be stored...*")

For metadata that is in inside a subgroup of general, first a call to the h5gate File object is used to create the subgroup, then calls are made using the subgroup to create and store the data inside it. Examples are:

Subject metadata – (inside group “/general/subject”)

g = f.make\_group("subject")

g.set\_dataset("subject\_id", "*content to be stored...*")

g.set\_dataset("species", "*content to be stored...*")

Extracellular ephys metadata – (inside group “/general/extracellular\_ephys”)

g = f.make\_group("extracellular\_ephys")

g.set\_dataset("electrode\_group", "*content to be stored...*")

Extracellular ephys electrode group metadata

# create subgroup using the group “g” made by the previous call

g2 = g.make\_group("<electrode\_group\_N>", "*name of electrode group...*")

# set dataset inside subgropu

g2.set\_dataset("description", "*content to be stored...*")

*3.6.2. Creating Modules and interfaces*

NWB “Modules” (which are stored as hdf5 groups inside the “/processing” group) are created using the following call:

gm = f.make\_group("<module>", "*name of module.*")

Once a module is created, NWB interfaces (which are groups inside a module) are created by calling “make\_group” using the group associated with the module. Examples:

gi = gm.make\_group("Clustering")

In the above call, the “Clustering” interface is created inside of the module created in the previous call. Storing data inside the interface is done by calling the methods of the group associated with the interface. For example, the Clustering interface contains a dataset named “times”. Data would be saved to it using:

gi.set\_dataset("times", *data\_to\_be\_stored*)

In the above call, *data\_to\_be\_stored* would be a list, tuple, or numpy array containing the data to be stored.

*3.6.3. Storing TimeSeries data*

NWB time series are stored by first creating a group that specifies the type of TimeSeries being stored than using methods of that group to store the associated data. All time series groups have a variable name (that is, the name is specified by in the API call). If the TimeSeries being created is not inside a previously created h5gate Group, then the h5gate File object must be used to create the TimeSeries. Otherwise, the parent h5gate Group object is used. Some examples are below.

Storing TimeSeries in /acquisition/timeseries

ts = f.make\_group("<TimeSeries>", "*name of TimeSeries group*",

path="/acquisition/timeseries", attrs= {"source": "..."})

ts.set\_dataset("data", *data\_to\_be\_stored,*

*attrs={*"unit": "*unit used*"}, compress=True)

ts.set\_dataset("timestamps", *times,* compress=True)

ts.set\_dataset("num\_samples", *len(times*))

In the above, *times* is a list, tuple or numpy array having the data to be stored in the timestamps array. Argument “compress” is set to true to save space.

Storing TimeSeries type ImageSeries in /stimulus/timeseries

ts = f.make\_group("<ImageSeries>", "*name of group*",

path="/stimulus/timeseries", attrs= {"source": "..."})

# calls below are the same as in the previous example

ts.set\_dataset("data", *data\_to\_be\_stored,*

*attrs={*"unit": "*unit used*"}, compress=True)

ts.set\_dataset("timestamps", *times,* compress=True)

ts.set\_dataset("num\_samples", *len(times*))

Storing a TimeSeries type in an interface

Many interfaces include as a member one or more TimeSeries types. Modules and interfaces are stored using the following hierarchical layout:

/processing/<module>/interface/interface-contents

To store a TimeSeries inside an interface,the group associated with the interface is used to create the TimeSeries group. An example:

# create the module

gm = f.make\_group("<module>", "*name of module.*")

# create the interface

gi = gm.make\_group("DfOverF")

# create the TimeSeries group using the interface group

ts = gi.make\_group("<RoiResponseSeries>", "*name of TimeSeries group*")

# set the TimeSeries datasets using the time series group

ts.set\_dataset("data", *data\_to\_be\_stored,*

*attrs={*"unit": "*unit used*"}, compress=True)

ts.set\_dataset("timestamps", *times,* compress=True)

ts.set\_dataset("num\_samples", *len(times*))

*3.6.4 Storing epochs*

To store epochs, first create the top-level epoch group:

fe = f.make\_group("epochs")

Then create each epoch inside the top-level epoch group:

e = fe.make\_group("<epoch\_X>", epoch\_name)

e.set\_dataset('description', description)

e.set\_dataset('start\_time', start\_time)

e.set\_dataset('stop\_time', stop\_time)

e.set\_dataset('tags', tags)

To add time (“timeseries\_X”) to the epoch, use the nwb\_utility routine “add\_epoch\_ts”:

ut.add\_epoch\_ts(e, start\_time, stop\_time, "pole\_in", tsg)

In the above “tsg” specifies a previously created time series to add to the epoch. It can either be an h5gate Group object, or a string specifying the hdf5 path to a previously created time series.

**4. Using extensions**

The Specification Language used for the NWB format can be used to create extensions to the format. Creating extensions requires understanding the specification language, which is described more completely in another document. This section gives a summary of what extensions are and the API for using them.

The specification language that is used to define the NWB format is also used for extensions. The language is written using Python dictionaries in a syntax that is similar to JSON. Each top-level key to the dictionary is a string specifying a “namespace”. The value associated with the key is the specification for the namespace. The namespaces provide a way to group related specifications (into the same namespace) and avoid name conflicts in a manner similar to how namespaces are used with XML. Schematically, the structure of specification language namespace and definition are shown below:

{ "core": <definition for core namespace>,

"ext1": <definition for ext1 namespace>,

"ext2": <definition for ext2 namespace>, ... }

In the above, “core” designates the namespace associated with the core (standard) NWB format and ext1, ext2, … designate the namespaces associated with extensions. (The name of an extension does not need to start with ext; any valid python identifier can be used).

*Providing extensions to the API*

Extensions are provided to the API using additional parameters in the call to nwb\_file. The full parameter list and default values nwb\_file is given below:

def nwb\_file(fname, start\_time="", ddef={}, dimp=[], default\_ns='core', options={})

The parameters are:

fname - name of nwb (hdf5) file to create.

start\_time - session starting time. If not specified, current time is used.

ddef - supplied file format specification

dimp - Array of data definition files to import. Can be used to import the

core definitions and/or extensions. Has dictionaries of form:

{'file': <file\_name>, 'var': <variable\_name> }

where <file\_name> is name of .py file defining the structures and locations,

and <variable\_name> is the variable having the definitions in that file.

default\_ns - default name space for referencing data definition structures

options - specified options. See 'validate\_options' in h5gate.py

There are two ways to provide extensions to the API. The first method is via parameter ddef. To use this method, parameter ddef is set to a Python dictionary containing the namespaces and definitions illustrated above (e.g. {"core": <>, “ext1”: <>, ...} ).

The second method uses parameter “dimp” to load the extensions from one or more files. To use this method the dictionary (or dictionaries) are stored in one or more Python files, and the dictionary within each file is assigned to a Python variable. Parameter “dimp” is set to an array of dictionaries, as described in the parameter description above, which each element specifying a Python file to load and the variable containing the dictionary of specifications. This is the method used to load the NWB core definition. See code in file nwb\_file.py.

*Referencing extensions*

Groups or datasets that are defined in extensions are referenced by prefixing the name of the group or dataset with the namespace identifier. For example, given an extension with namespace “ext1” that has defines a dataset named “relative\_humidity”, the dataset is created using the following code:

f.set\_dataset("ext1:relative\_humidity", *value\_to\_store*)

The only difference between using extensions and the NWB core format, is that when referencing the extensions, the extension namespace followed by a colon (:) is included in front of the group or dataset name. This is also the case if the name is variable, e.g. ext1:<group\_x> would be used to reference a variable named group in namespace ext1.