NeuroneLoader Documentation

Release 1.0

Felix Heilmeyer

Sep 09, 2021

Concepts

NeuroneLoader is a python module for loading neurophysiological data recorded with Bittium NeurOne (formerly MegaEMG). It therefore allows using the data in pure python processing workflows using the python scientifc software stack (e.g. numpy) without the need of prior conversion using other (proprietary) software (e.g. MATLAB). It can also export it to container objects used by the popular python-mne framework.

Props to Andreas Henelius at Finnish Institute of Occupational Health for figuring out how to read the NeurOne binary format in pure python as part of his export2hdf project.

CHAPTER 1

Installation

```
pip install neurone_loader
```

If you want to export to python-mne you must also install MNE and all it's dependencies.

```
pip install mne
```

CHAPTER 2

Quick start

```
>>> from neurone_loader import Recording
>>> rec = Recording(path_to_recording_folder)
>>> rec.event_codes
array([  0,   1,  12,  13,  99, 128], dtype=int32)
```

Please note that because raw EEG recordings can be quite large this package is very memory aware. Most data will be loaded from disk lazily, i.e. the moment you're actually accessing it, and redundant data will be removed from memory as soon as it has been copied - unless you specify otherwise. Be advised that working with big recordings might still require a lot of memory.

I recommend looking at the docstrings before executing anything and maybe having a look at Concepts section in the Documentation before you start working with this package.

CHAPTER 3

Contributing

If you encounter any problem feel free to open a issue on GitHub. If you found a bug and want to supply a fix or if you want to contribute a new feature open a pull request. Just make sure that your code is not breaking any tests and you also supply tests for your code.

3.1 Testing

To run the tests you must first get the test data and then you can run the test with the following commands. Please run them in the repository directory, not in the test subdirectory.

To get the test data (~2.8GB) you need to install wget. Then you can download the data by running

```
bash test/get_test_data.sh
```

Then you can run the tests with

```
python -m unittest discover -s test -t .
```

3.1.1 Containers and data structure

NeurOne recordings consist of three structures:

1. A **recording** containing (multiple)
2. **sessions** containing (multiple)
3. **phases** containing the actual data

Accordingly `neurone_loader` provides three containers representing these structures.

[2]:
```
from neurone_loader import Recording, Session, Phase
```

In each of these you can access the `data` and metadata like `sampling_rate` or `channels` and all of them support the features described in Lazy loading.

`Session` and `Recording` technically don't hold data themselves. On the other hand one usually wants to work with the whole recording or at least a whole session of a recording. Therefore accessing the `.data` or any other attribute of `Session` or `Recording` will concatenate the data for you. Be aware that in order to save memory, accessing the `.data` attribute of a superseeding object will replace the `.data` attribute of the included containers with a view on the concatenated data. So if you manipulate any one of them you will also manipulate the other.

[3]:
```
rec = Recording(test_data_path)
session0 = rec.sessions[0]
sum_of_samples_of_phases_in_session0 = sum([p.n_samples for p in session0.phases])
print('\n')
print(f'session0.n_samples == sum_of_samples_of_phases_in_session0 is {session0.n_
↪samples == sum_of_samples_of_phases_in_session0}')
```

```
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples


session0.n_samples == sum_of_samples_of_phases_in_session0 is True
```

For a detailed description refer to the API documentation of the respective objects

3.1.2 Lazy loading

Because raw EEG recordings can be quite large this package is very aware of memory restrictions and possible bottlenecks due to long loading times from disk.

Therefore most actions that require loading data from disk into memory are executed lazily, meaning:

1. the data is loaded from disk when you access it for the first time
2. the data remains in memory and can be accessed very fast subsequently

To make working with the data more comfortable, the creation of containers and the loading of metadata on the other hand happens instantly.

[2]:
```
from neurone_loader import Recording
```

The following examples use the `Recording` container, but all of the features shown below also work with `Session` and `Phase`!

---

[3]:
```
# fast: only relevant metadata is loaded from disk
%time rec = Recording(test_data_path)
```

```
CPU times: user 15.5 ms, sys: 0 ns, total: 15.5 ms
Wall time: 14.7 ms
```

[4]:
```
%%time
# fast: metadata is already in memory
print(f'Sessions: {len(rec.sessions)}')
print(f'Sampling rate: {rec.sampling_rate}Hz')
```

```
Sessions: 2
Sampling rate: 5000Hz
CPU times: user 293 µs, sys: 0 ns, total: 293 µs
Wall time: 206 µs
```

[5]:
```
%%time
# this is slow: the session data needs to be retrieved from disk first
print(f'Session 1 shape: {rec.sessions[0].data.shape}')
```

```
(Lazy) loading Session.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
Session 1 shape: (2504369, 138)
CPU times: user 9.7 s, sys: 2.73 s, total: 12.4 s
Wall time: 4.05 s
```

[6]:
```
%%time
# this will be faster because the data is already in memory
print(f'Session 1 shape again: {rec.sessions[0].data.shape}')
```

```
Session 1 shape again: (2504369, 138)
CPU times: user 2.03 ms, sys: 528 µs, total: 2.55 ms
Wall time: 162 µs
```

As you can see above the container object can be contructed and used very memory and time efficient. Reading the actual session data, which can take a long time and may consume a lot of memory, is only happening when the data is actually needed. On subsequent calls the already loaded data is retrieved from memory which is much faster.

To save memory the data can be cleared from memory using the `.clear_data()` function.

[7]:
```
rec.clear_data()
```

Preloading

In some cases you may want to load all of the data from disk at once. There are two ways to achieve this.

1. Invoke the loading of all (not yet loaded) data by calling `.preload()`
2. Load all the data on initialization by setting the argument `preload=True`

[8]:
```
#Reload all the data cleared in [7]
rec.preload()
```

```
Preloading property data of <neurone_loader.loader.Recording object at 0x7f905d013898>
(Lazy) loading Recording.data
(Lazy) loading Session.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Session.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
```

(continues on next page)

(continued from previous page)

```
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
```

[9]:
```
# Load all of the data on initialization
not_so_lazy_rec = Recording(test_data_path, preload=True)
```

```
Preloading property data of <neurone_loader.loader.Recording object at 0x7f9063412ef0>
(Lazy) loading Recording.data
(Lazy) loading Session.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Session.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.events
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.n_channels
```

(continues on next page)

(continued from previous page)

```
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
(Lazy) loading Phase.n_channels
```

3.1.3 Exporting

All three containers `Recording`, `Session` and `Phase` implement the ability to export the data (including events) to a `mne.io.RawArray`. To do this simply call the .to_mne() function of any container. Be aware that this process involves copying the data in memory, so it might require (at least momentary) up to double the size of the data in memory space.

[2]:
```
from neurone_loader import Recording
```

[3]:
```
rec = Recording(test_data_path)
session0 = rec.sessions[0]
```

[4]:
```
try:
    cnt = session0.to_mne()
except AssertionError as e:
    print(f'Error: {e}')
```

```
(Lazy) loading Phase.events
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.events
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.events
(Lazy) loading Phase.n_samples
(Lazy) loading Phase.events
(Lazy) loading Phase.n_samples
(Lazy) loading Session.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
(Lazy) loading Phase.data
Error: events with event code 0 are not supported by MNE, use the substitute_zero_
↪events_with parameter of this method to substitute with an alternative code
```

The way MNE represents events in a stimulation channel prevents it from using events with the event code 0. In order to successfully convert you must either

- manipulate the `.events` data of the container to remove or substitute the events with Code == 0
- or use the `substitute_zero_events_with` argument to provide a alternative number for automatic substitution

Make sure though that the alternative event code you are using is not already present in the data!

[5]:
```
alt_code = 10
assert alt_code not in session0.event_codes
cnt = session0.to_mne(substitute_zero_events_with=alt_code)
```

```
Creating RawArray with float64 data, n_channels=138, n_times=2504369
    Range : 0 ... 2504368 =      0.000 ...   500.874 secs
Ready.
```

For more details refer to the API documentation of .to_mne()

CHAPTER 4

API

neurone_loader.loader	Provides classes to load, represent and export data recorded with the Bittium NeurOne device.
neurone_loader.mne_export	Provides the metaclass *MneExportable* that allows subclasses implementing all the metaclass's properties to be converted to a *mne.io.RawArray*.
neurone_loader.neurone	Contains functions for reading data recorded with a Bittium NeurOne device.
neurone_loader.lazy	Provides the *Lazy* decorator to construct properties that are evaluated only once and the *preloadable* decorator to enable optional preloading of all lazy properties on initialization.

4.1 neurone_loader.loader

4.1.1 Classes

neurone_loader.loader.BaseContainer()	A metaclass that provides properties for accessing data shared between all subclasses.
neurone_loader.loader.Phase(*args, **kwargs)	Represents one recording phase of one NeurOne session in one NeurOne Recording
neurone_loader.loader.Recording(*args, **kwargs)	Represents one NeurOne Recording and contains all of the recording's sessions
neurone_loader.loader.Session(*args, **kwargs)	Represents one session in one NeurOne Recording and contains all of the session's phases

Provides classes to load, represent and export data recorded with the Bittium NeurOne device.

class neurone_loader.loader.**BaseContainer**

A metaclass that provides properties for accessing data shared between all subclasses. I cannot be used itself as

it is not implementing all required methods of its abstract superclass.

channels

> **Note:** This property is a lazy property. For details see *lazy.Lazy*
>
> **Returns** ordered list of all channel names, read from the session protocol
>
> **Return type** list[str]

drop_channels(*channels_to_drop*)

Remove specified channels from loaded data. Dropped channels will be remembered and when data is cleared from memory and reloaded from disk the channels will get removed again. To get them back create a new object of this type to reload from disk.

> **Parameters** **channels_to_drop** (*list[str]*) – names of channels to drop

sampling_rate

> **Returns** the sampling rate, read from the session protocol
>
> **Return type** int

class neurone_loader.loader.**Phase**(**args*, ***kwargs*)

Represents one recording phase of one NeurOne session in one NeurOne Recording

> **Parameters**
>
> - **path** (*str*) – path to the recording *session* folder
> - **phase** (*dict*) – phase object from a session protocol

clear_data()

Remove loaded data from memory

data

> **Note:** This property is a lazy property. For details see *lazy.Lazy*
>
> **Returns** recorded data with shape (samples, channels) in µV
>
> **Return type** numpy.ndarray

drop_channels(*channels_to_drop*)

Remove specified channels from loaded data. Dropped channels will be remembered and when data is cleared from memory and reloaded from disk the channels will get removed again. To get them back create a new object of this type to reload from disk.

> **Parameters** **channels_to_drop** (*list[str]*) – names of channels to drop

event_codes

> **Returns** all event codes used in the data as int32 in an numpy.ndarray
>
> **Return type** numpy.ndarray

events

Note: This property is a lazy property. For details see *lazy.Lazy*

Returns recorded events with Revision, Type, SourcePort, ChannelNumber, Code, StartSampleIndex, StopSampleIndex, DescriptionLength, DescriptionOffset, DataLength, DataOffset, StartTime, StopTime

Return type pandas.DataFrame

n_channels

Returns the number of channels, read from the session protocol

Return type int

n_samples

Returns the number of channels, inferred from the binary recording's file size

Return type int

preload()

Use this function to call all properties constructed with *lazy.Lazy*. It can also be used to reload all lazy properties without deleting them first.

Example

```
>>> @preloadable
>>> class Test:
>>>     @Lazy
>>>     def lazy_attribute(self):
>>>         print('lazy function called')
>>>         return 'lazy return'
>>>
>>> test_object = Test(preload=False) # The lazy property is not evaluated on␣
↪initialization
>>> test_object.preload()
lazy function called
>>> print(test_object.lazy_attribute) # The stored attribute is returned
lazy return
>>> test_object.preload() # All properties are reloaded even though already␣
↪stored
lazy function called
```

class neurone_loader.loader.**Recording**(**args*, ***kwargs*)

Represents one NeurOne Recording and contains all of the recording's sessions

Parameters **path** (*str*) – path to the recording *recording* folder

channels

Returns the channels used in all sessions and makes sure they're equal

Returns ordered list of all channel names, read from the session protocols

Return type list[str]

clear_data()

Remove loaded data in all phases of all sessions from memory

data

Note: This property is a lazy property. For details see *lazy.Lazy*

Returns concatenated data of all phases of all sessions with shape (samples, channels) in µV

Return type numpy.ndarray

Warning: Calling this replaces the data attribute of the contained phases and sessions with a view on the concatenated data to save memory. Keep this in mind when manipulating the contained sessions or phases.

drop_channels(*channels_to_drop*)
Remove specified channels from loaded data. Dropped channels will be remembered and when data is cleared from memory and reloaded from disk the channels will get removed again. To get them back create a new object of this type to reload from disk.

Parameters **channels_to_drop** (*list[str]*) – names of channels to drop

event_codes

Returns all event codes used in the data as int32 in an numpy.ndarray

Return type numpy.ndarray

events

Returns concatenated events of all phases of all sessions with Revision, Type, SourcePort, ChannelNumber, Code, StartSampleIndex, StopSampleIndex, DescriptionLength, DescriptionOffset, DataLength, DataOffset, StartTime, StopTime

Return type pandas.DataFrame

n_channels
Returns the number of channels used in all phases and makes sure they're equal

Returns the number of channels, read from the session protocol

Return type int

n_samples

Returns sum of the number of samples, inferred from the binary recording's file size, of all phases of all sessions

Return type int

preload()
Use this function to call all properties constructed with *lazy.Lazy*. It can also be used to reload all lazy properties without deleting them first.

Example

```
>>> @preloadable
>>> class Test:
>>>     @Lazy
>>>     def lazy_attribute(self):
>>>         print('lazy function called')
>>>         return 'lazy return'
>>>
```

(continues on next page)

(continued from previous page)

```
>>> test_object = Test(preload=False) # The lazy property is not evaluated on␣
↪initialization
>>> test_object.preload()
lazy function called
>>> print(test_object.lazy_attribute) # The stored attribute is returned
lazy return
>>> test_object.preload() # All properties are reloaded even though already␣
↪stored
lazy function called
```

sampling_rate
Returns the sampling rate used in all sessions and makes sure they're all equal

Returns the sampling rate, read from the session protocols

Return type int

class neurone_loader.loader.**Session**(**args*, ***kwargs*)
Represents one session in one NeurOne Recording and contains all of the session's phases

Parameters **path** (*str*) – path to the recording *session* folder

clear_data()
Remove loaded data in all phases from memory

data

Note: This property is a lazy property. For details see *lazy.Lazy*

Warning: Calling this replaces the data attribute of the contained phases with a view on the concatenated data to save memory. Keep this in mind when manipulating the contained sessions.

Returns concatenated data of all phases with shape (samples, channels) in µV

Return type numpy.ndarray

drop_channels(*channels_to_drop*)
Remove specified channels from loaded data. Dropped channels will be remembered and when data is cleared from memory and reloaded from disk the channels will get removed again. To get them back create a new object of this type to reload from disk.

Parameters **channels_to_drop** (*list[str]*) – names of channels to drop

event_codes

Returns all event codes used in the data as int32 in an numpy.ndarray

Return type numpy.ndarray

events

Returns concatenated events of all phases with Revision, Type, SourcePort, ChannelNumber, Code, StartSampleIndex, StopSampleIndex, DescriptionLength, DescriptionOffset, DataLength, DataOffset, StartTime, StopTime

Return type pandas.DataFrame

n_channels
Returns the number of channels used in all phases and makes sure they're equal

Returns the number of channels, read from the session protocol

Return type int

n_samples

Returns sum of the number of samples, inferred from the binary recording's file size, of all phases

Return type int

preload()
Use this function to call all properties constructed with *lazy.Lazy*. It can also be used to reload all lazy properties without deleting them first.

Example

```
>>> @preloadable
>>> class Test:
>>>     @Lazy
>>>     def lazy_attribute(self):
>>>         print('lazy function called')
>>>         return 'lazy return'
>>>
>>> test_object = Test(preload=False) # The lazy property is not evaluated on␣
↪initialization
>>> test_object.preload()
lazy function called
>>> print(test_object.lazy_attribute) # The stored attribute is returned
lazy return
>>> test_object.preload() # All properties are reloaded even though already␣
↪stored
lazy function called
```

4.2 neurone_loader.mne_export

4.2.1 Classes

neurone_loader.mne_export.MneExportable	A metaclass that provides a function allowing objects that expose data, events, channels and sampling_rate properties to be converted to an mne.io.RawArray.

4.2.2 Exceptions

neurone_loader.mne_export.UnknownChannelException	Raised if data contains a channel name that is neither in a list of well-known channels nor in an (optional) list of user supplied channel name to channel type mappings.

Provides the metaclass *MneExportable* that allows subclasses implementing all the metaclass's properties to be converted to a *mne.io.RawArray*.

class neurone_loader.mne_export.**MneExportable**

A metaclass that provides a function allowing objects that expose data, events, channels and sampling_rate properties to be converted to an mne.io.RawArray.

channels
Abstract Property

Returns should contain the names of channels, matching the sequence used in the data property

Return type list[str]

clear_data()
Abstract Method

Should delete loaded data from memory

data
Abstract Property

Returns should contain data in (n_samples, n_channels) shape

Return type numpy.ndarray

events
Abstract Property

Returns should contain the events as a DataFrame, required fields are *StartSampleIndex*, *StopSampleIndex* and *Code*. Additional fields are ignored.

Return type pandas.DataFrame

sampling_rate
Abstract Property

Returns should contain the used sampling rate

Return type int

to_mne(*substitute_zero_events_with=None*, *copy=False*, *channel_type_mappings=None*)
Convert loaded data to a mne.io.RawArray

Parameters

- **substitute_zero_events_with** (*None or int*) – None. events with code = 0 are not supported by MNE, if this parameter is set, the event code 0 will be substituted with this parameter
- **copy** (*bool*) – False. If False (default), the original data will be removed from memory to save space while creating the mne.io.RawArray. If the data is needed again it must be reloaded from disk
- **channel_type_mappings** (*None or dict*) – Optional. You can provide a dictionary of channel name to type mappings. If the data contains any channel not in the list of well-known channel names and not in this mapping the conversion will raise UnknownChannelException. You can choose to map any unknown channel to one specific type e.g. {'#unknown': 'eeg'}. For a list of available types see the documentation of `mne.pick_types()`. This setting takes precedence over the built-in list of common channel names.

Returns the converted data

Return type mne.io.RawArray

Raises

- **ImportError** – if the mne package is not installed

- *UnknownChannelException* – if a unknown channel name is encountered (see channel_type_mappings parameter)

exception neurone_loader.mne_export.**UnknownChannelException**
Raised if data contains a channel name that is neither in a list of well-known channels nor in an (optional) list of user supplied channel name to channel type mappings.

4.3 neurone_loader.neurone

4.3.1 Functions

neurone_loader.neurone.get_n1_event_format()	Define the format for the events in a neurone recording.
neurone_loader.neurone.read_neurone_data(fpath)	Read the NeurOne signal data from a binary file.
neurone_loader.neurone.read_neurone_data_info(fpath)	Read the sample and channel count from a NeurOne signal binary file.
neurone_loader.neurone.read_neurone_events(fpath)	Read the NeurOne events from a binary file.
neurone_loader.neurone.read_neurone_protocol(fpath)	Read the measurement protocol from an XML file.

Contains functions for reading data recorded with a Bittium NeurOne device. This module currently supports reading of data and events.

neurone_loader.neurone.**get_n1_event_format**()
Define the format for the events in a neurone recording.

Arguments: None.

Returns:

- A Struct (from the construct library) describing the event format.

neurone_loader.neurone.**read_neurone_data**(*fpath*, *session_phase=1*, *protocol=None*)
Read the NeurOne signal data from a binary file.

Arguments:

- **fpath: the path to the directory holding the** NeurOne measurement (i.e., the directory Protocol.xml and Session.xml files.
- **session_phase:** The phase of the measurement. Currently only reading of the first phase (1) is supported.
- **protocol:** The dictionary obtained using the function read_neurone_protocol. This argument is optional and if not given, the protocol is automatically read.

Returns:

- A numpy ndarray with the data, where each columns stores the data for one channel.

neurone_loader.neurone.**read_neurone_data_info**(*fpath*, *session_phase=1*, *protocol=None*)
Read the sample and channel count from a NeurOne signal binary file.

Arguments:

- **fpath: the path to the directory holding the** NeurOne measurement (i.e., the directory Protocol.xml and Session.xml files.
- **session_phase:** The phase of the measurement. Currently only reading of the first phase (1) is supported.
- **protocol:** The dictionary obtained using the function read_neurone_protocol. This argument is optional and if not given, the protocol is automatically read.

Returns: Returns: - a named tuple containing (i) the number of channels

and (ii) the number of samples in the recording.

(n_samples, n_channels)

neurone_loader.neurone.**read_neurone_events**(*fpath*, *session_phase=1*, *sampling_rate=None*)

Read the NeurOne events from a binary file.

Arguments:

- **fpath: the path to the directory holding the** NeurOne measurement (i.e., the directory Protocol.xml and Session.xml files.
- **sampling_rate:** The sampling rate of the recording. This argument is optional and if not given, the protocol is automatically read.
- **session_phase:** The phase of the measurement. Currently only reading of the first phase (1) is supported.

Returns:

- A dict containing the events and the data type for the events.

{“events” : <numpy structured array with the events>, “events_dtype” : <array with the numpy dtype for the events>}

neurone_loader.neurone.**read_neurone_protocol**(*fpath*)

Read the measurement protocol from an XML file.

Arguments:

- **fpath: the path to the directory holding the** NeurOne measurement (i.e., the directory Protocol.xml and Session.xml files.

Returns:

- a dictionary containing (i) the names of the channels in the recording and (ii) meta information (recording start/stop times, sampling rate).

{“meta” : <dict with metadata>, “channels” : <array with channel names>}

4.4 neurone_loader.lazy

4.4.1 Classes

neurone_loader.lazy.Lazy([fget, fset, fdel, doc])	Return a lazy property attribute.

4.4.2 Functions

neurone_loader.lazy.preloadable(cls)	Use this as a decorator for a class that contains properties constructed with *lazy.Lazy*.

Provides the *Lazy* decorator to construct properties that are evaluated only once and the *preloadable* decorator to enable optional preloading of all lazy properties on initialization.

class neurone_loader.lazy.**Lazy**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

Return a lazy property attribute.

This decorator can be used exactly like the `property` function to turn a function into an attribute, the difference being the following: A function decorated with `property` is evaluated every time the attribute is accessed. A function decorated with *lazy.Lazy* is only evaluated once and the result is stored as a private attribute. Subsequently the private attribute is returned when the property constructed with *lazy.Lazy* is accessed. The lazy property can also be set manually or deleted, just like every other attribute. When the lazy attribute is deleted and then accessed again, the property function is called again and the result stored as a private attribute.

Example

```
>>> class Test:
>>>     @Lazy
>>>     def lazy_attribute(self):
>>>         print('lazy function called')
>>>         return 'lazy return'
>>>
>>>     @property
>>>     def property_attribute(self):
>>>         print('property function called')
>>>         return 'property return'
>>>
>>> test_object = Test()
>>> print(test_object.property_attribute)
property function called
property return
>>> print(test_object.property_attribute) # A property function is evaluated on␣
↪every call
property function called
property return
>>> print(test_object.lazy_attribute) # The lazy function is evaluated on first␣
↪call
lazy function called
lazy return
>>> print(test_object.lazy_attribute) # but not on subsequent calls
lazy return
>>> del test_object.lazy_attribute    # When deleted the attribute is reset and␣
↪the
>>> print(test_object.lazy_attribute) # function is evaluated again on next call
lazy function called
lazy return
```

See also:

Decorate your class with the `lazy.preloadable` attribute to enable optional preloading of all lazy attributes on initialization.

neurone_loader.lazy.**preloadable**(*cls*)

Use this as a decorator for a class that contains properties constructed with *lazy.Lazy*. A class decorated

like this can be initialized with `preload=True` to call every lazy property once and store it's return value. Optionally the `preload` function can be used to do the same. It can also be used to reload all lazy properties without deleting them first.

Example

```
>>> @preloadable
>>> class Test:
>>>     @Lazy
>>>     def lazy_attribute(self):
>>>         print('lazy function called')
>>>         return 'lazy return'
>>>
>>> test_object = Test(preload=True) # The lazy property is evaluated on␣
↪initialization
lazy function called
>>> print(test_object.lazy_attribute) # The stored attribute is returned
lazy return
>>> del test_object.lazy_attribute    # When deleted the attribute is reset and␣
↪the
>>> print(test_object.lazy_attribute) # function is evaluated again on next call
lazy function called
lazy return
>>> test_object.preload() # All properties are reloaded even though already stored
lazy function called
```

neurone_loader.loader	Provides classes to load, represent and export data recorded with the Bittium NeurOne device.
neurone_loader.mne_export	Provides the metaclass *MneExportable* that allows subclasses implementing all the metaclass's properties to be converted to a *mne.io.RawArray*.
neurone_loader.neurone	Contains functions for reading data recorded with a Bittium NeurOne device.
neurone_loader.lazy	Provides the *Lazy* decorator to construct properties that are evaluated only once and the *preloadable* decorator to enable optional preloading of all lazy properties on initialization.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

P

R

S

T

U