PS C:\Users\willi\GITHUB\TG\PYGDS> python .\man\_pygds.py

Help on module pygds:

NAME

pygds

DESCRIPTION

Usage

-----

The ``demo\_()`` functions in ``pygds.py`` are an example of how to use ``pygds.GDS``.

Basic usage:

.. code:: py

>>> import pygds

>>> d = pygds.GDS()

``pygds.GDS`` hides the API differences between ``g.USBamp``, ``g.HIamp`` and ``g.Nautilus``.

E.g. ``d.GetImpedance()`` calls the right function of::

GDS\_GUSBAMP\_GetImpedance()

GDS\_GHIAMP\_GetImpedance()

GDS\_GNAUTILUS\_GetImpedance()

Similarly the configuration names are unified.

E.g. Trigger means TriggerEnabled, TriggerLinesEnabled or DigitalIOs. See ``name\_maps``.

The device-specific names also work:

.. code:: py

>>> d.TriggerEnabled == d.Trigger

True

For one device, the configuration fields are members of the device object:

.. code:: py

>>> d.Trigger = True

>>> d.SetConfiguration()

For more devices, use the ``Configs`` list:

.. code:: py

>>> for c in d.Configs:

... c.Trigger = True

>>> d.SetConfiguration()

``pygds.configure\_demo()`` configures all available channels:

.. code:: py

>>> pygds.configure\_demo(d,testsignal=1)

>>> d.SetConfiguration()

To acquire a fixed number of samples, please use:

.. code:: py

>>> a = d.GetData(d.SamplingRate)

>>> a.shape[0] == d.SamplingRate

True

To acquire a dynamic number of samples, provide a function ``more(samples)``.

A ``pygds.Scope`` object can be used as ``more`` parameter of ``GetData()``.

When closing the scope Window acquisition stops.

.. code:: py

>>> scope = pygds.Scope(1/d.SamplingRate, title="Channels: %s", ylabel = u"U[\u03bcV]")

>>> a = d.GetData(d.SamplingRate//2,scope)

>>> del scope

>>> a.shape[1]>=d.N\_electrodes

True

Don't forget ``del scope`` before repeating this.

To remove a GDS object manually, do:

.. code:: py

>>> d.Close()

>>> del d

In the doctest samples, this is done to make the next test succeed.

For a session where only one GDS object is used, there is no need to do this.

CLASSES

builtins.Exception(builtins.BaseException)

GDSError

builtins.list(builtins.object)

ConnectedDevices

builtins.object

Scope

\_config\_wrap(\_ffi\_struct\_wrap)

GDS

class ConnectedDevices(builtins.list)

| Lists all connected devices in a list of type ``[(serial, devicetype, inuse)]``:

|

| .. code:: py

|

| >>> import pygds

| >>> cd = pygds.ConnectedDevices()

|

| This is used by the ``pygds.GDS`` constructor.

| Use it separately only if you don't want to instantiate a pygds.GDS object,

| but still want to find out which devices are connected.

|

| Method resolution order:

| ConnectedDevices

| builtins.list

| builtins.object

|

| Methods defined here:

|

| \_\_del\_\_(self)

|

| \_\_init\_\_(self, server\_ip='127.0.0.1')

| Initialize self. See help(type(self)) for accurate signature.

|

| find(self, wanted\_type, exclude\_serials=None)

| ConnectedDevices.

|

| Find a device by type.

|

| .. code:: py

|

| >>> import pygds

| >>> cd = pygds.ConnectedDevices()

| >>> hiamp = cd.find(pygds.DEVICE\_TYPE\_GHIAMP)

| >>> hiamp is None or len(hiamp.split('.'))>0

| True

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

|

| ----------------------------------------------------------------------

| Methods inherited from builtins.list:

|

| \_\_add\_\_(self, value, /)

| Return self+value.

|

| \_\_contains\_\_(self, key, /)

| Return key in self.

|

| \_\_delitem\_\_(self, key, /)

| Delete self[key].

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getitem\_\_(...)

| x.\_\_getitem\_\_(y) <==> x[y]

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_iadd\_\_(self, value, /)

| Implement self+=value.

|

| \_\_imul\_\_(self, value, /)

| Implement self\*=value.

|

| \_\_iter\_\_(self, /)

| Implement iter(self).

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_len\_\_(self, /)

| Return len(self).

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_mul\_\_(self, value, /)

| Return self\*value.n

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_reversed\_\_(...)

| L.\_\_reversed\_\_() -- return a reverse iterator over the list

|

| \_\_rmul\_\_(self, value, /)

| Return self\*value.

|

| \_\_setitem\_\_(self, key, value, /)

| Set self[key] to value.

|

| \_\_sizeof\_\_(...)

| L.\_\_sizeof\_\_() -- size of L in memory, in bytes

|

| append(...)

| L.append(object) -> None -- append object to end

|

| clear(...)

| L.clear() -> None -- remove all items from L

|

| copy(...)

| L.copy() -> list -- a shallow copy of L

|

| count(...)

| L.count(value) -> integer -- return number of occurrences of value

|

| extend(...)

| L.extend(iterable) -> None -- extend list by appending elements from the iterable

|

| index(...)

| L.index(value, [start, [stop]]) -> integer -- return first index of value.

| Raises ValueError if the value is not present.

|

| insert(...)

| L.insert(index, object) -- insert object before index

|

| pop(...)

| L.pop([index]) -> item -- remove and return item at index (default last).

| Raises IndexError if list is empty or index is out of range.

|

| remove(...)

| L.remove(value) -> None -- remove first occurrence of value.

| Raises ValueError if the value is not present.

|

| reverse(...)

| L.reverse() -- reverse \*IN PLACE\*

|

| sort(...)

| L.sort(key=None, reverse=False) -> None -- stable sort \*IN PLACE\*

|

| ----------------------------------------------------------------------

| Data and other attributes inherited from builtins.list:

|

| \_\_hash\_\_ = None

class GDS(\_config\_wrap)

| The ``pygds.GDS`` class initializes the connection to g.NEEDaccess server.

|

| The constructor

|

| - initializes the connection to the wanted device(s) and

| - fetches the configuration(s).

|

| \*gds\_device\*: can be

|

| - omitted (default)

| - the first letter of the serial

| - one of DEVICE\_TYPE\_GUSBAMP, DEVICE\_TYPE\_GHIAMP or DEVICE\_TYPE\_GNAUTILUS

| - a single serial

| - comma-separated serials

|

| \*exclude\_serials\*: a list or set of serials to ignore. Default: None.

|

| \*server\_ip\*: the IP address of the GDS server. Default: pygds.SERVER\_IP

|

| - The g.NEEDaccess server port is pygds.SERVER\_PORT and it is fixed.

|

| - The client by default is pygds.CLIENT\_IP and pygds.CLIENT\_PORT.

| For a remote ``server\_ip``, the local IP is automatically determined.

|

| Without parameters the localhost g.NEEDaccess server is used

| and the first available device is connected.

|

| For one device the configuration fields are members of the GDS object.

| For more devices, every configuration is an entry in the ``Configs`` member.

|

| g.USBamp config:

|

|

|

|

| g.HIamp config:

|

|

| g.Nautilus config:

|

|

|

| Note that some names are unified to work for all devices. See ``name\_maps``.

|

| Method resolution order:

| GDS

| \_config\_wrap

| \_ffi\_struct\_wrap

| builtins.object

|

| Methods defined here:

|

| Calibrate(self)

| GDS.

|

| ``Calibrate()`` wraps the C API's ``GDS\_XXX\_Calibrate()`` functions(),

| which calibrates the device.

|

| The return value of each device is a dict entry in the returned list.

| d.Calibrate()[0] is a dict with these keys:

|

| +---------------+-------------------------------------------------------------------------------+

| | ScalingFactor | Array holding single type values with scaling factor for each analog channel. |

| +---------------+-------------------------------------------------------------------------------+

| | Offset | Array holding single type values with offset for each analog channel. |

| +---------------+-------------------------------------------------------------------------------+

|

| Close(self)

| GDS.

|

| Closes the device.

|

| All GDS objects are removed automatically when exiting Python.

|

| To remove a GDS object manually, use::

|

| d.Close()

| del d

|

| GetAsyncDigitalIOs(self)

| GDS.

|

| ``GetAsyncDigitalIOs()`` wraps the g.USBamp ``GDS\_GUSBAMP\_GetAsyncDigitalIOs()``.

| g.USBamp only.

|

| The return value of each device is an entry in the returned list.

| ``d.GetAsyncDigitalIOs()[0]`` is a list of dicts, each with these keys:

|

| +---------------+--------------------------------------------------------------+

| | ChannelNumber | Integer value representing the digital channel number |

| +---------------+--------------------------------------------------------------+

| | Direction | String holding the digital channel direction (In=0 or Out=1) |

| +---------------+--------------------------------------------------------------+

| | Value | Current value of the digital channel (true or false) |

| +---------------+--------------------------------------------------------------+

|

| GetAvailableChannels(self, combine=True)

| GDS.

|

| ``GetAvailableChannels()`` wraps C API's ``GDS\_XXX\_GetAvailableChannels()``.

| The return value of each device is an entry in the returned list.

| ``d.GetAvailableChannels()[0]`` is a list of 0 or 1.

|

| This is called when instantiating a GDS object to initialize the ``N\_electrodes`` member.

| It is also called in ``SetConfiguration()`` to ignore the channels that are not available.

| And it is called in ``IndexAfter()`` and thus also in

| ``N\_ch\_calc()`` to get the channel index or the configured channel count.

| There should be no reason to call this directly.

|

| GetAvailableDigitalIOs(self)

| GDS.

|

| ``GetAvailableDigitalIOs()`` wraps the g.Nautilus ``GDS\_GNAUTILUS\_GetAvailableDigitalIOs()``.

| g.Nautilus only.

|

| The return value of each device is an entry in the returned list.

| ``d.GetAvailableDigitalIOs()[0]`` is a list of dicts, each with these keys:

|

| +---------------+--------------------------------------------------------------------------+

| | ChannelNumber | Unsigned integer representing the digital IO number |

| +---------------+--------------------------------------------------------------------------+

| | Direction | String representing the direction of the digital channel (In=0 or Out=1) |

| +---------------+--------------------------------------------------------------------------+

|

| GetBandpassFilters(self)

| GDS.

|

| ``GetBandpassFilters()`` wraps the C API's ``GDS\_XXX\_GetBandpassFilters()`` functions.

|

| In the returned list an entry per device is a list of dicts, with one dict for each filter.

| The dicts also contain the key ``BandpassFilterIndex`` to be used to set the filter.

|

| The fields per filter are:

|

| +----------------------+----------------------------------------------------------------------+

| | BandpassFilterIndex | Use this for the according channel field |

| +----------------------+----------------------------------------------------------------------+

| | SamplingRate | Double value holding the sampling rate for which the filter is valid |

| +----------------------+----------------------------------------------------------------------+

| | Order | Unsigned integer holding filter order |

| +----------------------+----------------------------------------------------------------------+

| | LowerCutoffFrequency | Double representing lower cutoff frequency of the filter |

| +----------------------+----------------------------------------------------------------------+

| | UpperCutoffFrequency | Double representing upper cutoff frequency of the filter |

| +----------------------+----------------------------------------------------------------------+

| | TypeId | Representing type of filter |

| +----------------------+----------------------------------------------------------------------+

|

| To choose a filter for the desired sampling rate, you can do this:

|

| .. code:: py

|

| >>> import pygds; d = pygds.GDS()

| >>> f\_s\_2 = sorted(d.GetSupportedSamplingRates()[0].items())[1] #512 or 500

| >>> d.SamplingRate, d.NumberOfScans = f\_s\_2

| >>> BP = [x for x in d.GetBandpassFilters()[0] if x['SamplingRate'] == d.SamplingRate]

| >>> for ch in d.Channels:

| ... ch.Acquire = True

| ... if BP:

| ... ch.BandpassFilterIndex = BP[0]['BandpassFilterIndex']

| >>> d.SetConfiguration()

| >>> d.GetData(d.SamplingRate).shape[0] == d.SamplingRate

| True

| >>> d.Close(); del d

|

| GetChannelNames(self)

| GDS.

|

| ``GetChannelNames()`` wraps C API's ``GDS\_GNAUTILUS\_GetChannelNames()``.

|

| A list of channel names for each g.Nautilus device is an entry in the returned list.

| g.Nautilus only.

|

| ``d.GetChannelNames()[0]`` is a list of strings.

| The strings correspond to the labels on the electrodes.

|

| GetConfiguration(self)

| GDS.

|

| ``GetConfiguration()`` fetches the configuration from the device.

| This is done automatically when instantiating a GDS object.

|

| GetData(self, scanCount, more=None)

| GDS.

|

| ``GetData()`` gets the data from the device.

|

| GetData allocates ``scanCount\*N\_ch\*4`` memory two times.

| It fills one copy in a separate thread with sample data from the device,

| while the other copy is processed by the ``more`` function in the current thread.

| Then it swaps the two buffers.

|

|

| ``more`` must copy the samples to reuse them later.

|

| .. code:: py

|

| >>> import pygds; d = pygds.GDS()

| >>> samples = []

| >>> more = lambda s: samples.append(s.copy()) or len(samples)<2

| >>> data=d.GetData(d.SamplingRate, more)

| >>> len(samples)

| 2

| >>> d.Close(); del d

|

| GetDataInfo(self, scanCount)

| GDS.

|

| ``GetDatatInfo()`` returns (channelsPerDevice, bufferSizeInSamples).

|

| \*channelsPerDevice\* is a list of channels for each device.

|

| \*bufferSizeInSamples\* is the total number of samples.

|

| .. code:: py

|

| >>> import pygds

| >>> d = pygds.GDS()

| >>> scanCount = 500

| >>> channelsPerDevice, bufferSizeInSamples = d.GetDataInfo(scanCount)

| >>> sum(channelsPerDevice)\*scanCount == bufferSizeInSamples

| True

| >>> d.Close(); del d

|

| GetDeviceInformation(self)

| GDS.

|

| ``GetDeviceInformation()`` wraps the C API's ``GDS\_XXX\_GetDeviceInformation()`` functions.

|

| The device information for each device is a string entry in the returned list.

|

| GetFactoryScaling(self)

| GDS.

|

| ``GetFactoryScaling()`` wraps C API's ``GDS\_GHIAMP\_GetFactoryScaling()``.

|

| The factory scaling is an entry for each g.HIamp in the returned list.

| Only g.HIamp.

|

| GetImpedance(self, active=None)

| GDS.

|

| ``GetImpedance()`` wraps the C API's ``GDS\_XXX\_GetImpedance()`` functions.

|

| Gets the impedances for all channels of all devices.

| The impedances of each device are a list entry in the returned list.

|

| Note, that for g.Nautilus electrode 15 = Cz must be connected to GND,

| else an exception occurs.

|

| .. code:: py

|

| >>> import pygds; d = pygds.GDS()

| >>> imps = d.GetImpedance([1]\*len(d.Channels))

| >>> len(imps[0])==len(d.Channels)

| True

| >>> d.Close(); del d

|

| GetNetworkChannel(self)

| GDS.

|

| ``GetNetworkChannel()`` wraps the C API's ``GDS\_GNAUTILUS\_GetNetworkChannel()``.

|

| The currently used g.Nautilus network channel is an entry in the returned list.

|

| GetNotchFilters(self)

| GDS.

|

| ``GetNotchFilters()`` wraps the C API's ``GDS\_XXX\_GetNotchFilters()`` functions.

|

| In the returned list an entry per device is a list of dicts, with one dict for each filter.

| The dicts also contain the key ``NotchFilterIndex`` to be used to set the filter.

|

| The fields per filter are:

|

| +----------------------+----------------------------------------------------------------------+

| | NotchFilterIndex | Use this for the according channel field |

| +----------------------+----------------------------------------------------------------------+

| | SamplingRate | Double value holding the sampling rate for which the filter is valid |

| +----------------------+----------------------------------------------------------------------+

| | Order | Unsigned integer holding filter order |

| +----------------------+----------------------------------------------------------------------+

| | LowerCutoffFrequency | Double representing lower cutoff frequency of the filter |

| +----------------------+----------------------------------------------------------------------+

| | UpperCutoffFrequency | Double representing upper cutoff frequency of the filter |

| +----------------------+----------------------------------------------------------------------+

| | TypeId | Representing type of filter |

| +----------------------+----------------------------------------------------------------------+

|

| To choose a filter for the desired sampling rate you can do this:

|

| .. code:: py

|

| >>> import pygds; d = pygds.GDS()

| >>> f\_s\_2 = sorted(d.GetSupportedSamplingRates()[0].items())[1] #512 or 500

| >>> d.SamplingRate, d.NumberOfScans = f\_s\_2

| >>> N = [x for x in d.GetNotchFilters()[0] if x['SamplingRate'] == d.SamplingRate]

| >>> for ch in d.Channels:

| ... ch.Acquire = True

| ... if N:

| ... ch.NotchFilterIndex = N[0]['NotchFilterIndex']

| >>> d.SetConfiguration()

| >>> d.GetData(d.SamplingRate).shape[0] == d.SamplingRate

| True

| >>> d.Close(); del d

|

| GetScaling(self)

| GDS.

|

| ``GetScaling()`` wraps the C API's ``GDS\_XXX\_GetScaling()`` functions.

|

| The return value of each device is a dict entry in the returned list.

| Each dict has the fields:

|

| +--------+-------------------------------------------------------------------------------+

| | Factor | Array holding single type values with scaling factor for each analog channel. |

| +--------+-------------------------------------------------------------------------------+

| | Offset | Array holding single type values with offset for each analog channel. |

| +--------+-------------------------------------------------------------------------------+

|

| GetSupportedInputSources(self)

| GDS.

|

| ``GetSupportedInputSources()`` function wraps ``GDS\_GNAUTILUS\_GetSupportedInputSources()``.

|

| g.Nautilus only.

|

| ``d.GetSupportedInputSources()[0]`` is a list of

| integers corresponding to the ``pygds.GDS\_GNAUTILUS\_INPUT\_XXX`` constants.

| Each integer can be used for ``d.InputSignal``.

|

| GetSupportedNetworkChannels(self)

| GDS.

|

| ``GetSupportedNetworkChannels()`` wraps C API's ``GDS\_GNAUTILUS\_GetSupportedNetworkChannels()``.

|

| The supported network channels for each g.Nautilus device are an entry in the returned list.

| g.Nautilus only.

|

| ``GetSupportedNetworkChannels()[0]`` is a list of integers.

| Each integer can be used in ``d.SetNetworkChannel()``.

|

| GetSupportedSamplingRates(self)

| GDS.

|

|

| For each device a dict ``{SamplingRate:NumberOfScans}`` is an entry in the returned list.

|

| You can do ``d.NumberOfScans=d.GetSupportedSamplingRates()[0][d.SamplingRate]``

| to set the recommended NumberOfScans.

| This is done when using ``d.NumberOfScans\_calc()``, and if there are more devices per GDS object.

|

| GetSupportedSensitivities(self)

| GDS.

|

| ``GetSupportedSensitivities()`` wraps the C API's ``GDS\_GNAUTILUS\_GetSupportedSensitivities()``.

|

| The supported sensitivities for each g.Nautilus device are an entry in the returned list.

| g.Nautilus only.

|

| ``d.GetSupportedSensitivities()[0]`` is a list of integers.

| Each integer can be used as the channel's Sensitivity.

|

| IndexAfter(self, chname='')

| GDS.

|

| Get the channel 0-based index one position after the 1-based ``chname``.

| ``chname`` can also be one of::

|

| Counter

| Trigger

|

| and for g.Nautilus also::

|

| AccelerationData

| LinkQualityInformation

| BatteryLevel

| DigitalIOs

| ValidationIndicator

|

| Without ``chname`` it gives the count of configured channels.

|

| For more devices per GDS object one can use::

|

| name+serial, e.g. 1UB-2008.07.01

|

| to get the index of a channel of a specific device.

|

| .. code:: py

|

| >>> import pygds; d = pygds.GDS()

| >>> d.IndexAfter('4'+d.Name)

| 4

| >>> d.IndexAfter('4')

| 4

| >>> d.IndexAfter('AccelerationData')>=0

| True

| >>> d.IndexAfter('Counter')>=0

| True

| >>> d.IndexAfter('LinkQualityInformation')>=0

| True

| >>> d.IndexAfter('BatteryLevel')>=0

| True

| >>> d.IndexAfter('DigitalIOs')>=0

| True

| >>> d.IndexAfter('Trigger')>=0

| True

| >>> d.IndexAfter('ValidationIndicator')>=0

| True

| >>> d.IndexAfter('')==d.N\_ch\_calc()

| True

| >>> d.Close(); del d

|

| N\_ch\_calc(self)

| GDS.

|

| ``N\_ch\_calc()`` returns the number of configured channels.

| After the first call, you can use ``d.N\_ch`` to get the number of configured channels.

|

| .. code:: py

|

| >>> import pygds; d = pygds.GDS()

| >>> n = d.N\_ch\_calc()

| >>> d.N\_ch == n

| True

| >>> d.Close(); del d

|

| ``d.N\_electrodes`` is the number of electrodes in the GDS connection for all connected devices.

|

| NumberOfScans\_calc(self)

| GDS.

|

| Sets ``d.NumberOfScans`` by mapping ``d.SamplingRate`` via ``GetSupportedSamplingRates()``.

|

| ResetScaling(self)

| GDS.

|

| ``ResetScaling()`` wraps the g.Nautilus ``GDS\_GNAUTILUS\_ResetScaling()`` function.

|

| The scaling is reset to Offset=0.0 and Factor=1.0. g.Nautilus only.

|

| SetAsyncDigitalOutputs(self, outputs)

| GDS.

|

| ``SetAsyncDigitalOutputs()`` wraps the g.USBamp ``GDS\_GUSBAMP\_SetAsyncDigitalOutputs()``.

| g.USBamp only.

|

| SetConfiguration(self)

| GDS.

|

| ``SetConfiguration()`` needs to be called to send the configuration to the device.

|

|

| SetNetworkChannel(self, networkchannels)

| GDS.

|

| ``SetNetworkChannel()`` wraps the C API's ``GDS\_GNAUTILUS\_SetNetworkChannel()``.

| g.Nautilus only.

|

| ``SetNetworkChannel()`` sets the g.Nautilus network channel.

|

| \*networkchannels\* is one of the integers returned by GetSupportedNetworkChannels().

|

| SetScaling(self, scaling)

| GDS.

|

| ``SetScaling()`` wraps the C API's ``GDS\_XXX\_SetScaling()`` functions.

|

| ``SetScaling()`` sets the scaling on the device.

|

| \_\_del\_\_(self)

|

| \_\_init\_\_(self, gds\_device=None, exclude\_serials=None, server\_ip='127.0.0.1')

| Initialize self. See help(type(self)) for accurate signature.

|

| \_\_str\_\_(self)

| Return str(self).

|

| ----------------------------------------------------------------------

| Methods inherited from \_ffi\_struct\_wrap:

|

| \_\_delattr\_\_(self, key)

| Implement delattr(self, name).

|

| \_\_getattribute\_\_(self, key)

| Return getattr(self, name).

|

| \_\_repr\_\_(self)

| Return repr(self).

|

| \_\_setattr\_\_(self, key, value)

| Implement setattr(self, name, value).

|

| ----------------------------------------------------------------------

| Data descriptors inherited from \_ffi\_struct\_wrap:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

class GDSError(builtins.Exception)

| This is the exception that is raised in case of a g.NEEDaccess API error.

|

| Method resolution order:

| GDSError

| builtins.Exception

| builtins.BaseException

| builtins.object

|

| Methods defined here:

|

| \_\_init\_\_(self, message=None)

| Initialize self. See help(type(self)) for accurate signature.

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

|

| ----------------------------------------------------------------------

| Methods inherited from builtins.Exception:

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| ----------------------------------------------------------------------

| Methods inherited from builtins.BaseException:

|

| \_\_delattr\_\_(self, name, /)

| Implement delattr(self, name).

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_reduce\_\_(...)

| helper for pickle

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_setattr\_\_(self, name, value, /)

| Implement setattr(self, name, value).

|

| \_\_setstate\_\_(...)

|

| \_\_str\_\_(self, /)

| Return str(self).

|

| with\_traceback(...)

| Exception.with\_traceback(tb) --

| set self.\_\_traceback\_\_ to tb and return self.

|

| ----------------------------------------------------------------------

| Data descriptors inherited from builtins.BaseException:

|

| \_\_cause\_\_

| exception cause

|

| \_\_context\_\_

| exception context

|

| \_\_dict\_\_

|

| \_\_suppress\_context\_\_

|

| \_\_traceback\_\_

|

| args

class Scope(builtins.object)

| ``Scope`` makes a live update of a Matplotlib diagram and thus simulates an oscilloscope:

|

| .. code:: py

|

| >>> import numpy as np

| >>> import matplotlib.pyplot as plt

| >>> from pygds import Scope

| >>> import time

| >>> f = 10

| >>> scope=Scope(1/f)

| >>> t = np.linspace(0,100,100)/f

| >>> scope(np.array([np.sin(t+i/2) for i in range(10)]))

| True

| >>> time.sleep(0.1)

| >>> scope(np.array([np.sin(t+i/3) for i in range(10)]))

| True

| >>> time.sleep(0.1)

| >>> scope(np.array([np.sin(t+i/4) for i in range(10)]))

| True

| >>> time.sleep(0.1)

| >>> scope(np.array([np.sin(t+i/5) for i in range(10)]))

| True

| >>> del scope

|

| ``Scope`` can be used as the ``more`` argument of ``GetData()`` to have a live view on the data.

|

| To use ``Scope`` as a regular diagram, set ``modal=True``.

|

| The object's ``\_\_call\_\_(self,scan)`` displays the scans.

| On the first call to the object (via ``\_\_call\_\_()``), the diagram is initialized.

|

| Methods defined here:

|

| \_\_call\_\_(self, scans)

| Call self as a function.

|

| \_\_del\_\_(self)

|

| \_\_init\_\_(self, time\_factor, modal=False, subplots=None, \*\*subplot\_kw)

| Initialize self. See help(type(self)) for accurate signature.

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

FUNCTIONS

Initialize(gds\_headers=['C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI.h', 'C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI\_gHIamp.h', 'C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI\_gNautilus.h', 'C:/Program Files/gtec/gNEEDaccess Client API/C/GDSClientAPI\_gUSBamp.h'], gds\_dll=None)

Initializes pygds. This is done automatically at ``import pygds``.

If the GDS service is running, then GDSClientAPI.dll is used, else GDSServer.dll.

To manually change, first call Uninitialize(), then e.g. Initialize(gds\_dll="GDSServer.dll").

``pygds.Initialize()``

- populates the pygds namespace with definitions from the GDS headers. The ``GDS\_`` prefix is dropped

- loads the GDS client DLL

- calls ``GDS\_Initialize()``

If g.NEEDaccess is installed in a non-standard location, then ``pygds.Initialize()`` will fail.

Then you need to call ``pygds.Initialize()`` manually and

provide the header file paths and the DLL path as parameters.

The return value is True if initialization succeeded.

Uninitialize()

Clean up is done automatically when exiting Python.

``Uninitialize()`` tries not to block, by taking into account these GDS API behaviors:

- ``GDS\_Uninitialized()`` blocks, if called after calling ``GDS\_Disconnect()`` on all connections.

- On the other hand, to prevent a freeze, one must call ``GDS\_Uninitialized()``, if no device was ever connected, but ``GDS\_Initialize()`` had been called.

configure\_demo(d, testsignal=False, acquire=1)

Makes a configuration for the demos.

The device configuration fields are members of the device object d.

If d.ConfigCount>1, i.e. more devices are connected, use d.Configs[i] instead.

Config names are unified: See ``name\_maps``.

This does not configure a filter.

Note that g.HIamp version < 1.0.9 will have wrong first value without filters.

demo\_all()

Runs all demos.

demo\_all\_api()

This demo calls all wrapped API functions.

It can be used as a regression test.

Have a device

- connected to the PC and

- switched on

demo\_counter()

This demo

- configures to internal test signal

- records 1 second

- displays the counter

- displays channel 2

Have a device

- connected to the PC and

- switched on

demo\_di()

This demo

- records the DI channel

- displays it with the live scope

Have a device

- connected to the PC and

- switched on

demo\_filter()

This demo demonstrates the use of filters.

Have a device

- connected to the PC and

- switched on

demo\_impedance()

This demo demonstrates impedance measurement.

Have a device

- connected to the PC and

- switched on

- for g.Nautilus Cz must be connected to GND

demo\_remote()

This demo shows how to connect a remote PC.

Have a device

- connected to the PC and

- switched on

demo\_save()

This demo

- records the internal test signal

- saves the acquired data after recording

Have a device

- connected to the PC and

- switched on

demo\_scaling()

This demo tests the function GetScaling.

Have a device

- connected to the PC and

- switched on

demo\_scope()

This demo

- records a test signal

- displays it in the live scope

Have a device

- connected to the PC and

- switched on

demo\_scope\_all()

This demo

- records a test signal for all channels with maximum sampling rate

- displays it in the live scope

Have a device

- connected to the PC and

- switched on

demo\_usbamp\_sync()

This demo

- configures two g.USBamp with the sinus test signal

- records all 32 channels of the two synchronized g.USBamp and

- displays all 32 channels in the time scope.

Have two switched on g.USBamp devices

- connected to the PC and

- connected with each other via the synch cables

main()

DATA

CHANNEL\_DIRECTION = <ctype 'GDS\_CHANNEL\_DIRECTION'>

CHANNEL\_DIRECTION\_IN = 0

CHANNEL\_DIRECTION\_OUT = 1

CLIENT\_IP = '127.0.0.1'

CLIENT\_PORT = 50224

DEVICE\_NAME\_LENGTH\_MAX = 32

DEVICE\_TYPE = <ctype 'GDS\_DEVICE\_TYPE'>

DEVICE\_TYPE\_GHIAMP = 2

DEVICE\_TYPE\_GNAUTILUS = 3

DEVICE\_TYPE\_GUSBAMP = 1

DEVICE\_TYPE\_NOT\_SUPPORTED = 0

ERROR\_ACCESS\_DENIED = 5

ERROR\_ACQUISITION\_BUFFER\_OVERFLOW = 16

ERROR\_ACQUISITION\_DIED = 18

ERROR\_BUFFER\_TOO\_SMALL = 15

ERROR\_DEVICE\_ALREADY\_IN\_USE = 9

ERROR\_HANDLE\_ALREADY\_ASSOCIATED = 4

ERROR\_HANDLE\_NOT\_ASSOCIATED = 3

ERROR\_INCOMPATIBLE\_CONFIGURATIONS = 11

ERROR\_INVALID\_ACQUISITION\_STATE = 13

ERROR\_INVALID\_CONFIGURATION = 10

ERROR\_INVALID\_DEVICE = 7

ERROR\_INVALID\_DEVICE\_LIST = 6

ERROR\_INVALID\_ENDPOINT = 20

ERROR\_INVALID\_HANDLE = 2

ERROR\_INVALID\_PARAMETER = 1

ERROR\_INVALID\_REQUEST\_STATUS = 21

ERROR\_INVALID\_STREAMING\_STATE = 14

ERROR\_MESSAGE\_LENGTH\_MAX = 1024

ERROR\_OPEN\_DEVICE = 8

ERROR\_READ\_CONFIGURATION = 12

ERROR\_REQUEST\_TIMED\_OUT = 22

ERROR\_SERIALIZATION = 19

ERROR\_SUCCESS = 0

ERROR\_TRANSMISSION\_BUFFER\_OVERFLOW = 17

ERROR\_UNKNOWN = 4294967295

FALSE = 0

FILTER\_TYPE = <ctype 'GDS\_FILTER\_TYPE'>

FILTER\_TYPE\_BESSEL = 2

FILTER\_TYPE\_BUTTERWORTH = 1

FILTER\_TYPE\_CHEBYSHEV = 0

GDSCLIENTAPI\_API = ''

GHIAMP\_CHANNELS\_MAX = 256

GHIAMP\_DEVICE\_INFORMATION\_LENGTH\_MAX = 65536

GNAUTILUS\_ADDITIONAL\_CHANNELS\_MAX = 8

GNAUTILUS\_CHANNELS\_MAX = 64

GNAUTILUS\_CHANNELS\_PER\_MODULE\_MAX = 8

GNAUTILUS\_DEVICE\_INFORMATION\_LENGTH\_MAX = 1024

GNAUTILUS\_DIGITAL\_IO\_CHANNELS\_MAX = 8

GNAUTILUS\_ELECTRODE\_NAME\_LENGTH\_MAX = 4

GNAUTILUS\_INPUT\_SIGNAL = <ctype 'GDS\_GNAUTILUS\_INPUT\_SIGNAL'>

GNAUTILUS\_INPUT\_SIGNAL\_ELECTRODE = 0

GNAUTILUS\_INPUT\_SIGNAL\_MVDD = 2

GNAUTILUS\_INPUT\_SIGNAL\_SHORTED = 1

GNAUTILUS\_INPUT\_SIGNAL\_TEST\_SIGNAL = 5

GNAUTILUS\_INPUT\_TEMPERATURE = 4

GNAUTILUS\_MODULES\_MAX = 8

GNAUTIUS\_INPUT\_SIGNAL\_DRL = 3

GNAUTIUS\_INPUT\_SIGNAL\_DRL\_CHIP = 7

GNAUTIUS\_INPUT\_SIGNAL\_SHORTCUT\_CHIP = 6

GNAUTIUS\_INPUT\_SIGNAL\_TEST\_SIGNAL\_CHIP = 8

GUSBAMP\_ASYNC\_DIGITAL\_IO\_CHANNELS\_MAX = 4

GUSBAMP\_CHANNELS\_MAX = 16

GUSBAMP\_DEVICE\_INFORMATION\_LENGTH\_MAX = 256

GUSBAMP\_GROUPS\_MAX = 4

GUSBAMP\_WAVESHAPE = <ctype 'GDS\_GUSBAMP\_WAVESHAPE'>

GUSBAMP\_WAVESHAPE\_DRL = 3

GUSBAMP\_WAVESHAPE\_NOISE = 4

GUSBAMP\_WAVESHAPE\_SAWTOOTH = 1

GUSBAMP\_WAVESHAPE\_SINE = 2

GUSBAMP\_WAVESHAPE\_SQUARE = 0

IP\_ADDRESS\_LENGTH\_MAX = 16

MaxEmptyDataLoops = 1000

NULL = <cdata 'void \*' NULL>

OpenDevices = {}

SERVER\_IP = '127.0.0.1'

SERVER\_PORT = 50223

absolute\_import = \_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0...

division = \_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192...

gNEEDaccessHeaders = ['C:/Program Files/gtec/gNEEDaccess Client API/C/...

g\_gds\_dll = 'GDSClientAPI.dll'

gds\_dll\_client = 'GDSClientAPI.dll'

gds\_dll\_standalone = 'GDSServer.dll'

generators = \_Feature((2, 2, 0, 'alpha', 1), (2, 3, 0, 'final', 0), 0)

name\_maps = {'GDS\_GHIAMP\_CHANNEL\_CONFIGURATION': {'BipolarChannel': 'R...

nested\_scopes = \_Feature((2, 1, 0, 'beta', 1), (2, 2, 0, 'alpha', 0), ...

print\_function = \_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0)...

with\_statement = \_Feature((2, 5, 0, 'alpha', 1), (2, 6, 0, 'alpha', 0)...