PyTango Documentation

Release 9.2.0

PyTango team

January 30, 2017

Contents

CHAPTER 1

Getting started

1.1 Installing

1.1.1 Linux

PyTango is available on linux as an official debian/ubuntu package:

```
$ sudo apt-get install python-pytango
```

RPM packages are also available for RHEL & CentOS:

- CentOS 6 32bits
- CentOS 6 64bits
- CentOS 7 64bits
- Fedora 23 32bits
- Fedora 23 64bits

1.1.2 PyPi

You can also install the latest version from PyPi.

First, make sure you have the following packages already installed (all of them are available from the major official distribution repositories):

- boost-python (including boost-python-dev)
- numpy

Then install PyTango either from pip:

```
$ pip install PyTango
```

or easy_install:

```
$ easy_install -U PyTango
```

1.1.3 Windows

First, make sure Python and numpy are installed.

PyTango team provides a limited set of binary PyTango distributables for Windows XP/Vista/7/8. The complete list of binaries can be downloaded from PyPI.

Select the proper windows package, download it and finally execute the installion wizard.

1.2 Compiling

1.2.1 Linux

Since PyTango 9 the build system used to compile PyTango is the standard python setuptools.

Besides the binaries for the three dependencies mentioned above, you also need the development files for the respective libraries.

You can get the latest `.tar.gz` from PyPI or directly the latest SVN checkout:

```
$ git clone https://github.com/tango-cs/pytango.git
$ cd pytango
$ python setup.py build
$ sudo python setup.py install
```

This will install PyTango in the system python installation directory. (Since PyTango9, *ITango* has been removed to a separate project and it will not be installed with PyTango.) If you whish to install in a different directory, replace the last line with:

```
$ # private installation to your user (usually ~/.local/lib/python<X>.<Y>/site-packages)
$ python setup.py install --user

$ # or specific installation directory
$ python setup.py install --prefix=/home/homer/local
```

1.2.2 Windows

On windows, PyTango must be built using MS VC++. Since it is rarely needed and the instructions are so complicated, I have choosen to place the how-to in a separate text file. You can find it in the source package under `doc/windows_notes.txt`.

1.3 Testing

To test the installation, import `tango` and check `tango.Release.version`:

```
$ python -c "import tango; print(tango.Release.version)"
9.2.0
```

Next steps: Check out the *Quick tour*.

CHAPTER 2

Quick tour

This quick tour will guide you through the first steps on using PyTango.

2.1 Fundamental TANGO concepts

Before you begin there are some fundamental TANGO concepts you should be aware of.

Tango consists basically of a set of *devices* running somewhere on the network.

A device is identified by a unique case insensitive name in the format <*domain*>/<*family*>/<*member*>. Examples: *LAB-01/PowerSupply/01*, *ID21/OpticsHutch/energy*.

Each device has a series of *attributes*, *pipes*, *properties* and *commands*.

An attribute is identified by a name in a device. It has a value that can be read. Some attributes can also be changed (read-write attributes). Each attribute has a well known, fixed data type.

A pipe is a kind of attribute. Unlike attributes, the pipe data type is strucured (in the sence of C struct) and it is dynamic.

A property is identified by a name in a device. Usually, devices properties are used to provide a way to configure a device.

A command is also identified by a name. A command may or not receive a parameter and may or not return a value when it is executed.

Any device has **at least** a *State* and *Status* attributes and *State*, *Status* and *Init* commands. Reading the *State* or *Status* attributes has the same effect as executing the *State* or *Status* commands.

Each device as an associated *TANGO Class*. Most of the times the TANGO class has the same name as the object oriented programming class which implements it but that is not mandatory.

TANGO devices *live* inside a operating system process called *TANGO Device Server*. This server acts as a container of devices. A device server can host multiple devices of multiple TANGO classes. Devices are, therefore, only accessible when the corresponding TANGO Device Server is running.

A special TANGO device server called the *TANGO Database Server* will act as a naming service between TANGO servers and clients. This server has a known address where it can be reached. The machines that run TANGO Device Servers and/or TANGO clients, should export an environment variable called `TANGO_HOST` that points to the TANGO Database server address. Example: `TANGO_HOST=homer.lab.eu:10000`

2.2 Minimum setup

This chapter assumes you have already installed PyTango.

To explore PyTango you should have a running Tango system. If you are working in a facility/institute that uses Tango, this has probably already been prepared for you. You need to ask your facility/institute tango contact for the `TANGO_HOST` variable where Tango system is running.

If you are working in an isolate machine you first need to make sure the Tango system is installed and running (see tango how to).

Most examples here connect to a device called *sys/tg_test/1* that runs in a TANGO server called *TangoTest* with the instance name *test*. This server comes with the TANGO installation. The TANGO installation also registers the *test* instance. All you have to do is start the TangoTest server on a console:

```
$ TangoTest test
Ready to accept request
```

Note: if you receive a message saying that the server is already running, it just means that somebody has already started the test server so you don't need to do anything.

2.3 Client

Finally you can get your hands dirty. The first thing to do is start a python console and import the *tango* module. The following example shows how to create a proxy to an existing TANGO device, how to read and write attributes and execute commands from a python console:

```
>>> import tango

>>> # create a device object
>>> test_device = tango.DeviceProxy("sys/tg_test/1")

>>> # every device has a state and status which can be checked with:
>>> print(test_device.state())
RUNNING

>>> print(test_device.status())
The device is in RUNNING state.

>>> # this device has an attribute called "long_scalar". Let's see which value it has...
>>> data = test_device.read_attribute("long_scalar")

>>> # ...PyTango provides a shortcut to do the same:
>>> data = test_device["long_scalar"]

>>> # the result of reading an attribute is a DeviceAttribute python object.
>>> # It has a member called "value" which contains the value of the attribute
>>> data.value
136

>>> # Check the complete DeviceAttribute members:
>>> print(data)
DeviceAttribute[
data_format = SCALAR
      dim_x = 1
      dim_y = 0
 has_failed = False
   is_empty = False
       name = 'long_scalar'
    nb_read = 1
 nb_written = 1
    quality = ATTR_VALID
r_dimension = AttributeDimension(dim_x = 1, dim_y = 0)
```

```
       time = TimeVal(tv_nsec = 0, tv_sec = 1399450183, tv_usec = 323990)
       type = DevLong
      value = 136
    w_dim_x = 1
    w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 1, dim_y = 0)
    w_value = 0]

>>> # PyTango provides a handy pythonic shortcut to read the attribute value:
>>> test_device.long_scalar
136

>>> # Setting an attribute value is equally easy:
>>> test_device.write_attribute("long_scalar", 8776)

>>> # ... and a handy shortcut to do the same exists as well:
>>> test_device.long_scalar = 8776

>>> # TangoTest has a command called "DevDouble" which receives a number
>>> # as parameter and returns the same number as a result. Let's
>>> # execute this command:
>>> test_device.command_inout("DevDouble", 45.67)
45.67

>>> # PyTango provides a handy shortcut: it exports commands as device methods:
>>> test_device.DevDouble(45.67)
45.67

>>> # Introspection: check the list of attributes:
>>> test_device.get_attribute_list()
['ampli', 'boolean_scalar', 'double_scalar', '...', 'State', 'Status']

>>>
```

This is just the tip of the iceberg. Check the *DeviceProxy* for the complete API.

PyTango used to come with an integrated IPython based console called *ITango*, now moved to a separate project. It provides helpers to simplify console usage. You can use this console instead of the traditional python console. Be aware, though, that many of the *tricks* you can do in an *ITango* console cannot be done in a python program.

2.4 Server

Since PyTango 8.1 it has become much easier to program a Tango device server. PyTango provides some helpers that allow developers to simplify the programming of a Tango device server.

Before creating a server you need to decide:

1. The name of the device server (example: *PowerSupplyDS*). This will be the mandatory name of your python file.
2. The Tango Class name of your device (example: *PowerSupply*). In our example we will use the same name as the python class name.
3. the list of attributes of the device, their data type, access (read-only vs read-write), data_format (scalar, 1D, 2D)
4. the list of commands, their parameters and their result

In our example we will write a fake power supply device server. The server will be called *PowerSupplyDS*. There will be a class called *PowerSupply* which will have attributes:

- *voltage* (scalar, read-only, numeric)

- *current* (scalar, read_write, numeric, expert mode)
- *noise* (2D, read-only, numeric)

pipes:

- *info* (read-only)

commands:

- *TurnOn* (argument: None, result: None)
- *TurnOff* (argument: None, result: None)
- *Ramp* (param: scalar, numeric; result: bool)

properties:

- *host* (string representing the host name of the actual power supply)
- *port* (port number in the host with default value = 9788)

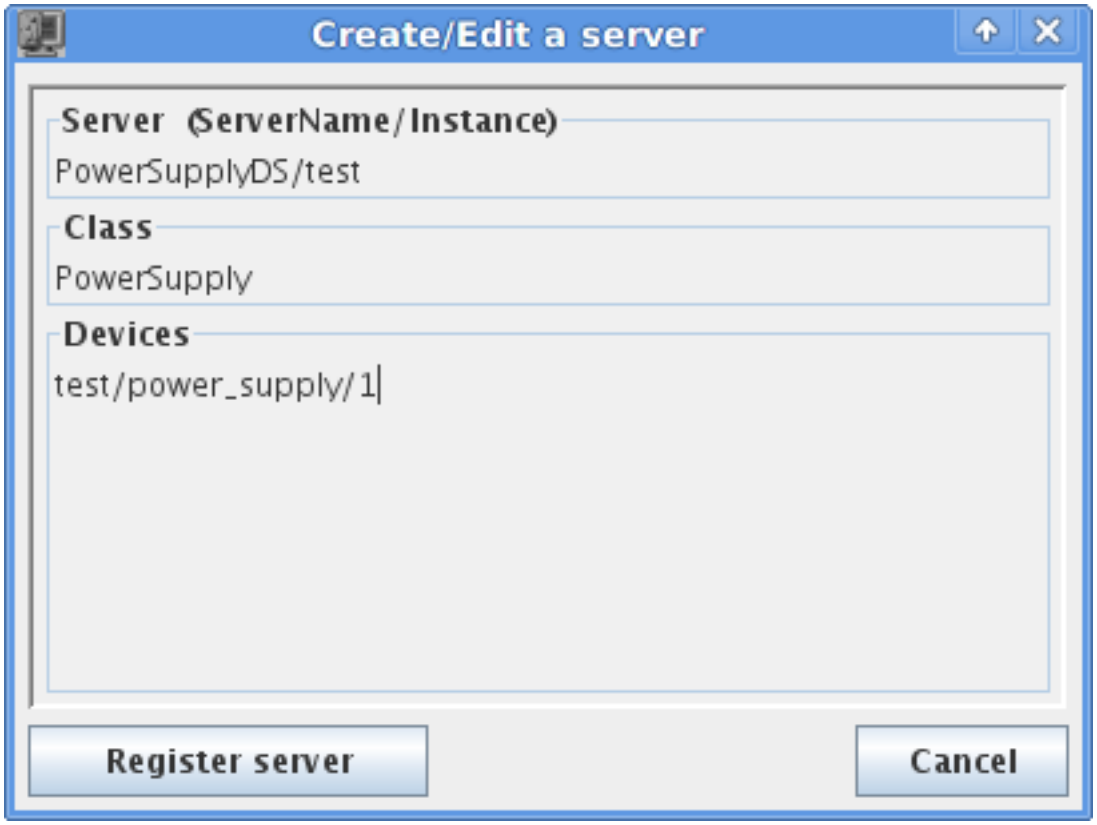
Here is the code for the `PowerSupplyDS.py`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Demo power supply tango device server"""

import time
import numpy

from tango import AttrQuality, AttrWriteType, DispLevel, DevState, DebugIt
from tango.server import Device, DeviceMeta, attribute, command, pipe, run
from tango.server import device_property


class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    voltage = attribute(label="Voltage", dtype=float,
                        display_level=DispLevel.OPERATOR,
                        access=AttrWriteType.READ,
                        unit="V",format="8.4f",
                        doc="the power supply voltage")

    current = attribute(label="Current", dtype=float,
                        display_level=DispLevel.EXPERT,
                        access=AttrWriteType.READ_WRITE,
                        unit="A",format="8.4f",
                        min_value=0.0, max_value=8.5,
                        min_alarm=0.1, max_alarm=8.4,
                        min_warning=0.5, max_warning=8.0,
                        fget="get_current",
                        fset="set_current",
                        doc="the power supply current")

    noise = attribute(label="Noise",
                      dtype=((int,),),
                      max_dim_x=1024, max_dim_y=1024)

    info = pipe(label='Info')

    host = device_property(dtype=str)
    port = device_property(dtype=int, default_value=9788)

    def init_device(self):
```

```
        Device.init_device(self)
        self.__current = 0.0
        self.set_state(DevState.STANDBY)

    def read_voltage(self):
        self.info_stream("read_voltage(%s, %d)", self.host, self.port)
        return 9.99, time.time(), AttrQuality.ATTR_WARNING

    def get_current(self):
        return self.__current

    def set_current(self, current):
        # should set the power supply current
        self.__current = current

    def read_info(self):
        return 'Information', dict(manufacturer='Tango',
                                   model='PS2000',
                                   version_number=123)

    @DebugIt()
    def read_noise(self):
        return numpy.random.random_integers(1000, size=(100, 100))

    @command
    def TurnOn(self):
        # turn on the actual power supply here
        self.set_state(DevState.ON)

    @command
    def TurnOff(self):
        # turn off the actual power supply here
        self.set_state(DevState.OFF)

    @command(dtype_in=float, doc_in="Ramp target current",
             dtype_out=bool, doc_out="True if ramping went well, False otherwise")
    def Ramp(self, target_current):
        # should do the ramping
        return True


if __name__ == "__main__":
    run([PowerSupply])

```

Check the *high level server API* for the complete reference API. The *write a server how to* can help as well.

Before running this brand new server we need to register it in the Tango system. You can do it with Jive (*Jive->Edit->Create server*):



... or in a python script:

```
>>> import tango

>>> dev_info = tango.DbDevInfo()
>>> dev_info.server = "PowerSupplyDS/test"
>>> dev_info._class = "PowerSupply"
>>> dev_info.name = "test/power_supply/1"

>>> db = tango.Database()
>>> db.add_device(dev_info)
```

After, you can run the server on a console with:

```
$ python PowerSupplyDS.py test
Ready to accept request
```

Now you can access it from a python console:

```
>>> import tango

>>> power_supply = tango.DeviceProxy("test/power/supply/1")
>>> power_supply.state()
STANDBY

>>> power_supply.current = 2.3

>>> power_supply.current
2.3

>>> power_supply.PowerOn()
>>> power_supply.Ramp(2.1)
True

>>> power_supply.state()
```

```
ON
```

Next steps: Check out the *PyTango API*.

CHAPTER 3

ITango

ITango is a PyTango CLI based on IPython. It is designed to be used as an IPython profile.

ITango is available since PyTango 7.1.2 and has been moved to a separate project since PyTango 9.2.0:

- package and instructions on PyPI
- sources on GitHub
- documentation on pythonhosted

CHAPTER 4

Green mode

PyTango supports cooperative green Tango objects. Since version 8.1 two *green* modes have been added: `Futures` and `Gevent`.

The `Futures` uses the standard python module `concurrent.futures`. The `Gevent` mode uses the well known gevent library.

Currently, in version 8.1, only *DeviceProxy* has been modified to work in a green cooperative way. If the work is found to be useful, the same can be implemented in the future for *AttributeProxy*, *Database*, *Group* or even in the server side.

You can set the PyTango green mode at a global level. Set the environment variable `PYTANGO_GREEN_MODE` to either *futures* or *gevent* (case incensitive). If this environment variable is not defined the PyTango global green mode defaults to *Synchronous*.

You can also change the active global green mode at any time in your program:

```
>>> from tango import DeviceProxy, GreenMode
>>> from tango import set_green_mode, get_green_mode

>>> get_green_mode()
tango.GreenMode.Synchronous

>>> dev = DeviceProxy("sys/tg_test/1")
>>> dev.get_green_mode()
tango.GreenMode.Synchronous

>>> set_green_mode(GreenMode.Futures)
>>> get_green_mode()
tango.GreenMode.Futures

>>> dev.get_green_mode()
tango.GreenMode.Futures
```

As you can see by the example, the global green mode will affect any previously created *DeviceProxy* using the default *DeviceProxy* constructor parameters.

You can specificy green mode on a *DeviceProxy* at creation time. You can also change the green mode at any time:

```
>>> from tango.futures import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
>>> dev.get_green_mode()
tango.GreenMode.Futures

>>> dev.set_green_mode(GreenMode.Synchronous)
>>> dev.get_green_mode()
tango.GreenMode.Synchronous
```

4.1 futures mode

Using `concurrent.futures` cooperative mode in PyTango is relatively easy:

```
>>> from tango.futures import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
>>> dev.get_green_mode()
tango.GreenMode.Futures

>>> print(dev.state())
RUNNING
```

The *tango.futures.DeviceProxy()* API is exactly the same as the standard *DeviceProxy*. The difference is in the semantics of the methods that involve synchronous network calls (constructor included) which may block the execution for a relatively big amount of time. The list of methods that have been modified to accept *futures* semantics are, on the *tango.futures.DeviceProxy()*:

- Constructor
- *state()*
- *status()*
- *read_attribute()*
- *write_attribute()*
- *write_read_attribute()*
- *read_attributes()*
- *write_attributes()*
- *ping()*

So how does this work in fact? I see no difference from using the *standard* *DeviceProxy*. Well, this is, in fact, one of the goals: be able to use a *futures* cooperation without changing the API. Behind the scenes the methods mentioned before have been modified to be able to work cooperatively.

All of the above methods have been boosted with two extra keyword arguments *wait* and *timeout* which allow to fine tune the behaviour. The *wait* parameter is by default set to *True* meaning wait for the request to finish (the default semantics when not using green mode). If *wait* is set to *True*, the timeout determines the maximum time to wait for the method to execute. The default is *None* which means wait forever. If *wait* is set to *False*, the *timeout* is ignored.

If *wait* is set to *True*, the result is the same as executing the *standard* method on a *DeviceProxy*. If, *wait* is set to *False*, the result will be a `concurrent.futures.Future`. In this case, to get the actual value you will need to do something like:

```
>>> from tango.futures import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
>>> result = dev.state(wait=False)
>>> result
<Future at 0x16cb310 state=pending>

>>> # this will be the blocking code
>>> state = result.result()
>>> print(state)
RUNNING
```

Here is another example using *read_attribute()*:

```
>>> from tango.futures import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
```

```
>>> result = dev.read_attribute('wave', wait=False)
>>> result
<Future at 0x16cbe50 state=pending>

>>> dev_attr = result.result()
>>> print(dev_attr)
DeviceAttribute[
data_format = tango.AttrDataFormat.SPECTRUM
      dim_x = 256
      dim_y = 0
 has_failed = False
   is_empty = False
       name = 'wave'
    nb_read = 256
 nb_written = 0
    quality = tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 256, dim_y = 0)
       time = TimeVal(tv_nsec = 0, tv_sec = 1383923329, tv_usec = 451821)
       type = tango.CmdArgType.DevDouble
      value = array([ -9.61260664e-01,  -9.65924853e-01,  -9.70294813e-01,
        -9.74369212e-01,  -9.78146810e-01,  -9.81626455e-01,
        -9.84807087e-01,  -9.87687739e-01,  -9.90267531e-01,
        ...
        5.15044507e-1])
    w_dim_x = 0
    w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    w_value = None]
```

4.2 gevent mode

Warning: Before using gevent mode please note that at the time of writing this documentation, *tango.gevent* requires the latest version 1.0 of gevent (which has been released the day before :-P). Also take into account that gevent 1.0 is *not* available on python 3. Please consider using the *Futures* mode instead.

Using gevent cooperative mode in PyTango is relatively easy:

```
>>> from tango.gevent import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
>>> dev.get_green_mode()
tango.GreenMode.Gevent

>>> print(dev.state())
RUNNING
```

The *tango.gevent.DeviceProxy()* API is exactly the same as the standard *DeviceProxy*. The difference is in the semantics of the methods that involve synchronous network calls (constructor included) which may block the execution for a relatively big amount of time. The list of methods that have been modified to accept *gevent* semantics are, on the *tango.gevent.DeviceProxy()*:

- Constructor
- *state()*
- *status()*
- *read_attribute()*
- *write_attribute()*

- *write_read_attribute()*
- *read_attributes()*
- *write_attributes()*
- *ping()*

So how does this work in fact? I see no difference from using the *standard* *DeviceProxy*. Well, this is, in fact, one of the goals: be able to use a gevent cooperation without changing the API. Behind the scenes the methods mentioned before have been modified to be able to work cooperatively with other greenlets.

All of the above methods have been boosted with two extra keyword arguments *wait* and *timeout* which allow to fine tune the behaviour. The *wait* parameter is by default set to *True* meaning wait for the request to finish (the default semantics when not using green mode). If *wait* is set to *True*, the timeout determines the maximum time to wait for the method to execute. The default timeout is *None* which means wait forever. If *wait* is set to *False*, the *timeout* is ignored.

If *wait* is set to *True*, the result is the same as executing the *standard* method on a *DeviceProxy*. If, *wait* is set to *False*, the result will be a `gevent.event.AsyncResult`. In this case, to get the actual value you will need to do something like:

```
>>> from tango.gevent import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
>>> result = dev.state(wait=False)
>>> result
<gevent.event.AsyncResult at 0x1a74050>

>>> # this will be the blocking code
>>> state = result.get()
>>> print(state)
RUNNING
```

Here is another example using *read_attribute()*:

```
>>> from tango.gevent import DeviceProxy

>>> dev = DeviceProxy("sys/tg_test/1")
>>> result = dev.read_attribute('wave', wait=False)
>>> result
<gevent.event.AsyncResult at 0x1aff54e>

>>> dev_attr = result.get()
>>> print(dev_attr)
DeviceAttribute[
data_format = tango.AttrDataFormat.SPECTRUM
      dim_x = 256
      dim_y = 0
 has_failed = False
   is_empty = False
       name = 'wave'
    nb_read = 256
 nb_written = 0
    quality = tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 256, dim_y = 0)
       time = TimeVal(tv_nsec = 0, tv_sec = 1383923292, tv_usec = 886720)
       type = tango.CmdArgType.DevDouble
      value = array([ -9.61260664e-01,  -9.65924853e-01,  -9.70294813e-01,
        -9.74369212e-01,  -9.78146810e-01,  -9.81626455e-01,
        -9.84807087e-01,  -9.87687739e-01,  -9.90267531e-01,
        ...
        5.15044507e-1])
    w_dim_x = 0
```

```
    w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    w_value = None]
```

Note: due to the internal workings of gevent, setting the *wait* flag to *True* (default) doesn't prevent other greenlets from running in *parallel*. This is, in fact, one of the major bonus of working with `gevent` when compared with `concurrent.futures`

CHAPTER 5

PyTango API

This module implements the Python Tango Device API mapping.

5.1 Data types

This chapter describes the mapping of data types between Python and Tango.

Tango has more data types than Python which is more dynamic. The input and output values of the commands are translated according to the array below. Note that if PyTango is compiled with `numpy` support the numpy type will be the used for the input arguments. Also, it is recomended to use numpy arrays of the appropiate type for output arguments as well, as they tend to be much more efficient.

For scalar types (SCALAR)

Tango data type	Python 2.x type	Python 3.x type (*New in PyTango 8.0*)
DEV_VOID	No data	No data
DEV_BOOLEAN	`bool`	`bool`
DEV_SHORT	`int`	`int`
DEV_LONG	`int`	`int`
DEV_LONG64	• `long` (on a 32 bits computer) • `int` (on a 64 bits computer)	`int`
DEV_FLOAT	`float`	`float`
DEV_DOUBLE	`float`	`float`
DEV_USHORT	`int`	`int`
DEV_ULONG	`int`	`int`
DEV_ULONG64	• `long` (on a 32 bits computer) • `int` (on a 64 bits computer)	`int`
DEV_STRING	`str`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
DEV_ENCODED (*New in PyTango 8.0*)	sequence of two elements: 0. `str` 1. `bytes` (for any value of *extract_as*)	sequence of two elements: 0. `str` (decoded with *latin-1*, aka *ISO-8859-1*) 1. `bytes` (for any value of *extract_as*, except String. In this case it is `str` (decoded with default python encoding *utf-8*))

For array types (SPECTRUM/IMAGE)

Tango data type	ExtractAs	Data type (Python 2.x)	Data type (Python 3.x) (*New in PyTango 8.0*)
DEVVAR_CHARARRAY	Numpy	`numpy.ndarray` (dtype= `numpy.uint8`)	`numpy.ndarray` (dtype= `numpy.uint8`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list` <`int`>	`list` <`int`>
	Tuple	`tuple` <`int`>	`tuple` <`int`>
DEVVAR_SHORTARRAY or (DEV_SHORT + SPECTRUM) or (DEV_SHORT + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.uint16`)	`numpy.ndarray` (dtype= `numpy.uint16`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list` <`int`>	`list` <`int`>

Continued on next page

Table 5.1 – continued from previous page

Tango data type	ExtractAs	Data type (Python 2.x)	Data type (Python 3.x) (*New in PyTango 8.0*)
	Tuple	`tuple <int>`	`tuple <int>`
DEVVAR_LONGARRAY or (DEV_LONG + SPECTRUM) or (DEV_LONG + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.uint32`)	`numpy.ndarray` (dtype= `numpy.uint32`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list <int>`	`list <int>`
	Tuple	`tuple <int>`	`tuple <int>`
DEVVAR_LONG64ARRAY or (DEV_LONG64 + SPECTRUM) or (DEV_LONG64 + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.uint64`)	`numpy.ndarray` (dtype= `numpy.uint64`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list` <int (64 bits) / long (32 bits)>	`list <int>`
	Tuple	`tuple` <int (64 bits) / long (32 bits)>	`tuple <int>`
DEVVAR_FLOATARRAY or (DEV_FLOAT + SPECTRUM) or (DEV_FLOAT + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.float32`)	`numpy.ndarray` (dtype= `numpy.float32`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list <float>`	`list <float>`
	Tuple	`tuple <float>`	`tuple <float>`
DEVVAR_DOUBLEARRAY or (DEV_DOUBLE + SPECTRUM) or (DEV_DOUBLE + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.float64`)	`numpy.ndarray` (dtype= `numpy.float64`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list <float>`	`list <float>`
	Tuple	`tuple <float>`	`tuple <float>`
DEVVAR_USHORTARRAY or (DEV_USHORT + SPECTRUM) or (DEV_USHORT + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.uint16`)	`numpy.ndarray` (dtype= `numpy.uint16`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`

Continued on next page

Table 5.1 – continued from previous page

Tango data type	ExtractAs	Data type (Python 2.x)	Data type (Python 3.x) (*New in PyTango 8.0*)
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list` <`int`>	`list` <`int`>
	Tuple	`tuple` <`int`>	`tuple` <`int`>
DEVVAR_ULONGARRAY or (DEV_ULONG + SPECTRUM) or (DEV_ULONG + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.uint32`)	`numpy.ndarray` (dtype= `numpy.uint32`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list` <`int`>	`list` <`int`>
	Tuple	`tuple` <`int`>	`tuple` <`int`>
DEVVAR_ULONG64ARRAY or (DEV_ULONG64 + SPECTRUM) or (DEV_ULONG64 + IMAGE)	Numpy	`numpy.ndarray` (dtype= `numpy.uint64`)	`numpy.ndarray` (dtype= `numpy.uint64`)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	String `str` (decoded with default python encoding *utf-8*!!!)
	List	`list` <int (64 bits) / long (32 bits)>	`list` <`int`>
	Tuple	`tuple` <int (64 bits) / long (32 bits)>	`tuple` <`int`>
DEVVAR_STRINGARRAY or (DEV_STRING + SPECTRUM) or (DEV_STRING + IMAGE)		sequence<`str`>	sequence<`str`> (decoded with *latin-1*, aka *ISO-8859-1*)
DEV_LONGSTRINGARRAY		sequence of two elements: 0. `numpy.ndarray` (dtype= `numpy.int32`) or sequence<`int`> 1. sequence<`str`>	sequence of two elements: 0. `numpy.ndarray` (dtype= `numpy.int32`) or sequence<`int`> 1. sequence<`str`> (decoded with *latin-1*, aka *ISO-8859-1*)

Continued on next page

Table 5.1 – continued from previous page

Tango data type	ExtractAs	Data type (Python 2.x)	Data type (Python 3.x) (*New in PyTango 8.0*)
DEV_DOUBLESTRINGARRAY		sequence of two elements: 0. `numpy.ndarray` (dtype= `numpy.float64`) or sequence<`int`> 1. sequence<`str`>	sequence of two elements: 0. `numpy.ndarray` (dtype= `numpy.float64`) or sequence<`int`> 1. sequence<`str`> (decoded with *latin-1*, aka *ISO-8859-1*)

For SPECTRUM and IMAGES the actual sequence object used depends on the context where the tango data is used, and the availability of `numpy`.

1. for properties the sequence is always a `list`. Example:

```
>>> import tango
>>> db = tango.Database()
>>> s = db.get_property(["TangoSynchrotrons"])
>>> print type(s)
<type 'list'>
```

2. **for attribute/command values**

   - `numpy.ndarray` if PyTango was compiled with `numpy` support (default) and `numpy` is installed.
   - `list` otherwise

5.1.1 Pipe data types

Pipes require different data types. You can think of them as a structured type.

A pipe transports data which is called a *blob*. A *blob* consists of name and a list of fields. Each field is called *data element*. Each *data element* consists of a name and a value. *Data element* names must be unique in the same blob.

The value can be of any of the SCALAR or SPECTRUM tango data types (except DevEnum).

Additionally, the value can be a *blob* itself.

In PyTango, a *blob* is represented by a sequence of two elements:

- blob name (str)
- data is either:
  - sequence (`list`, `tuple`, or other) of data elements where each element is a `dict` with the following keys:
    - *name* (mandatory): (str) data element name
    - *value* (mandatory): data (compatible with any of the SCALAR or SPECTRUM data types except DevEnum). If value is to be a sub-*blob* then it should be sequence of [*blob name*, sequence of data elements] (see above)
    - *dtype* (optional, mandatory if a DevEncoded is required): see *Data type equivalence*. If dtype key is not given, PyTango will try to find the proper tango type by inspecting the value.

- a `dict` where key is the data element name and value is the data element value (compact version)

When using the compact dictionary version note that the order of the data elements is lost. If the order is important for you, consider using `collections.OrderedDict` instead (if you have python >=2.7. If not you can use `ordereddict` backport module available on pypi). Also, in compact mode it is not possible to enforce a specific type. As a consequence, DevEncoded is not supported in compact mode.

The description sounds more complicated that it actually is. Here are some practical examples of what you can return in a server as a read request from a pipe:

```
import numpy as np

# plain (one level) blob showing different tango data types
# (explicity and implicit):

PIPE0 = \
('BlobCase0',
 ({'name': 'DE1', 'value': 123,},                                  # converts to DevLong64
  {'name': 'DE2', 'value': np.int32(456),},                        # converts to DevLong
  {'name': 'DE3', 'value': 789, 'dtype': 'int32'},                 # converts to DevLong
  {'name': 'DE4', 'value': np.uint32(123)},                        # converts to DevULong
  {'name': 'DE5', 'value': range(5), 'dtype': ('uint16',)},        # converts to DevVarUShortArray
  {'name': 'DE6', 'value': [1.11, 2.22], 'dtype': ('float64',)}, # converts to DevVarDoubleArray
  {'name': 'DE7', 'value': numpy.zeros((100,))},                   # converts to DevVarDoubleArray
  {'name': 'DE8', 'value': True},                                  # converts to DevBoolean
 )
)


# similar as above but in compact version (implicit data type conversion):

PIPE1 = \
('BlobCase1', dict(DE1=123, DE2=np.int32(456), DE3=np.int32(789),
                   DE4=np.uint32(123), DE5=np.arange(5, dtype='uint16'),
                   DE6=[1.11, 2.22], DE7=numpy.zeros((100,)),
                   DE8=True)
)

# similar as above but order matters so we use an ordered dict:

import collections

data = collections.OrderedDict()
data['DE1'] = 123
data['DE2'] = np.int32(456)
data['DE3'] = np.int32(789)
data['DE4'] = np.uint32(123)
data['DE5'] = np.arange(5, dtype='uint16')
data['DE6'] = [1.11, 2.22]
data['DE7'] = numpy.zeros((100,))
data['DE8'] = True

PIPE2 = 'BlobCase2', data

# another plain blob showing string, string array and encoded data types:

PIPE3 = \
('BlobCase3',
 ({'name': 'stringDE',  'value': 'Hello'},
  {'name': 'VectorStringDE', 'value': ('bonjour', 'le', 'monde')},
  {'name': 'DevEncodedDE', 'value': ('json', '"isn\'t it?"'), 'dtype': 'bytes'},
 )
)
```

```
# blob with sub-blob which in turn has a sub-blob

PIPE4 = \
('BlobCase4',
 ({'name': '1DE', 'value': ('Inner', ({'name': '1_1DE', 'value': 'Grenoble'},
                                      {'name': '1_2DE', 'value': ('InnerInner',
                                                                  ({'name': '1_1_1DE', 'value': np
                                                                   {'name': '1_1_2DE', 'value': [
                                      })
  )},
  {'name': '2DE', 'value': (3,4,5,6), 'dtype': ('int32',) },
 )
)
```

5.2 Client API

5.2.1 DeviceProxy

class tango.**DeviceProxy**

DeviceProxy is the high level Tango object which provides the client with an easy-to-use interface to TANGO devices. DeviceProxy provides interfaces to all TANGO Device interfaces.The DeviceProxy manages timeouts, stateless connections and reconnection if the device server is restarted. To create a DeviceProxy, a Tango Device name must be set in the object constructor.

Example : dev = tango.DeviceProxy("sys/tg_test/1")

DeviceProxy(dev_name, green_mode=None, wait=True, timeout=True) -> DeviceProxy DeviceProxy(self, dev_name, need_check_acc, green_mode=None, wait=True, timeout=True) -> DeviceProxy

Creates a new *DeviceProxy*.

Parameters

- **dev_name** (*str*) – the device name or alias
- **need_check_acc** (*bool*) – in first version of the function it defaults to True. Determines if at creation time of DeviceProxy it should check for channel access (rarely used)
- **green_mode** (*GreenMode*) – determines the mode of execution of the device (including. the way it is created). Defaults to the current global green_mode (check *get_green_mode()* and *set_green_mode()*)
- **wait** (*bool*) – whether or not to wait for result. If green_mode Ignored when green_mode is Synchronous (always waits).
- **timeout** (*float*) – The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Returns

if green_mode is Synchronous or wait is True: *DeviceProxy*

elif green_mode is Futures: concurrent.futures.Future

elif green_mode is Gevent: gevent.event.AsynchResult

Throws

- : class:*~tango.DevFailed* if green_mode is Synchronous or wait is True and there is an error creating the device.

- : class:*concurrent.futures.TimeoutError* if green_mode is Futures, wait is False, timeout is not None and the time to create the device has expired.
- : class:*gevent.timeout.Timeout* if green_mode is Gevent, wait is False, timeout is not None and the time to create the device has expired.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

delete_property(*self*, *value*)

Delete a the given of properties for this device. This method accepts the following types as value parameter:

1.string [in] - single property to be deleted

2.tango.DbDatum [in] - single property data to be deleted

3.tango.DbData [in] - several property data to be deleted

4.sequence<string> [in]- several property data to be deleted

5.sequence<DbDatum> [in] - several property data to be deleted

6.dict<str, obj> [in] - keys are property names to be deleted (values are ignored)

7.dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Parameters

value can be one of the following:

1. string [in] - single property data to be deleted
2. tango.DbDatum [in] - single property data to be deleted
3. tango.DbData [in] - several property data to be deleted
4. sequence<string> [in]- several property data to be deleted
5. sequence<DbDatum> [in] - several property data to be deleted
6. dict<str, obj> [in] - keys are property names to be deleted (values are ignored)
7. dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Return None

Throws *ConnectionFailed*, *CommunicationFailed* *DevFailed* from device (DB_SQLError)

get_attribute_config(*self*, *name*) → AttributeInfoEx

Return the attribute configuration for a single attribute.

Parameters

name (*str*) attribute name

Return (*AttributeInfoEx*) Object containing the attribute information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

Deprecated: use get_attribute_config_ex instead

get_attribute_config(self, names) -> AttributeInfoList

Return the attribute configuration for the list of specified attributes. To get all the attributes pass a sequence containing the constant tango.:class:*constants*.AllAttr

Parameters

names (sequence<str>) attribute names

Return (AttributeInfoList) Object containing the attributes information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

Deprecated: use get_attribute_config_ex instead

get_attribute_config_ex(*self*, *name*) → AttributeInfoListEx :

Return the extended attribute configuration for a single attribute.

Parameters

name (str) attribute name

Return (*AttributeInfoEx*) Object containing the attribute information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_attribute_config(self, names) -> AttributeInfoListEx :

Return the extended attribute configuration for the list of specified attributes. To get all the attributes pass a sequence containing the constant tango.:class:*constants*.AllAttr

Parameters

names (sequence<str>) attribute names

Return (AttributeInfoList) Object containing the attributes information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_command_config(*self*) → CommandInfoList

Return the command configuration for all commands.

Return (CommandInfoList) Object containing the commands information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_command_config(self, name) -> *CommandInfo*

Return the command configuration for a single command.

Parameters

name (str) command name

Return (*CommandInfo*) Object containing the command information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_command_config(self, names) -> CommandInfoList

Return the command configuration for the list of specified commands.

Parameters

names (sequence<str>) command names

Return (CommandInfoList) Object containing the commands information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_events(*event_id*, *callback=None*, *extract_as=Numpy*) → None

The method extracts all waiting events from the event reception buffer.

If callback is not None, it is executed for every event. During event subscription the client must have chosen the pull model for this event. The callback will receive a parameter of type EventData, AttrConfEventData or DataReadyEventData depending on the type of the event (event_type parameter of subscribe_event).

If callback is None, the method extracts all waiting events from the event reception buffer. The returned event_list is a vector of EventData, AttrConfEventData or DataReadyEventData pointers, just the same data the callback would have received.

Parameters

event_id (int) is the event identifier returned by the DeviceProxy.subscribe_event() method.

callback (callable) Any callable object or any object with a "push_event" method.

extract_as (ExtractAs)

Return None

Throws *EventSystemFailed*

See Also subscribe_event

New in PyTango 7.0.0

get_green_mode()

Returns the green mode in use by this DeviceProxy.

Returns the green mode in use by this DeviceProxy.

Return type *GreenMode*

See also:

tango.get_green_mode() tango.set_green_mode()

New in PyTango 8.1.0

get_pipe_config(*self*) → PipeInfoList

Return the pipe configuration for all pipes.

Return (PipeInfoList) Object containing the pipes information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_pipe_config(self, name) -> PipeInfo

Return the `pipe` configuration for a single `pipe`.

Parameters

name (`str`) pipe name

Return (`PipeInfo`) Object containing the pipe information

Throws *`ConnectionFailed`*, *`CommunicationFailed`*, *`DevFailed`* from device

get_pipe_config(self, names) -> PipeInfoList

Return the `pipe` configuration for the list of specified pipes. To get all the pipes pass a sequence containing the constant tango.:class:*constants*.AllPipe

Parameters

names (sequence<`str`>) pipe names

Return (`PipeInfoList`) Object containing the pipes information

Throws *`ConnectionFailed`*, *`CommunicationFailed`*, *`DevFailed`* from device

New in PyTango 9.2.0

`get_property` (*propname*, *value=None*) → tango.DbData

Get a (list) property(ies) for a device.

This method accepts the following types as propname parameter: 1. string [in] - single property data to be fetched 2. sequence<string> [in] - several property data to be fetched 3. tango.DbDatum [in] - single property data to be fetched 4. tango.DbData [in,out] - several property data to be fetched. 5. sequence<DbDatum> - several property data to be feteched

Note: for cases 3, 4 and 5 the 'value' parameter if given, is IGNORED.

If value is given it must be a tango.DbData that will be filled with the property values

Parameters

propname (`any`) property(ies) name(s)

value (`DbData`) (optional, default is None meaning that the method will create internally a tango.DbData and return it filled with the property values

Return (`DbData`) object containing the property(ies) value(s). If a tango.DbData is given as parameter, it returns the same object otherwise a new tango.DbData is returned

Throws *`NonDbDevice`*, *`ConnectionFailed`* (with database), *`CommunicationFailed`* (with database), *`DevFailed`* from database device

`get_property_list` (*self*, *filter*, *array=None*) → obj

Get the list of property names for the device. The parameter filter allows the user to filter the returned name list. The wildcard character is '*'. Only one wildcard character is allowed in the filter parameter.

Parameters

filter[in] (str) the filter wildcard

array[out] (sequence obj or None) (optional, default is None) an array to be filled with the property names. If None a new list will be created internally with the values.

Return the given array filled with the property names (or a new list if array is None)

Throws *NonDbDevice*, *WrongNameSyntax*, *ConnectionFailed* (with database), *CommunicationFailed* (with database) *DevFailed* from database device

New in PyTango 7.0.0

ping(*self*) → int

A method which sends a ping to the device

Parameters None

Return (int) time elapsed in microseconds

Throws exception if device is not alive

put_property(*self*, *value*) → None

Insert or update a list of properties for this device. This method accepts the following types as value parameter: 1. tango.DbDatum - single property data to be inserted 2. tango.DbData - several property data to be inserted 3. sequence<DbDatum> - several property data to be inserted 4. dict<str, DbDatum> - keys are property names and value has data to be inserted 5. dict<str, seq<str>> - keys are property names and value has data to be inserted 6. dict<str, obj> - keys are property names and str(obj) is property value

Parameters

value can be one of the following: 1. tango.DbDatum - single property data to be inserted 2. tango.DbData - several property data to be inserted 3. sequence<DbDatum> - several property data to be inserted 4. dict<str, DbDatum> - keys are property names and value has data to be inserted 5. dict<str, seq<str>> - keys are property names and value has data to be inserted 6. dict<str, obj> - keys are property names and str(obj) is property value

Return None

Throws *ConnectionFailed*, *CommunicationFailed* *DevFailed* from device (DB_SQLError)

read_attribute(*self*, *attr_name*, *extract_as=ExtractAs.Numpy*, *green_mode=None*, *wait=True*, *timeout=None*) → DeviceAttribute

Read a single attribute.

Parameters

attr_name (str) The name of the attribute to read.

extract_as (ExtractAs) Defaults to numpy.

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (bool) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (float) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Return (*DeviceAttribute*)

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

Changed in version 7.1.4: For DevEncoded attributes, before it was returning a *DeviceAttribute*.value as a tuple **(format<str>, data<str>)** no matter what was the *extract_as* value was. Since 7.1.4, it returns a **(format<str>, data<buffer>)** unless *extract_as* is String, in which case it returns **(format<str>, data<str>)**.

Changed in version 8.0.0: For DevEncoded attributes, now returns a *DeviceAttribute*.value as a tuple **(format<str>, data<bytes>)** unless *extract_as* is String, in which case it returns **(format<str>, data<str>)**. Carefull, if using python >= 3 data<str> is decoded using default python *utf-8* encoding. This means that PyTango assumes tango DS was written encapsulating string into *utf-8* which is the default python encoding.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

read_attribute_asynch (*self*, *attr_name*) → int
read_attribute_asynch *(self, attr_name, callback)* **->** None

Shortcut to self.read_attributes_asynch([attr_name], cb)

New in PyTango 7.0.0

read_attribute_reply (*self*, *id*, *extract_as*) → int
read_attribute_reply *(self, id, timeout, extract_as)* **->** None

Shortcut to self.read_attributes_reply()[0]

New in PyTango 7.0.0

read_attributes (*self*, *attr_names*, *extract_as=ExtractAs.Numpy*, *green_mode=None*, *wait=True*, *timeout=None*) → sequence<DeviceAttribute>

Read the list of specified attributes.

Parameters

attr_names (sequence<str>) A list of attributes to read.

extract_as (ExtractAs) Defaults to numpy.

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (bool) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (float) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Return (sequence<*DeviceAttribute*>)

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

read_attributes_asynch(*self*, *attr_names*) → int

Read asynchronously (polling model) the list of specified attributes.

Parameters

attr_names (sequence<str>) A list of attributes to read. It should be a StdStringVector or a sequence of str.

Return an asynchronous call identifier which is needed to get attributes value.

Throws *ConnectionFailed*

New in PyTango 7.0.0

read_attributes_asynch *(self, attr_names, callback, extract_as=Numpy)* **->** None

Read asynchronously (push model) an attribute list.

Parameters

attr_names (sequence<str>) A list of attributes to read. See read_attributes.

callback (callable) This callback object should be an instance of a user class with an attr_read() method. It can also be any callable object.

extract_as (ExtractAs) Defaults to numpy.

Return None

Throws *ConnectionFailed*

New in PyTango 7.0.0

Important: by default, TANGO is initialized with the **polling** model. If you want to use the **push** model (the one with the callback parameter), you need to change the global TANGO model to PUSH_CALLBACK. You can do this with the `tango.ApiUtil.set_asynch_cb_sub_model()`

---

read_pipe(*self*, *pipe_name*, *extract_as=ExtractAs.Numpy*, *green_mode=None*, *wait=True*, *timeout=None*) → tuple

Read a single pipe. The result is a *blob*: a tuple with two elements: blob name (string) and blob data (sequence). The blob data consists of a sequence where each element is a dictionary with the following keys:

- name: blob element name
- dtype: tango data type

- value: blob element data (str for DevString, etc)

In case dtype is `DevPipeBlob`, value is again a *blob*.

Parameters

pipe_name (`str`) The name of the pipe to read.

extract_as (`ExtractAs`) Defaults to numpy.

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (`bool`) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (`float`) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Return tuple<str, sequence>

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in PyTango 9.2.0

set_attribute_config(*self*, *attr_info*) → None

Change the attribute configuration for the specified attribute

Parameters

attr_info (*AttributeInfo*) attribute information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_attribute_config(self, attr_info_ex) -> None

Change the extended attribute configuration for the specified attribute

Parameters

attr_info_ex (*AttributeInfoEx*) extended attribute information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_attribute_config(self, attr_info) -> None

Change the attributes configuration for the specified attributes

Parameters

attr_info (sequence<*AttributeInfo*>) attributes information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_attribute_config(self, attr_info_ex) -> None

Change the extended attributes configuration for the specified attributes

Parameters

attr_info_ex (sequence<`AttributeInfoListEx`>) extended attributes information

Return None

Throws *`ConnectionFailed`*, *`CommunicationFailed`*, *`DevFailed`* from device

`set_green_mode`(*green_mode=None*)

Sets the green mode to be used by this DeviceProxy Setting it to None means use the global PyTango green mode (see *`tango.get_green_mode()`*).

Parameters **`green_mode`** (`GreenMode`) – the new green mode

New in PyTango 8.1.0

`set_pipe_config`(*self, pipe_info*) → None

Change the pipe configuration for the specified pipe

Parameters

pipe_info (`PipeInfo`) pipe information

Return None

Throws *`ConnectionFailed`*, *`CommunicationFailed`*, *`DevFailed`* from device

set_pipe_config(self, pipe_info) -> None

Change the pipes configuration for the specified pipes

Parameters

pipe_info (sequence<`PipeInfo`>) pipes information

Return None

Throws *`ConnectionFailed`*, *`CommunicationFailed`*, *`DevFailed`* from device

`state`(*self*) → DevState

A method which returns the state of the device.

Parameters None

Return (*`DevState`*) constant

Example

```
dev_st = dev.state()
if dev_st == DevState.ON : ...
```

`status`(*self*) → str

A method which returns the status of the device as a string.

Parameters None

Return (`str`) describing the device status

subscribe_event(*self*, *attr_name*, *event*, *callback*, *filters=[]*, *stateless=False*, *extract_as=Numpy*) → int

The client call to subscribe for event reception in the push model. The client implements a callback method which is triggered when the event is received. Filtering is done based on the reason specified and the event type. For example when reading the state and the reason specified is "change" the event will be fired only when the state changes. Events consist of an attribute name and the event reason. A standard set of reasons are implemented by the system, additional device specific reasons can be implemented by device servers programmers.

Parameters

attr_name (`str`) The device attribute name which will be sent as an event e.g. "current".

event_type (*EventType*) Is the event reason and must be on the enumerated values: * EventType.CHANGE_EVENT * EventType.PERIODIC_EVENT * EventType.ARCHIVE_EVENT * EventType.ATTR_CONF_EVENT * EventType.DATA_READY_EVENT * EventType.USER_EVENT

callback (`callable`) Is any callable object or an object with a callable "push_event" method.

filters (sequence<`str`>) A variable list of name,value pairs which define additional filters for events.

stateless (`bool`) When the this flag is set to false, an exception will be thrown when the event subscription encounters a problem. With the stateless flag set to true, the event subscription will always succeed, even if the corresponding device server is not running. The keep alive thread will try every 10 seconds to subscribe for the specified event. At every subscription retry, a callback is executed which contains the corresponding exception

extract_as (`ExtractAs`)

Return An event id which has to be specified when unsubscribing from this event.

Throws *EventSystemFailed*

subscribe_event(self, attr_name, event, queuesize, filters=[], stateless=False) -> int

The client call to subscribe for event reception in the pull model. Instead of a `callback` method the client has to specify the size of the event reception buffer. The event reception buffer is implemented as a round robin buffer. This way the client can set-up different ways to receive events:

- Event reception buffer size = 1 : The client is interested only in the value of the last event received. All other events that have been received since the last reading are discarded.
- Event reception buffer size > 1 : The client has chosen to keep an event history of a given size. When more events arrive since the last reading, older events will be discarded.
- Event reception buffer size = ALL_EVENTS : The client buffers all received events. The buffer size is unlimited and only restricted by the available memory for the client.

All other parameters are similar to the descriptions given in the other subscribe_event() version.

unsubscribe_event (*self*, *event_id*) → None

Unsubscribes a client from receiving the event specified by event_id.

Parameters

event_id (int) is the event identifier returned by the DeviceProxy::subscribe_event(). Unlike in TangoC++ we chech that the event_id has been subscribed in this DeviceProxy.

Return None

Throws *EventSystemFailed*

write_attribute (*self*, *attr_name*, *value*, *green_mode=None*, *wait=True*, *timeout=None*) → None
write_attribute *(self, attr_info, value, green_mode=None, wait=True, timeout=None)* **->** None

Write a single attribute.

Parameters

attr_name (str) The name of the attribute to write.

attr_info (*AttributeInfo*)

value The value. For non SCALAR attributes it may be any sequence of sequences.

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (bool) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (float) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Throws *ConnectionFailed*, *CommunicationFailed*, *DeviceUnlocked*, *DevFailed* from device TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

write_attribute_asynch (*attr_name*, *value*, *cb=None*)
write_attributes_asynch(self, values) -> int write_attributes_asynch(self, values, callback) -> None

Shortcut to self.write_attributes_asynch([attr_name, value], cb)

New in PyTango 7.0.0

write_attribute_reply (*self*, *id*) → None

Check if the answer of an asynchronous write_attribute is arrived (polling model). If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived.

Parameters

id (int) the asynchronous call identifier.

Return None

Throws *AsynCall*, *AsynReplyNotArrived*, *CommunicationFailed*, *DevFailed* from device.

New in PyTango 7.0.0

write_attribute_reply *(self, id, timeout)* **->** None

Check if the answer of an asynchronous write_attribute is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived.

Parameters

id (int) the asynchronous call identifier.

timeout (int) the timeout

Return None

Throws *AsynCall*, *AsynReplyNotArrived*, *CommunicationFailed*, *DevFailed* from device.

New in PyTango 7.0.0

write_attributes(*self*, *name_val*, *green_mode=None*, *wait=True*, *timeout=None*) → None

Write the specified attributes.

Parameters

name_val A list of pairs (attr_name, value). See write_attribute

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (bool) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (float) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Throws *ConnectionFailed*, *CommunicationFailed*, *DeviceUnlocked*, *DevFailed* or *NamedDevFailedList* from device TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

write_attributes_asynch(*self*, *values*) → int

Write asynchronously (polling model) the specified attributes.

Parameters

values (any) See write_attributes.

Return An asynchronous call identifier which is needed to get the server reply

Throws *ConnectionFailed*

New in PyTango 7.0.0

write_attributes_asynch *(self, values, callback)* **->** None

Write asynchronously (callback model) a single attribute.

Parameters

values (any) See write_attributes.

callback (callable) This callback object should be an instance of a user class with an attr_written() method . It can also be any callable object.

Return None

Throws *ConnectionFailed*

New in PyTango 7.0.0

Important: by default, TANGO is initialized with the **polling** model. If you want to use the **push** model (the one with the callback parameter), you need to change the global TANGO model to PUSH_CALLBACK. You can do this with the `tango.ApiUtil.set_asynch_cb_sub_model()`

---

write_pipe(**args*, ***kwargs*)
TODO

write_read_attribute(*self*, *attr_name*, *value*, *extract_as=ExtractAs.Numpy*, *green_mode=None*, *wait=True*, *timeout=None*) → DeviceAttribute

Write then read a single attribute in a single network call. By default (serialisation by device), the execution of this call in the server can't be interrupted by other clients.

Parameters see write_attribute(attr_name, value)

Return A tango.DeviceAttribute object.

Throws *ConnectionFailed*, *CommunicationFailed*, *DeviceUnlocked*, *DevFailed* from device, *WrongData* TimeOutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in PyTango 7.0.0

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

write_read_attributes(*self*, *name_val*, *attr_names*, *extract_as=ExtractAs.Numpy*, *green_mode=None*, *wait=True*, *timeout=None*) → DeviceAttribute

Write then read attribute(s) in a single network call. By default (serialisation by device), the execution of this call in the server can't be interrupted by other clients. On the server side, attribute(s) are first written and if no exception has been thrown during the write phase, attributes will be read.

Parameters

name_val A list of pairs (attr_name, value). See write_attribute

attr_names (sequence<`str`>) A list of attributes to read.

extract_as (`ExtractAs`) Defaults to numpy.

green_mode (*`GreenMode`*) Defaults to the current DeviceProxy GreenMode. (see *`get_green_mode()`* and *`set_green_mode()`*).

wait (`bool`) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (`float`) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Return (sequence<*`DeviceAttribute`*>)

Throws *`ConnectionFailed`*, *`CommunicationFailed`*, *`DeviceUnlocked`*, *`DevFailed`* from device, *`WrongData`* TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in PyTango 9.2.0

5.2.2 AttributeProxy

class `tango.`**`AttributeProxy`**(**args*, ***kwds*)

AttributeProxy is the high level Tango object which provides the client with an easy-to-use interface to TANGO attributes.

To create an AttributeProxy, a complete attribute name must be set in the object constructor.

Example: att = AttributeProxy("tango/tangotest/1/long_scalar")

Note: PyTango implementation of AttributeProxy is in part a python reimplementation of the AttributeProxy found on the C++ API.

`delete_property`(*self*, *value*) → None

Delete a the given of properties for this attribute. This method accepts the following types as value parameter:

1.string [in] - single property to be deleted

2.tango.DbDatum [in] - single property data to be deleted

3.tango.DbData [in] - several property data to be deleted

4.sequence<string> [in]- several property data to be deleted

5.sequence<DbDatum> [in] - several property data to be deleted

6.dict<str, obj> [in] - keys are property names to be deleted (values are ignored)

7.dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Parameters

value can be one of the following:

1. string [in] - single property data to be deleted
2. tango.DbDatum [in] - single property data to be deleted
3. tango.DbData [in] - several property data to be deleted
4. sequence<string> [in]- several property data to be deleted
5. sequence<DbDatum> [in] - several property data to be deleted
6. dict<str, obj> [in] - keys are property names to be deleted (values are ignored)
7. dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Return None

Throws *ConnectionFailed*, *CommunicationFailed* *DevFailed* from device (DB_SQLError)

event_queue_size(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().event_queue_size(...)

For convenience, here is the documentation of DeviceProxy.event_queue_size(...): None

get_config(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().get_attribute_config(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.get_attribute_config(...):

get_attribute_config(self, name) -> AttributeInfoEx

Return the attribute configuration for a single attribute.

Parameters

name (str) attribute name

Return (*AttributeInfoEx*) Object containing the attribute information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

Deprecated: use get_attribute_config_ex instead

get_attribute_config(self, names) -> AttributeInfoList

Return the attribute configuration for the list of specified attributes. To get all the attributes pass a sequence containing the constant tango.:class:*constants*.AllAttr

Parameters

names (sequence<str>) attribute names

Return (AttributeInfoList) Object containing the attributes information

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

Deprecated: use get_attribute_config_ex instead

get_device_proxy(*self*) → DeviceProxy

A method which returns the device associated to the attribute

Parameters None

Return (*DeviceProxy*)

get_events(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().get_events(...)

For convenience, here is the documentation of DeviceProxy.get_events(...):

get_events(event_id, callback=None, extract_as=Numpy) -> None

The method extracts all waiting events from the event reception buffer.

If callback is not None, it is executed for every event. During event subscription the client must have chosen the pull model for this event. The callback will receive a parameter of type EventData, AttrConfEventData or DataReadyEventData depending on the type of the event (event_type parameter of subscribe_event).

If callback is None, the method extracts all waiting events from the event reception buffer. The returned event_list is a vector of EventData, AttrConfEventData or DataReadyEventData pointers, just the same data the callback would have received.

Parameters

event_id (`int`) is the event identifier returned by the DeviceProxy.subscribe_event() method.

callback (`callable`) Any callable object or any object with a "push_event" method.

extract_as (`ExtractAs`)

Return None

Throws *EventSystemFailed*

See Also subscribe_event

New in PyTango 7.0.0

get_last_event_date(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().get_last_event_date(...)

For convenience, here is the documentation of DeviceProxy.get_last_event_date(...): None

get_poll_period(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().get_attribute_poll_period(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.get_attribute_poll_period(...): None

get_property(*self*, *propname*, *value*) → DbData

Get a (list) property(ies) for an attribute.

This method accepts the following types as propname parameter: 1. string [in] - single property data to be fetched 2. sequence<string> [in] - several property data to be fetched 3. tango.DbDatum [in] - single property data to be fetched 4. tango.DbData [in,out] - several property data to be fetched. 5. sequence<DbDatum> - several property data to be feteched

Note: for cases 3, 4 and 5 the 'value' parameter if given, is IGNORED.

If value is given it must be a tango.DbData that will be filled with the property values

Parameters

propname (str) property(ies) name(s)

value (tango.DbData) (optional, default is None meaning that the method will create internally a tango.DbData and return it filled with the property values

Return (DbData) containing the property(ies) value(s). If a tango.DbData is given as parameter, it returns the same object otherwise a new tango.DbData is returned

Throws *NonDbDevice*, *ConnectionFailed* (with database), *CommunicationFailed* (with database), *DevFailed* from database device

get_transparency_reconnection(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().get_transparency_reconnection(...)

For convenience, here is the documentation of DeviceProxy.get_transparency_reconnection(...): None

history(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().attribute_history(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.attribute_history(...): None

is_event_queue_empty(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().is_event_queue_empty(...)

For convenience, here is the documentation of DeviceProxy.is_event_queue_empty(...): None

is_polled(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().is_attribute_polled(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.is_attribute_polled(...): None

name(*self*) → str

Returns the attribute name

Parameters None

Return (str) with the attribute name

ping(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().ping(...)

For convenience, here is the documentation of DeviceProxy.ping(...):

ping(self) -> int

A method which sends a ping to the device

Parameters None

Return (int) time elapsed in microseconds

Throws exception if device is not alive

poll(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().poll_attribute(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.poll_attribute(...): None

put_property(*self*, *value*) → None

Insert or update a list of properties for this attribute. This method accepts the following types as value parameter: 1. tango.DbDatum - single property data to be inserted 2. tango.DbData - several property data to be inserted 3. sequence<DbDatum> - several property data to be inserted 4. dict<str, DbDatum> - keys are property names and value has data to be inserted 5. dict<str, seq<str>> - keys are property names and value has data to be inserted 6. dict<str, obj> - keys are property names and str(obj) is property value

Parameters

value can be one of the following: 1. tango.DbDatum - single property data to be inserted 2. tango.DbData - several property data to be inserted 3. sequence<DbDatum> - several property data to be inserted 4. dict<str, DbDatum> - keys are property names and value has data to be inserted 5. dict<str, seq<str>> - keys are property names and value has data to be inserted 6. dict<str, obj> - keys are property names and str(obj) is property value

Return None

Throws *ConnectionFailed*, *CommunicationFailed* *DevFailed* from device (DB_SQLError)

read(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().read_attribute(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.read_attribute(...):

read_attribute(self, attr_name, extract_as=ExtractAs.Numpy, green_mode=None, wait=True, timeout=None) -> DeviceAttribute

Read a single attribute.

Parameters

attr_name (`str`) The name of the attribute to read.

extract_as (`ExtractAs`) Defaults to numpy.

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (`bool`) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (`float`) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Return (*DeviceAttribute*)

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

Changed in version 7.1.4: For `DevEncoded` attributes, before it was returning a *DeviceAttribute*.value as a tuple **(format<str>, data<str>)** no matter what was the *extract_as* value was. Since 7.1.4, it returns a **(format<str>, data<buffer>)** unless *extract_as* is String, in which case it returns **(format<str>, data<str>)**.

Changed in version 8.0.0: For `DevEncoded` attributes, now returns a *DeviceAttribute*.value as a tuple **(format<str>, data<bytes>)** unless *extract_as* is String, in which case it returns **(format<str>, data<str>)**. Carefull, if using python >= 3 data<str> is decoded using default python *utf-8* encoding. This means that PyTango assumes tango DS was written encapsulating string into *utf-8* which is the default python encoding.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

read_asynch(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().read_attribute_asynch(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.read_attribute_asynch(...):

read_attribute_asynch(self, attr_name) -> int read_attribute_asynch(self, attr_name, callback) -> None

Shortcut to self.read_attributes_asynch([attr_name], cb)

New in PyTango 7.0.0

read_reply(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().read_attribute_reply(...)

For convenience, here is the documentation of DeviceProxy.read_attribute_reply(...):

read_attribute_reply(self, id, extract_as) -> int read_attribute_reply(self, id, timeout, extract_as) -> None

Shortcut to self.read_attributes_reply()[0]

New in PyTango 7.0.0

set_config(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().set_attribute_config(...)

For convenience, here is the documentation of DeviceProxy.set_attribute_config(...):

set_attribute_config(self, attr_info) -> None

Change the attribute configuration for the specified attribute

Parameters

attr_info (*AttributeInfo*) attribute information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_attribute_config(self, attr_info_ex) -> None

Change the extended attribute configuration for the specified attribute

Parameters

attr_info_ex (*AttributeInfoEx*) extended attribute information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_attribute_config(self, attr_info) -> None

Change the attributes configuration for the specified attributes

Parameters

attr_info (sequence<*AttributeInfo*>) attributes information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_attribute_config(self, attr_info_ex) -> None

Change the extended attributes configuration for the specified attributes

Parameters

attr_info_ex (sequence<AttributeInfoListEx>) extended attributes information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

set_transparency_reconnection(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().set_transparency_reconnection(...)

For convenience, here is the documentation of DeviceProxy.set_transparency_reconnection(...): None

state(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().state(...)

For convenience, here is the documentation of DeviceProxy.state(...): **state** *(self)* **->** *DevState*

A method which returns the state of the device.

Parameters None

Return (*DevState*) constant

Example

```
dev_st = dev.state()
if dev_st == DevState.ON : ...
```

status (**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().status(...)

For convenience, here is the documentation of DeviceProxy.status(...): **status** *(self)* **->** `str`

A method which returns the status of the device as a string.

Parameters None

Return (`str`) describing the device status

stop_poll (**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().stop_poll_attribute(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.stop_poll_attribute(...): None

subscribe_event (**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().subscribe_event(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.subscribe_event(...):

subscribe_event(self, attr_name, event, callback, filters=[], stateless=False, extract_as=Numpy) -> int

The client call to subscribe for event reception in the push model. The client implements a callback method which is triggered when the event is received. Filtering is done based on the reason specified and the event type. For example when reading the state and the reason specified is "change" the event will be fired only when the state changes. Events consist of an attribute name and the event reason. A standard set of reasons are implemented by the system, additional device specific reasons can be implemented by device servers programmers.

Parameters

attr_name (`str`) The device attribute name which will be sent as an event e.g. "current".

event_type (*EventType*) Is the event reason and must be on the enumerated values: * EventType.CHANGE_EVENT * EventType.PERIODIC_EVENT * EventType.ARCHIVE_EVENT * EventType.ATTR_CONF_EVENT * EventType.DATA_READY_EVENT * EventType.USER_EVENT

callback (`callable`) Is any callable object or an object with a callable "push_event" method.

filters (sequence<`str`>) A variable list of name,value pairs which define additional filters for events.

stateless (`bool`) When the this flag is set to false, an exception will be thrown when the event subscription encounters a problem. With the stateless flag set to true, the event subscription will always succeed, even if the corresponding device server is not

running. The keep alive thread will try every 10 seconds to subscribe for the specified event. At every subscription retry, a callback is executed which contains the corresponding exception

extract_as (ExtractAs)

Return An event id which has to be specified when unsubscribing from this event.

Throws *EventSystemFailed*

subscribe_event(self, attr_name, event, queuesize, filters=[], stateless=False) -> int

The client call to subscribe for event reception in the pull model. Instead of a callback method the client has to specify the size of the event reception buffer. The event reception buffer is implemented as a round robin buffer. This way the client can set-up different ways to receive events:

- Event reception buffer size = 1 : The client is interested only in the value of the last event received. All other events that have been received since the last reading are discarded.
- Event reception buffer size > 1 : The client has chosen to keep an event history of a given size. When more events arrive since the last reading, older events will be discarded.
- Event reception buffer size = ALL_EVENTS : The client buffers all received events. The buffer size is unlimited and only restricted by the available memory for the client.

All other parameters are similar to the descriptions given in the other subscribe_event() version.

unsubscribe_event(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().unsubscribe_event(...)

For convenience, here is the documentation of DeviceProxy.unsubscribe_event(...):

unsubscribe_event(self, event_id) -> None

Unsubscribes a client from receiving the event specified by event_id.

Parameters

event_id (int) is the event identifier returned by the DeviceProxy::subscribe_event(). Unlike in TangoC++ we chech that the event_id has been subscribed in this DeviceProxy.

Return None

Throws *EventSystemFailed*

write(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().write_attribute(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.write_attribute(...):

write_attribute(self, attr_name, value, green_mode=None, wait=True, timeout=None) -> None write_attribute(self, attr_info, value, green_mode=None, wait=True, timeout=None) -> None

Write a single attribute.

Parameters

attr_name (str) The name of the attribute to write.

attr_info (*AttributeInfo*)

value The value. For non SCALAR attributes it may be any sequence of sequences.

green_mode (*GreenMode*) Defaults to the current DeviceProxy GreenMode. (see *get_green_mode()* and *set_green_mode()*).

wait (bool) whether or not to wait for result. If green_mode is *Synchronous*, this parameter is ignored as it always waits for the result. Ignored when green_mode is Synchronous (always waits).

timeout (float) The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when green_mode is Synchronous or wait is False.

Throws *ConnectionFailed*, *CommunicationFailed*, *DeviceUnlocked*, *DevFailed* from device TimeOutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

write_asynch(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().write_attribute_asynch(...)

For convenience, here is the documentation of DeviceProxy.write_attribute_asynch(...):

write_attributes_asynch(self, values) -> int write_attributes_asynch(self, values, callback) -> None

Shortcut to self.write_attributes_asynch([attr_name, value], cb)

New in PyTango 7.0.0

write_read(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().write_read_attribute(self.name(), ...)

For convenience, here is the documentation of DeviceProxy.write_read_attribute(...):

write_read_attribute(self, attr_name, value, extract_as=ExtractAs.Numpy, green_mode=None, wait=True, timeout=None) -> DeviceAttribute

Write then read a single attribute in a single network call. By default (serialisation by device), the execution of this call in the server can't be interrupted by other clients.

Parameters see write_attribute(attr_name, value)

Return A tango.DeviceAttribute object.

Throws *ConnectionFailed*, *CommunicationFailed*, *DeviceUnlocked*, *DevFailed* from device, *WrongData* TimeoutError (green_mode == Futures) If the future didn't finish executing before the given timeout. Timeout (green_mode == Gevent) If the async result didn't finish executing before the given timeout.

New in PyTango 7.0.0

New in version 8.1.0: *green_mode* parameter. *wait* parameter. *timeout* parameter.

write_reply(**args*, ***kwds*)

This method is a simple way to do: self.get_device_proxy().write_attribute_reply(...)

For convenience, here is the documentation of DeviceProxy.write_attribute_reply(...):

write_attribute_reply(self, id) -> None

Check if the answer of an asynchronous write_attribute is arrived (polling model). If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived.

Parameters

id (int) the asynchronous call identifier.

Return None

Throws *AsynCall*, *AsynReplyNotArrived*, *CommunicationFailed*, *DevFailed* from device.

New in PyTango 7.0.0

write_attribute_reply(self, id, timeout) -> None

Check if the answer of an asynchronous write_attribute is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived.

Parameters

id (int) the asynchronous call identifier.

timeout (int) the timeout

Return None

Throws *AsynCall*, *AsynReplyNotArrived*, *CommunicationFailed*, *DevFailed* from device.

New in PyTango 7.0.0

5.2.3 Group

Group class

class tango.**Group**(*name*)
Bases: object

A Tango Group represents a hierarchy of tango devices. The hierarchy may have more than one level. The main goal is to group devices with same attribute(s)/command(s) to be able to do parallel requests.

add(*self*, *subgroup*, *timeout_ms=-1*) → None

Attaches a (sub)_RealGroup.

To remove the subgroup use the remove() method.

Parameters

subgroup (str)

timeout_ms (int) If timeout_ms parameter is different from -1, the client side timeout associated to each device composing the _RealGroup added is set to timeout_ms milliseconds. If timeout_ms is -1, timeouts are not changed.

Return None

Throws TypeError, ArgumentError

command_inout(*self*, *cmd_name*, *forward=True*) → sequence<GroupCmdReply>
command_inout *(self, cmd_name, param, forward=True)* **->** sequence<*GroupCmdReply*>

command_inout *(self, cmd_name, param_list, forward=True)* **->** sequence<*GroupCmdReply*>

Just a shortcut to do: self.command_inout_reply(self.command_inout_asynch(...))

Parameters

cmd_name (str) Command name

param (any) parameter value

param_list (tango.DeviceDataList) sequence of parameters. When given, it's length must match the group size.

forward (bool) If it is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Return (sequence<*GroupCmdReply*>)

read_attribute(*self*, *attr_name*, *forward=True*) → sequence<GroupAttrReply>

Just a shortcut to do: self.read_attribute_reply(self.read_attribute_asynch(...))

read_attributes(*self*, *attr_names*, *forward=True*) → sequence<GroupAttrReply>

Just a shortcut to do: self.read_attributes_reply(self.read_attributes_asynch(...))

write_attribute(*self*, *attr_name*, *value*, *forward=True*, *multi=False*) → sequence<GroupReply>

Just a shortcut to do: self.write_attribute_reply(self.write_attribute_asynch(...))

GroupReply classes

Group member functions do not return the same as their DeviceProxy counterparts, but objects that contain them. This is:

- *write attribute* family returns tango.GroupReplyList
- *read attribute* family returns tango.GroupAttrReplyList
- *command inout* family returns tango.GroupCmdReplyList

The Group*ReplyList objects are just list-like objects containing *GroupReply*, *GroupAttrReply* and *GroupCmdReply* elements that will be described now.

Note also that GroupReply is the base of GroupCmdReply and GroupAttrReply.

class tango.**GroupReply**

This is the base class for the result of an operation on a PyTangoGroup, being it a write attribute, read attribute, or command inout operation.

It has some trivial common operations:

- has_failed(self) -> bool
- group_element_enabled(self) ->bool
- dev_name(self) -> str
- obj_name(self) -> str
- get_err_stack(self) -> DevErrorList

class tango.**GroupAttrReply**

get_data(*self*, *extract_as=ExtractAs.Numpy*) → DeviceAttribute

Get the DeviceAttribute.

Parameters

extract_as (ExtractAs)

Return (*DeviceAttribute*) Whatever is stored there, or None.

class tango.**GroupCmdReply**

get_data(*self*) → any

Get the actual value stored in the GroupCmdRply, the command output value. It's the same as self.get_data_raw().extract()

Parameters None

Return (any) Whatever is stored there, or None.

5.2.4 Green API

Summary:

- *tango.get_green_mode()*
- *tango.set_green_mode()*
- *tango.futures.DeviceProxy()*
- *tango.gevent.DeviceProxy()*

tango.**get_green_mode**()

Returns the current global default PyTango green mode.

Returns the current global default PyTango green mode

Return type *GreenMode*

tango.**set_green_mode**(*green_mode=None*)
Sets the global default PyTango green mode.

Advice: Use only in your final application. Don't use this in a python library in order not to interfere with the beavior of other libraries and/or application where your library is being.

Parameters **green_mode** (GreenMode) – the new global default PyTango green mode

tango.futures.**DeviceProxy**(*self*, *dev_name*, *wait=True*, *timeout=True*) → DeviceProxy
DeviceProxy(self, dev_name, need_check_acc, wait=True, timeout=True) -> DeviceProxy

Creates a *futures* enabled *DeviceProxy*.

The DeviceProxy constructor internally makes some network calls which makes it *slow*. By using the futures *green mode* you are allowing other python code to be executed in a cooperative way.

Note: The timeout parameter has no relation with the tango device client side timeout (gettable by `get_timeout_millis()` and settable through `set_timeout_millis()`)

Parameters

- **dev_name** (*str*) – the device name or alias
- **need_check_acc** (*bool*) – in first version of the function it defaults to True. Determines if at creation time of DeviceProxy it should check for channel access (rarely used)
- **wait** (*bool*) – whether or not to wait for result of creating a DeviceProxy.
- **timeout** (*float*) – The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when wait is False.

Returns

if wait is True: *DeviceProxy*

else: `concurrent.futures.Future`

Throws

- a *DevFailed* if wait is True and there is an error creating the device.
- a *concurrent.futures.TimeoutError* if wait is False, timeout is not None and the time to create the device has expired.

New in PyTango 8.1.0

tango.gevent.**DeviceProxy**(*self*, *dev_name*, *wait=True*, *timeout=True*) → DeviceProxy
DeviceProxy(self, dev_name, need_check_acc, wait=True, timeout=True) -> DeviceProxy

Creates a *gevent* enabled *DeviceProxy*.

The DeviceProxy constructor internally makes some network calls which makes it *slow*. By using the gevent *green mode* you are allowing other python code to be executed in a cooperative way.

Note: The timeout parameter has no relation with the tango device client side timeout (gettable by `get_timeout_millis()` and settable through `set_timeout_millis()`)

Parameters

- **dev_name** (*str*) – the device name or alias
- **need_check_acc** (*bool*) – in first version of the function it defaults to True. Determines if at creation time of DeviceProxy it should check for channel access (rarely used)

- **wait** (*bool*) – whether or not to wait for result of creating a DeviceProxy.
- **timeout** (*float*) – The number of seconds to wait for the result. If None, then there is no limit on the wait time. Ignored when wait is False.

Returns

if wait is True: *DeviceProxy*

else: `gevent.event.AsynchResult`

Throws

- a *DevFailed* if wait is True and there is an error creating the device.
- a *gevent.timeout.Timeout* if wait is False, timeout is not None and the time to create the device has expired.

New in PyTango 8.1.0

5.2.5 API util

class `tango.`**`ApiUtil`**

This class allows you to access the tango syncronization model API. It is designed as a singleton. To get a reference to the singleton object you must do:

```
import tango
apiutil = tango.ApiUtil.instance()
```

New in PyTango 7.1.3

5.2.6 Information classes

See also *Event configuration information*

Attribute

class `tango.`**`AttributeAlarmInfo`**

A structure containing available alarm information for an attribute with the folowing members:

- min_alarm : (`str`) low alarm level
- max_alarm : (`str`) high alarm level
- min_warning : (`str`) low warning level
- max_warning : (`str`) high warning level
- delta_t : (`str`) time delta
- delta_val : (`str`) value delta
- extensions : (`StdStringVector`) extensions (currently not used)

class `tango.`**`AttributeDimension`**

A structure containing x and y attribute data dimensions with the following members:

- dim_x : (`int`) x dimension
- dim_y : (`int`) y dimension

class `tango.`**`AttributeInfo`**

A structure (inheriting from *DeviceAttributeConfig*) containing available information for an attribute with the following members:

- disp_level : (*DispLevel*) display level (OPERATOR, EXPERT)

Inherited members are:

- name : (`str`) attribute name
- writable : (*AttrWriteType*) write type (R, W, RW, R with W)

- data_format : (*AttrDataFormat*) data format (SCALAR, SPECTRUM, IMAGE)
- data_type : (int) attribute type (float, string,..)
- max_dim_x : (int) first dimension of attribute (spectrum or image attributes)
- max_dim_y : (int) second dimension of attribute(image attribute)
- description : (int) attribute description
- label : (str) attribute label (Voltage, time, ...)
- unit : (str) attribute unit (V, ms, ...)
- standard_unit : (str) standard unit
- display_unit : (str) display unit
- format : (str) how to display the attribute value (ex: for floats could be '%6.2f')
- min_value : (str) minimum allowed value
- max_value : (str) maximum allowed value
- min_alarm : (str) low alarm level
- max_alarm : (str) high alarm level
- writable_attr_name : (str) name of the writable attribute
- extensions : (StdStringVector) extensions (currently not used)

class tango.**AttributeInfoEx**

A structure (inheriting from *AttributeInfo*) containing available information for an attribute with the following members:

- alarms : object containing alarm information (see AttributeAlarmInfo).
- events : object containing event information (see AttributeEventInfo).
- sys_extensions : StdStringVector

Inherited members are:

- name : (str) attribute name
- writable : (*AttrWriteType*) write type (R, W, RW, R with W)
- data_format : (*AttrDataFormat*) data format (SCALAR, SPECTRUM, IMAGE)
- data_type : (int) attribute type (float, string,..)
- max_dim_x : (int) first dimension of attribute (spectrum or image attributes)
- max_dim_y : (int) second dimension of attribute(image attribute)
- description : (int) attribute description
- label : (str) attribute label (Voltage, time, ...)
- unit : (str) attribute unit (V, ms, ...)
- standard_unit : (str) standard unit
- display_unit : (str) display unit
- format : (str) how to display the attribute value (ex: for floats could be '%6.2f')
- min_value : (str) minimum allowed value
- max_value : (str) maximum allowed value
- min_alarm : (str) low alarm level
- max_alarm : (str) high alarm level
- writable_attr_name : (str) name of the writable attribute

- extensions : (StdStringVector) extensions (currently not used)
- disp_level : (*DispLevel*) display level (OPERATOR, EXPERT)

see also *AttributeInfo*

class tango.**DeviceAttributeConfig**

A base structure containing available information for an attribute with the following members:

- name : (str) attribute name
- writable : (*AttrWriteType*) write type (R, W, RW, R with W)
- data_format : (*AttrDataFormat*) data format (SCALAR, SPECTRUM, IMAGE)
- data_type : (int) attribute type (float, string,..)
- max_dim_x : (int) first dimension of attribute (spectrum or image attributes)
- max_dim_y : (int) second dimension of attribute(image attribute)
- description : (int) attribute description
- label : (str) attribute label (Voltage, time, ...)
- unit : (str) attribute unit (V, ms, ...)
- standard_unit : (str) standard unit
- display_unit : (str) display unit
- format : (str) how to display the attribute value (ex: for floats could be '%6.2f')
- min_value : (str) minimum allowed value
- max_value : (str) maximum allowed value
- min_alarm : (str) low alarm level
- max_alarm : (str) high alarm level
- writable_attr_name : (str) name of the writable attribute
- extensions : (StdStringVector) extensions (currently not used)

Command

class tango.**DevCommandInfo**

A device command info with the following members:

- cmd_name : (str) command name
- cmd_tag : command as binary value (for TACO)
- in_type : (*CmdArgType*) input type
- out_type : (*CmdArgType*) output type
- in_type_desc : (str) description of input type
- out_type_desc : (str) description of output type

New in PyTango 7.0.0

class tango.**CommandInfo**

A device command info (inheriting from *DevCommandInfo*) with the following members:

- disp_level : (*DispLevel*) command display level

Inherited members are (from *DevCommandInfo*):

- cmd_name : (str) command name
- cmd_tag : (str) command as binary value (for TACO)
- in_type : (*CmdArgType*) input type
- out_type : (*CmdArgType*) output type
- in_type_desc : (str) description of input type
- out_type_desc : (str) description of output type

Other

class tango.**DeviceInfo**

A structure containing available information for a device with the" following members:

- dev_class : (str) device class

- server_id : (str) server ID
- server_host : (str) host name
- server_version : (str) server version
- doc_url : (str) document url

class tango.**LockerInfo**

A structure with information about the locker with the folowing members:

- ll : (*tango.LockerLanguage*) the locker language
- li : (pid_t / UUID) the locker id
- locker_host : (str) the host
- locker_class : (str) the class

pid_t should be an int, UUID should be a tuple of four numbers.

New in PyTango 7.0.0

class tango.**PollDevice**

A structure containing PollDevice information with the folowing members:

- dev_name : (str) device name
- ind_list : (sequence<int>) index list

New in PyTango 7.0.0

5.2.7 Storage classes

Attribute: DeviceAttribute

class tango.**DeviceAttribute**

This is the fundamental type for RECEIVING data from device attributes.

It contains several fields. The most important ones depend on the ExtractAs method used to get the value. Normally they are:

- value : Normal scalar value or numpy array of values.
- w_value : The write part of the attribute.

See other ExtractAs for different possibilities. There are some more fields, these really fixed:

- name : (str)
- data_format : (*AttrDataFormat*) Attribute format
- quality : (*AttrQuality*)
- time : (*TimeVal*)
- dim_x : (int) attribute dimension x
- dim_y : (int) attribute dimension y
- w_dim_x : (int) attribute written dimension x
- w_dim_y : (int) attribute written dimension y
- r_rimension : (tuple) Attribute read dimensions.
- w_dimension : (tuple) Attribute written dimensions.
- nb_read : (int) attribute read total length
- nb_written : (int) attribute written total length

And two methods:

- get_date
- get_err_stack

ExtractAs = <ExtensionMock name='_tango.ExtractAs' id='140222474787808'>

Command: DeviceData

Device data is the type used internally by Tango to deal with command parameters and return values. You don't usually need to deal with it, as command_inout will automatically convert the parameters from any other type and the result value to another type.

You can still use them, using command_inout_raw to get the result in a DeviceData.

You also may deal with it when reading command history.

class tango.**DeviceData**
This is the fundamental type for sending and receiving data from device commands. The values can be inserted and extracted using the insert() and extract() methods.

5.2.8 Callback related classes

If you subscribe a callback in a DeviceProxy, it will be run with a parameter. This parameter depends will be of one of the following classes depending on the callback type.

class tango.**AttrReadEvent**
This class is used to pass data to the callback method in asynchronous callback model for read_attribute(s) execution.
It has the following members:

- device : (*DeviceProxy*) The DeviceProxy object on which the call was executed
- attr_names : (sequence<str>) The attribute name list
- argout : (*DeviceAttribute*) The attribute value
- err : (bool) A boolean flag set to true if the command failed. False otherwise
- errors : (sequence<*DevError*>) The error stack
- ext :

class tango.**AttrWrittenEvent**
This class is used to pass data to the callback method in asynchronous callback model for write_attribute(s) execution
It has the following members:

- device : (*DeviceProxy*) The DeviceProxy object on which the call was executed
- attr_names : (sequence<str>) The attribute name list
- err : (bool) A boolean flag set to true if the command failed. False otherwise
- errors : (*NamedDevFailedList*) The error stack
- ext :

class tango.**CmdDoneEvent**
This class is used to pass data to the callback method in asynchronous callback model for command execution.
It has the following members:

- device : (*DeviceProxy*) The DeviceProxy object on which the call was executed.
- cmd_name : (str) The command name
- argout_raw : (*DeviceData*) The command argout
- argout : The command argout
- err : (bool) A boolean flag set to true if the command failed. False otherwise
- errors : (sequence<*DevError*>) The error stack
- ext :

5.2.9 Event related classes

Event configuration information

class tango.**AttributeEventInfo**
A structure containing available event information for an attribute with the folowing members:
- ch_event : (*ChangeEventInfo*) change event information
- per_event : (*PeriodicEventInfo*) periodic event information
- arch_event : (*ArchiveEventInfo*) archiving event information

class tango.**ArchiveEventInfo**
A structure containing available archiving event information for an attribute with the folowing members:
- archive_rel_change : (str) relative change that will generate an event
- archive_abs_change : (str) absolute change that will generate an event
- archive_period : (str) archive period
- extensions : (sequence<str>) extensions (currently not used)

class tango.**ChangeEventInfo**
A structure containing available change event information for an attribute with the folowing members:
- rel_change : (str) relative change that will generate an event
- abs_change : (str) absolute change that will generate an event
- extensions : (StdStringVector) extensions (currently not used)

class tango.**PeriodicEventInfo**
A structure containing available periodic event information for an attribute with the folowing members:
- period : (str) event period
- extensions : (StdStringVector) extensions (currently not used)

Event arrived structures

class tango.**EventData**
This class is used to pass data to the callback method when an event is sent to the client. It contains the following public fields:
- device : (*DeviceProxy*) The DeviceProxy object on which the call was executed.
- attr_name : (str) The attribute name
- event : (str) The event name
- attr_value : (*DeviceAttribute*) The attribute data (DeviceAttribute)
- err : (bool) A boolean flag set to true if the request failed. False otherwise
- errors : (sequence<*DevError*>) The error stack
- reception_date: (*TimeVal*)

class tango.**AttrConfEventData**
This class is used to pass data to the callback method when a configuration event is sent to the client. It contains the following public fields:
- device : (*DeviceProxy*) The DeviceProxy object on which the call was executed
- attr_name : (str) The attribute name
- event : (str) The event name
- attr_conf : (*AttributeInfoEx*) The attribute data
- err : (bool) A boolean flag set to true if the request failed. False otherwise
- errors : (sequence<*DevError*>) The error stack
- reception_date: (*TimeVal*)

class tango.**DataReadyEventData**
This class is used to pass data to the callback method when an attribute data ready event is sent to the clien. It contains the following public fields:
- device : (*DeviceProxy*) The DeviceProxy object on which the call was executed

- attr_name : (str) The attribute name
- event : (str) The event name
- attr_data_type : (int) The attribute data type
- ctr : (int) The user counter. Set to 0 if not defined when sent by the server
- err : (bool) A boolean flag set to true if the request failed. False otherwise
- errors : (sequence<*DevError*>) The error stack
- reception_date: (*TimeVal*)

New in PyTango 7.0.0

5.2.10 History classes

class tango.**DeviceAttributeHistory**

See *DeviceAttribute*.

class tango.**DeviceDataHistory**

See *DeviceData*.

5.2.11 Enumerations & other classes

Enumerations

class tango.**LockerLanguage**

An enumeration representing the programming language in which the client application who locked is written.

- CPP : C++/Python language
- JAVA : Java language

New in PyTango 7.0.0

class tango.**CmdArgType**

An enumeration representing the command argument type.

- DevVoid
- DevBoolean
- DevShort
- DevLong
- DevFloat
- DevDouble
- DevUShort
- DevULong
- DevString
- DevVarCharArray
- DevVarShortArray
- DevVarLongArray
- DevVarFloatArray
- DevVarDoubleArray
- DevVarUShortArray
- DevVarULongArray
- DevVarStringArray
- DevVarLongStringArray
- DevVarDoubleStringArray
- DevState
- ConstDevString
- DevVarBooleanArray
- DevUChar

- DevLong64
- DevULong64
- DevVarLong64Array
- DevVarULong64Array
- DevInt
- DevEncoded
- DevEnum
- DevPipeBlob

class tango.**MessBoxType**
An enumeration representing the MessBoxType
- STOP
- INFO

New in PyTango 7.0.0

class tango.**PollObjType**
An enumeration representing the PollObjType
- POLL_CMD
- POLL_ATTR
- EVENT_HEARTBEAT
- STORE_SUBDEV

New in PyTango 7.0.0

class tango.**PollCmdCode**
An enumeration representing the PollCmdCode
- POLL_ADD_OBJ
- POLL_REM_OBJ
- POLL_START
- POLL_STOP
- POLL_UPD_PERIOD
- POLL_REM_DEV
- POLL_EXIT
- POLL_REM_EXT_TRIG_OBJ
- POLL_ADD_HEARTBEAT
- POLL_REM_HEARTBEAT

New in PyTango 7.0.0

class tango.**SerialModel**
An enumeration representing the type of serialization performed by the device server
- BY_DEVICE
- BY_CLASS
- BY_PROCESS
- NO_SYNC

class tango.**AttReqType**
An enumeration representing the type of attribute request
- READ_REQ
- WRITE_REQ

class tango.**LockCmdCode**
An enumeration representing the LockCmdCode
- LOCK_ADD_DEV
- LOCK_REM_DEV
- LOCK_UNLOCK_ALL_EXIT
- LOCK_EXIT

New in PyTango 7.0.0

class tango.**LogLevel**
An enumeration representing the LogLevel
- LOG_OFF
- LOG_FATAL
- LOG_ERROR

- LOG_WARN
- LOG_INFO
- LOG_DEBUG

New in PyTango 7.0.0

class tango.**LogTarget**
An enumeration representing the LogTarget
- LOG_CONSOLE
- LOG_FILE
- LOG_DEVICE

New in PyTango 7.0.0

class tango.**EventType**
An enumeration representing event type
- CHANGE_EVENT
- QUALITY_EVENT
- PERIODIC_EVENT
- ARCHIVE_EVENT
- USER_EVENT
- ATTR_CONF_EVENT
- DATA_READY_EVENT

DATA_READY_EVENT - New in PyTango 7.0.0

class tango.**KeepAliveCmdCode**
An enumeration representing the KeepAliveCmdCode
- EXIT_TH

New in PyTango 7.0.0

class tango.**AccessControlType**
An enumeration representing the AccessControlType
- ACCESS_READ
- ACCESS_WRITE

New in PyTango 7.0.0

class tango.**asyn_req_type**
An enumeration representing the asynchronous request type
- POLLING
- CALLBACK
- ALL_ASYNCH

class tango.**cb_sub_model**
An enumeration representing callback sub model
- PUSH_CALLBACK
- PULL_CALLBACK

class tango.**AttrQuality**
An enumeration representing the attribute quality
- ATTR_VALID
- ATTR_INVALID
- ATTR_ALARM
- ATTR_CHANGING
- ATTR_WARNING

class tango.**AttrWriteType**
An enumeration representing the attribute type
- READ
- READ_WITH_WRITE
- WRITE
- READ_WRITE

class tango.**AttrDataFormat**
An enumeration representing the attribute format
- SCALAR
- SPECTRUM
- IMAGE
- FMT_UNKNOWN

class tango.**PipeWriteType**
An enumeration representing the pipe type
- PIPE_READ
- PIPE_READ_WRITE

class tango.**DevSource**
An enumeration representing the device source for data
- DEV
- CACHE
- CACHE_DEV

class tango.**ErrSeverity**
An enumeration representing the error severity
- WARN
- ERR
- PANIC

class tango.**DevState**
An enumeration representing the device state
- ON
- OFF
- CLOSE
- OPEN
- INSERT
- EXTRACT
- MOVING
- STANDBY
- FAULT
- INIT
- RUNNING
- ALARM
- DISABLE
- UNKNOWN

class tango.**DispLevel**
An enumeration representing the display level
- OPERATOR
- EXPERT

class tango.**GreenMode**
An enumeration representing the GreenMode
- Synchronous
- Futures
- Gevent

New in PyTango 8.1.0

Other classes

class tango.**Release**
Release information:

- name : (str) package name

- version_info : (tuple) The five components of the version number: major, minor, micro, releaselevel, and serial.
- version : (str) package version in format <major>.<minor>.<micro>
- version_long : (str) package version in format <major>.<minor>.<micro><releaselevel><serial>
- version_description : (str) short description for the current version
- version_number : (int) <major>*100 + <minor>*10 + <micro>
- description : (str) package description
- long_description : (str) longer package description
- authors : (dict<str(last name), tuple<str(full name),str(email)>>) package authors
- url : (str) package url
- download_url : (str) package download url
- platform : (seq) list of available platforms
- keywords : (seq) list of keywords
- license : (str) the license

class tango.**TimeVal**

Time value structure with the following members:

- tv_sec : seconds
- tv_usec : microseconds
- tv_nsec : nanoseconds

isoformat (*self*, *sep='T'*) → str

Returns a string in ISO 8601 format, YYYY-MM-DDTHH:MM:SS[.mmmmmm][+HH:MM]

Parameters sep : (str) sep is used to separate the year from the time, and defaults to 'T'

Return (str) a string representing the time according to a format specification.

New in version 7.1.0.

New in version 7.1.2: Documented

Changed in version 7.1.2: The *sep* parameter is not mandatory anymore and defaults to 'T' (same as datetime.datetime.isoformat())

strftime (*self*, *format*) → str

Convert a time value to a string according to a format specification.

Parameters format : (str) See the python library reference manual for formatting codes

Return (str) a string representing the time according to a format specification.

New in version 7.1.0.

New in version 7.1.2: Documented

todatetime (*self*) → datetime.datetime

Returns a datetime.datetime object representing the same time value

Parameters None

Return (datetime.datetime) the time value in datetime format

New in version 7.1.0.

totime (*self*) → float

Returns a float representing this time value

Parameters None

Return a float representing the time value

New in version 7.1.0.

5.3 Server API

5.3.1 High level server API

Server helper classes for writing Tango device servers.

- *Device*
- *attribute*
- *command*
- *pipe*
- *device_property*
- *class_property*
- *run()*
- *server_run()*

This module provides a high level device server API. It implements *TEP1*. It exposes an easier API for developing a Tango device server.

Here is a simple example on how to write a *Clock* device server using the high level API:

```
import time
from tango.server import run
from tango.server import Device, DeviceMeta
from tango.server import attribute, command


class Clock(Device):
    __metaclass__ = DeviceMeta

    time = attribute()

    def read_time(self):
        return time.time()

    @command(din_type=str, dout_type=str)
    def strftime(self, format):
        return time.strftime(format)


if __name__ == "__main__":
    run((Clock,))
```

Here is a more complete example on how to write a *PowerSupply* device server using the high level API. The example contains:

1. a read-only double scalar attribute called *voltage*
2. a read/write double scalar expert attribute *current*
3. a read-only double image attribute called *noise*
4. a *ramp* command

5. a *host* device property
6. a *port* class property

```
from time import time
from numpy.random import random_sample

from tango import AttrQuality, AttrWriteType, DispLevel, server_run
from tango.server import Device, DeviceMeta, attribute, command
from tango.server import class_property, device_property

class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    voltage = attribute()

    current = attribute(label="Current", dtype=float,
                        display_level=DispLevel.EXPERT,
                        access=AttrWriteType.READ_WRITE,
                        unit="A", format="8.4f",
                        min_value=0.0, max_value=8.5,
                        min_alarm=0.1, max_alarm=8.4,
                        min_warning=0.5, max_warning=8.0,
                        fget="get_current", fset="set_current",
                        doc="the power supply current")

    noise = attribute(label="Noise", dtype=((float,),),
                      max_dim_x=1024, max_dim_y=1024,
                      fget="get_noise")

    host = device_property(dtype=str)
    port = class_property(dtype=int, default_value=9788)

    def read_voltage(self):
        self.info_stream("get voltage(%s, %d)" % (self.host, self.port))
        return 10.0

    def get_current(self):
        return 2.3456, time(), AttrQuality.ATTR_WARNING

    def set_current(self, current):
        print("Current set to %f" % current)

    def get_noise(self):
        return random_sample((1024, 1024))

    @command(dtype_in=float)
    def ramp(self, value):
        print("Ramping up...")

if __name__ == "__main__":
    server_run((PowerSupply,))
```

Pretty cool, uh?

Note: the `__metaclass__` statement is mandatory due to a limitation in the *boost-python* library used by PyTango.

If you are using python 3 you can write instead:

```
class PowerSupply(Device, metaclass=DeviceMeta)
    pass
```

Data types

When declaring attributes, properties or commands, one of the most important information is the data type. It is given by the keyword argument *dtype*. In order to provide a more *pythonic* interface, this argument is not restricted to the *CmdArgType* options.

For example, to define a *SCALAR* `DevLong` attribute you have several possibilities:

1. `int`
2. 'int'
3. 'int32'
4. 'integer'
5. `tango.CmdArgType.DevLong`
6. 'DevLong'
7. `numpy.int32`

To define a *SPECTRUM* attribute simply wrap the scalar data type in any python sequence:

- using a *tuple*: `(:obj:`int`,)` or
- using a *list*: `[:obj:`int`]` or
- any other sequence type

To define an *IMAGE* attribute simply wrap the scalar data type in any python sequence of sequences:

- using a *tuple*: `((:obj:`int`,),)` or
- using a *list*: `[[:obj:`int`]]` or
- any other sequence type

Below is the complete table of equivalences.

dtype argument	converts to tango type
None	DevVoid
'None'	DevVoid
DevVoid	DevVoid
'DevVoid'	DevVoid
DevState	DevState
'DevState'	DevState
bool	DevBoolean
'bool'	DevBoolean
'boolean'	DevBoolean
DevBoolean	DevBoolean
'DevBoolean'	DevBoolean
numpy.bool_	DevBoolean
'char'	DevUChar
'chr'	DevUChar
'byte'	DevUChar
chr	DevUChar
DevUChar	DevUChar
'DevUChar'	DevUChar
numpy.uint8	DevUChar
'int16'	DevShort
DevShort	DevShort

Continued on next page

Table 5.2 – continued from previous page

dtype argument	converts to tango type
'DevShort'	DevShort
numpy.int16	DevShort
'uint16'	DevUShort
DevUShort	DevUShort
'DevUShort'	DevUShort
numpy.uint16	DevUShort
int	DevLong
'int'	DevLong
'int32'	DevLong
DevLong	DevLong
'DevLong'	DevLong
numpy.int32	DevLong
'uint'	DevULong
'uint32'	DevULong
DevULong	DevULong
'DevULong'	DevULong
numpy.uint32	DevULong
'int64'	DevLong64
DevLong64	DevLong64
'DevLong64'	DevLong64
numpy.int64	DevLong64
'uint64'	DevULong64
DevULong64	DevULong64
'DevULong64'	DevULong64
numpy.uint64	DevULong64
DevInt	DevInt
'DevInt'	DevInt
'float32'	DevFloat
DevFloat	DevFloat
'DevFloat'	DevFloat
numpy.float32	DevFloat
float	DevDouble
'double'	DevDouble
'float'	DevDouble
'float64'	DevDouble
DevDouble	DevDouble
'DevDouble'	DevDouble
numpy.float64	DevDouble
str	DevString
'str'	DevString
'string'	DevString
'text'	DevString
DevString	DevString
'DevString'	DevString
bytearray	DevEncoded
'bytearray'	DevEncoded
'bytes'	DevEncoded
DevEncoded	DevEncoded
'DevEncoded'	DevEncoded
DevVarBooleanArray	DevVarBooleanArray
'DevVarBooleanArray'	DevVarBooleanArray
DevVarCharArray	DevVarCharArray
'DevVarCharArray'	DevVarCharArray

Continued on next page

Table 5.2 – continued from previous page

dtype argument	converts to tango type
DevVarShortArray	DevVarShortArray
'DevVarShortArray'	DevVarShortArray
DevVarLongArray	DevVarLongArray
'DevVarLongArray'	DevVarLongArray
DevVarLong64Array	DevVarLong64Array
'DevVarLong64Array'	DevVarLong64Array
DevVarULong64Array	DevVarULong64Array
'DevVarULong64Array'	DevVarULong64Array
DevVarFloatArray	DevVarFloatArray
'DevVarFloatArray'	DevVarFloatArray
DevVarDoubleArray	DevVarDoubleArray
'DevVarDoubleArray'	DevVarDoubleArray
DevVarUShortArray	DevVarUShortArray
'DevVarUShortArray'	DevVarUShortArray
DevVarULongArray	DevVarULongArray
'DevVarULongArray'	DevVarULongArray
DevVarStringArray	DevVarStringArray
'DevVarStringArray'	DevVarStringArray
DevVarLongStringArray	DevVarLongStringArray
'DevVarLongStringArray'	DevVarLongStringArray
DevVarDoubleStringArray	DevVarDoubleStringArray
'DevVarDoubleStringArray'	DevVarDoubleStringArray
DevPipeBlob	DevPipeBlob
'DevPipeBlob'	DevPipeBlob

tango.server.**Device**

class tango.server.**attribute**(*fget=None*, ***kwargs*)

Declares a new tango attribute in a *Device*. To be used like the python native property function. For example, to declare a scalar, *tango.DevDouble*, read-only attribute called *voltage* in a *PowerSupply* *Device* do:

```
class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    voltage = attribute()

    def read_voltage(self):
        return 999.999
```

The same can be achieved with:

```
class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    @attribute
    def voltage(self):
        return 999.999
```

It receives multiple keyword arguments.

parameter	type	default value	description
name	str	class member name	alternative attribute name
dtype	object	DevDouble	data type (see *Data type equivalence*)

Table 5.3 – continued from previous page

parameter	type	default value	description
dformat	*AttrDataFormat*	SCALAR	data format
max_dim_x	int	1	maximum size for x dimension (ignored for
max_dim_y	int	0	maximum size for y dimension (ignored for
display_level	*DispLevel*	OPERATOR	display level
polling_period	int	-1	polling period
memorized	bool	False	attribute should or not be memorized
hw_memorized	bool	False	write method should be called at startup wh
access	*AttrWriteType*	READ	read only/ read write / write only access
fget (or fread)	str or callable	'read_<attr_name>'	read method name or method object
fset (or fwrite)	str or callable	'write_<attr_name>'	write method name or method object
is_allowed	str or callable	'is_<attr_name>_allowed'	is allowed method name or method object
label	str	'<attr_name>'	attribute label
enum_labels	sequence	None	the list of enumeration labels (enum data ty
doc (or description)	str	''	attribute description
unit	str	''	physical units the attribute value is in
standard_unit	str	''	physical standard unit
display_unit	str	''	physical display unit (hint for clients)
format	str	'6.2f'	attribute representation format
min_value	str	None	minimum allowed value
max_value	str	None	maximum allowed value
min_alarm	str	None	minimum value to trigger attribute alarm
max_alarm	str	None	maximum value to trigger attribute alarm
min_warning	str	None	minimum value to trigger attribute warning
max_warning	str	None	maximum value to trigger attribute warning
delta_val	str	None	
delta_t	str	None	
abs_change	str	None	minimum value change between events that
rel_change	str	None	minimum relative change between events th
period	str	None	
archive_abs_change	str	None	
archive_rel_change	str	None	
archive_period	str	None	
green_mode	*GreenMode*	None	green mode for read and write. None means
read_green_mode	*GreenMode*	None	green mode for read. None means use serve
write_green_mode	*GreenMode*	None	green mode for write. None means use serv

Note: avoid using *dformat* parameter. If you need a SPECTRUM attribute of say, boolean type, use instead `dtype=(bool,)`.

Example of a integer writable attribute with a customized label, unit and description:

```
class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    current = attribute(label="Current", unit="mA", dtype=int,
                        access=AttrWriteType.READ_WRITE,
                        doc="the power supply current")

    def init_device(self):
        Device.init_device(self)
        self._current = -1

    def read_current(self):
        return self._current
```

```
        def write_current(self, current):
            self._current = current
```

The same, but using attribute as a decorator:

```
    class PowerSupply(Device):
        __metaclass__ = DeviceMeta

        def init_device(self):
            Device.init_device(self)
            self._current = -1

        @attribute(label="Current", unit="mA", dtype=int)
        def current(self):
            """the power supply current"""
            return 999.999

        @current.write
        def current(self, current):
            self._current = current
```

In this second format, defining the *write* implicitly sets the attribute access to READ_WRITE.

New in version 8.1.7: added green_mode, read_green_mode and write_green_mode options

tango.server.**command**(*f=None, dtype_in=None, dformat_in=None, doc_in='', dtype_out=None, dformat_out=None, doc_out='', display_level=None, polling_period=None, green_mode=None*)

Declares a new tango command in a *Device*. To be used like a decorator in the methods you want to declare as tango commands. The following example declares commands:

- *void TurnOn(void)*
- *void Ramp(DevDouble current)*
- *DevBool Pressurize(DevDouble pressure)*

```
    class PowerSupply(Device):
        __metaclass__ = DeviceMeta

        @command
        def TurnOn(self):
            self.info_stream('Turning on the power supply')

        @command(dtype_in=float)
        def Ramp(self, current):
            self.info_stream('Ramping on %f...' % current)

        @command(dtype_in=float, doc_in='the pressure to be set',
                 dtype_out=bool, doc_out='True if it worked, False otherwise')
        def Pressurize(self, pressure):
            self.info_stream('Pressurizing to %f...' % pressure)
            return True
```

Note: avoid using *dformat* parameter. If you need a SPECTRUM attribute of say, boolean type, use instead `dtype=(bool,)`.

Parameters

- **dtype_in** – a *data type* describing the type of parameter. Default is None meaning no parameter.
- **dformat_in** (AttrDataFormat) – parameter data format. Default is None.

- **doc_in** (*str*) – parameter documentation
- **dtype_out** – a *data type* describing the type of return value. Default is None meaning no return value.
- **dformat_out** (AttrDataFormat) – return value data format. Default is None.
- **doc_out** (*str*) – return value documentation
- **display_level** (DispLevel) – display level for the command (optional)
- **polling_period** (*int*) – polling period in milliseconds (optional)
- **green_mode** – set green mode on this specific command. Default value is None meaning use the server green mode. Set it to GreenMode.Synchronous to force a non green command in a green server.

New in version 8.1.7: added green_mode option

New in version 9.2.0: added display_level and polling_period optional argument

class tango.server.**pipe**(*fget=None*, ***kwargs*)

Declares a new tango pipe in a *Device*. To be used like the python native property function.

Checkout the *pipe data types* to see what you should return on a pipe read request and what to expect as argument on a pipe write request.

For example, to declare a read-only pipe called *ROI* (for Region Of Interest), in a *Detector* *Device* do:

```
class Detector(Device):
    __metaclass__ = DeviceMeta

    ROI = pipe()

    def read_ROI(self):
        return ('ROI', ({'name': 'x', 'value': 0},
                        {'name': 'y', 'value': 10},
                        {'name': 'width', 'value': 100},
                        {'name': 'height', 'value': 200}))
```

The same can be achieved with (also showing that a dict can be used to pass blob data):

```
class Detector(Device):
    __metaclass__ = DeviceMeta

    @pipe
    def ROI(self):
        return 'ROI', dict(x=0, y=10, width=100, height=200)
```

It receives multiple keyword arguments.

parameter	type	default value	description
name	str	class member name	alternative pipe name
display_level	DispLevel	OPERATOR	display level
access	PipeWriteType	READ	read only/ read write access
fget (or fread)	str or callable	'read_<pipe_name>'	read method name or method object
fset (or fwrite)	str or callable	'write_<pipe_name>'	write method name or method object
is_allowed	str or callable	'is_<pipe_name>_allowed'	is allowed method name or method object
label	str	'<pipe_name>'	pipe label
doc (or description)	str	''	pipe description
green_mode	GreenMode	None	green mode for read and write. None means use server green mode.
read_green_mode	GreenMode	None	green mode for read. None means use server green mode.
write_green_mode	GreenMode	None	green mode for write. None means use server green mode.

The same example with a read-write ROI, a customized label and description:

```
class Detector(Device):
    __metaclass__ = DeviceMeta

    ROI = pipe(label='Region Of Interest', doc='The active region of interest',
               access=PipeWriteType.PIPE_READ_WRITE)

    def init_device(self):
        Device.init_device(self)
        self.__roi = 'ROI', dict(x=0, y=10, width=100, height=200)

    def read_ROI(self):
        return self.__roi

    def write_ROI(self, roi):
        self.__roi = roi
```

The same, but using pipe as a decorator:

```
class Detector(Device):
    __metaclass__ = DeviceMeta

    def init_device(self):
        Device.init_device(self)
        self.__roi = 'ROI', dict(x=0, y=10, width=100, height=200)

    @pipe(label="Region Of Interest")
    def ROI(self):
        """The active region of interest"""
        return self.__roi

    @ROI.write
    def ROI(self, roi):
        self.__roi = roi
```

In this second format, defining the *write / setter* implicitly sets the pipe access to READ_WRITE.

New in version 9.2.0.

class tango.server.**device_property**(*dtype*, *doc=''*, *default_value=None*, *update_db=False*)
Declares a new tango device property in a *Device*. To be used like the python native property function. For example, to declare a scalar, *tango.DevString*, device property called *host* in a *PowerSupply* *Device* do:

```
from tango.server import Device, DeviceMeta
from tango.server import device_property

class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    host = device_property(dtype=str)
```

Parameters

- **dtype** – Data type (see *Data types*)
- **doc** – property documentation (optional)
- **default_value** – default value for the property (optional)
- **update_db** (*bool*) – tells if set value should write the value to database. [default: False]

New in version 8.1.7: added update_db option

class tango.server.**class_property**(*dtype*, *doc=''*, *default_value=None*, *update_db=False*)
Declares a new tango class property in a *Device*. To be used like the python native property function. For example, to declare a scalar, *tango.DevString*, class property called *port* in a *PowerSupply* *Device* do:

```
from tango.server import Device, DeviceMeta
from tango.server import class_property

class PowerSupply(Device):
    __metaclass__ = DeviceMeta

    port = class_property(dtype=int, default_value=9788)
```

Parameters

- **dtype** – Data type (see *Data types*)
- **doc** – property documentation (optional)
- **default_value** – default value for the property (optional)
- **update_db** (*bool*) – tells if set value should write the value to database. [default: False]

New in version 8.1.7: added update_db option

tango.server.**run**(*classes*, *args=None*, *msg_stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *verbose=False*, *util=None*, *event_loop=None*, *post_init_callback=None*, *green_mode=None*)
Provides a simple way to run a tango server. It handles exceptions by writting a message to the msg_stream.

The *classes* parameter can be either a sequence of:
•: class:*~tango.server.Device* or
•a sequence of two elements *DeviceClass*, *DeviceImpl* or
•a sequence of three elements *DeviceClass*, *DeviceImpl*, tango class name (str)
or a dictionary where:
•key is the tango class name
•**value is either:**

– a : class:*~tango.server.Device* class or

– a sequence of two elements *DeviceClass*, *DeviceImpl* or

– a sequence of three elements *DeviceClass*, *DeviceImpl*, tango class name (str)

The optional *post_init_callback* can be a callable (without arguments) or a tuple where the first element is the callable, the second is a list of arguments (optional) and the third is a dictionary of keyword arguments (also optional).

Note: the order of registration of tango classes defines the order tango uses to initialize the corresponding devices. if using a dictionary as argument for classes be aware that the order of registration becomes arbitrary. If you need a predefined order use a sequence or an OrderedDict.

Example 1: registering and running a PowerSupply inheriting from *Device*:

```
from tango.server import Device, DeviceMeta, run

class PowerSupply(Device):
    __metaclass__ = DeviceMeta

run((PowerSupply,))
```

Example 2: registering and running a MyServer defined by tango classes *MyServerClass* and *MyServer*:

```
from tango import Device_4Impl, DeviceClass
from tango.server import run

class MyServer(Device_4Impl):
    pass

class MyServerClass(DeviceClass):
    pass

run({'MyServer': (MyServerClass, MyServer)})
```

Example 3: registering and running a MyServer defined by tango classes *MyServerClass* and *MyServer*:

```
from tango import Device_4Impl, DeviceClass
from tango.server import Device, DeviceMeta, run

class PowerSupply(Device):
    __metaclass__ = DeviceMeta

class MyServer(Device_4Impl):
    pass

class MyServerClass(DeviceClass):
    pass

run([PowerSupply, [MyServerClass, MyServer]])
# or: run({'MyServer': (MyServerClass, MyServer)})
```

Parameters

- **classes** (*sequence or dict*) – a sequence of *Device* classes or a dictionary where keyword is the tango class name and value is a sequence of Tango Device Class python class, and Tango Device python class

- **args** (*list*) – list of command line arguments [default: None, meaning use sys.argv]
- **msg_stream** – stream where to put messages [default: sys.stdout]
- **util** (*Util*) – PyTango Util object [default: None meaning create a Util instance]
- **event_loop** (*callable*) – event_loop callable
- **post_init_callback** (*callable or tuple (see description above)*) – an optional callback that is executed between the calls Util.server_init and Util.server_run

Returns The Util singleton object

Return type *Util*

New in version 8.1.2.

Changed in version 8.1.4: when classes argument is a sequence, the items can also be a sequence <TangoClass, TangoClassClass>[, tango class name]

tango.server.**server_run**(*classes*, *args=None*, *msg_stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *verbose=False*, *util=None*, *event_loop=None*, *post_init_callback=None*, *green_mode=None*)

Since PyTango 8.1.2 it is just an alias to *run()*. Use *run()* instead.

New in version 8.0.0.

Changed in version 8.0.3: Added *util* keyword parameter. Returns util object

Changed in version 8.1.1: Changed default msg_stream from *stderr* to *stdout* Added *event_loop* keyword parameter. Returns util object

Changed in version 8.1.2: Added *post_init_callback* keyword parameter

Deprecated since version 8.1.2: Use *run()* instead.

5.3.2 Device

DeviceImpl

class tango.**DeviceImpl**

Base class for all TANGO device. This class inherits from CORBA classes where all the network layer is implemented.

add_attribute(*self*, *attr*, *r_meth=None*, *w_meth=None*, *is_allo_meth=None*) → Attr

Add a new attribute to the device attribute list. Please, note that if you add an attribute to a device at device creation time, this attribute will be added to the device class attribute list. Therefore, all devices belonging to the same class created after this attribute addition will also have this attribute.

Parameters

attr (Attr or AttrData) the new attribute to be added to the list.

r_meth (callable) the read method to be called on a read request

w_meth (callable) the write method to be called on a write request (if attr is writable)

is_allo_meth (callable) the method that is called to check if it is possible to access the attribute or not

Return (*Attr*) the newly created attribute.

Throws *DevFailed*

debug_stream(*self*, *msg*, **args*) → None

Sends the given message to the tango debug stream.

Since PyTango 7.1.3, the same can be achieved with:

```
print(msg, file=self.log_debug)
```

Parameters

msg (str) the message to be sent to the debug stream

Return None

error_stream(*self*, *msg*, **args*) → None

Sends the given message to the tango error stream.

Since PyTango 7.1.3, the same can be achieved with:

```
print(msg, file=self.log_error)
```

Parameters

msg (str) the message to be sent to the error stream

Return None

fatal_stream(*self*, *msg*, **args*) → None

Sends the given message to the tango fatal stream.

Since PyTango 7.1.3, the same can be achieved with:

```
print(msg, file=self.log_fatal)
```

Parameters

msg (str) the message to be sent to the fatal stream

Return None

get_device_properties(*self*, *ds_class = None*) → None

Utility method that fetches all the device properties from the database and converts them into members of this DeviceImpl.

Parameters

ds_class (*DeviceClass*) the DeviceClass object. Optional. Default value is None meaning that the corresponding DeviceClass object for this DeviceImpl will be used

Return None

Throws *DevFailed*

info_stream(*self*, *msg*, **args*) → None

Sends the given message to the tango info stream.

Since PyTango 7.1.3, the same can be achieved with:

```
print(msg, file=self.log_info)
```

Parameters

msg (str) the message to be sent to the info stream

Return None

remove_attribute(*self*, *attr_name*) → None

Remove one attribute from the device attribute list.

Parameters

attr_name (str) attribute name

Return None

Throws *DevFailed*

warn_stream(*self*, *msg*, **args*) → None

Sends the given message to the tango warn stream.

Since PyTango 7.1.3, the same can be achieved with:

```
print(msg, file=self.log_warn)
```

Parameters

msg (str) the message to be sent to the warn stream

Return None

Device_2Impl

class tango.**Device_2Impl**

Device_3Impl

class tango.**Device_3Impl**

Device_4Impl

class tango.**Device_4Impl**

DServer

class tango.**DServer**

5.3.3 DeviceClass

class tango.**DeviceClass**

Base class for all TANGO device-class class. A TANGO device-class class is a class where is stored all data/method common to all devices of a TANGO device class

create_device(*self*, *device_name*, *alias=None*, *cb=None*) → None

Creates a new device of the given class in the database, creates a new DeviceImpl for it and calls init_device (just like it is done for existing devices when the DS starts up)

An optional parameter callback is called AFTER the device is registered in the database and BEFORE the init_device for the newly created device is called

Throws tango.DevFailed:

- the device name exists already or
- the given class is not registered for this DS.
- the cb is not a callable

New in PyTango 7.1.2

Parameters

device_name (str) the device name

alias (str) optional alias. Default value is None meaning do not create device alias

cb (callable) a callback that is called AFTER the device is registered in the database and BEFORE the init_device for the newly created device is called. Typically you may want to put device and/or attribute properties in the database here. The callback must receive a parameter: device name (str). Default value is None meaning no callback

Return None

delete_device(*self*, *klass_name*, *device_name*) → None

Deletes an existing device from the database and from this running server

Throws tango.DevFailed:

- the device name doesn't exist in the database
- the device name doesn't exist in this DS.

New in PyTango 7.1.2

Parameters

klass_name (str) the device class name

device_name (str) the device name

Return None

device_destroyer(*name*)
for internal usage only

device_factory(*device_list*)
for internal usage only

device_name_factory(*self*, *dev_name_list*) → None

Create device(s) name list (for no database device server). This method can be re-defined in DeviceClass sub-class for device server started without database. Its rule is to initialise class device name. The default method does nothing.

Parameters

dev_name_list (seq) sequence of devices to be filled

Return None

dyn_attr (*self*, *device_list*) → None

Default implementation does not do anything Overwrite in order to provide dynamic attributes

Parameters

device_list (*seq*) sequence of devices of this class

Return None

5.3.4 Logging decorators

LogIt

class tango.**LogIt**(*show_args=False*, *show_kwargs=False*, *show_ret=False*)

A class designed to be a decorator of any method of a *tango.DeviceImpl* subclass. The idea is to log the entrance and exit of any decorated method.

Example:

```
class MyDevice(tango.Device_4Impl):

    @tango.LogIt()
    def read_Current(self, attr):
        attr.set_value(self._current, 1)
```

All log messages generated by this class have DEBUG level. If you whish to have different log level messages, you should implement subclasses that log to those levels. See, for example, *tango.InfoIt*.

The constructor receives three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs - shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

DebugIt

class tango.**DebugIt**(*show_args=False*, *show_kwargs=False*, *show_ret=False*)

A class designed to be a decorator of any method of a *tango.DeviceImpl* subclass. The idea is to log the entrance and exit of any decorated method as DEBUG level records.

Example:

```
class MyDevice(tango.Device_4Impl):

    @tango.DebugIt()
    def read_Current(self, attr):
        attr.set_value(self._current, 1)
```

All log messages generated by this class have DEBUG level.

The constructor receives three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs - shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

InfoIt

class tango.**InfoIt**(*show_args=False, show_kwargs=False, show_ret=False*)
A class designed to be a decorator of any method of a *tango.DeviceImpl* subclass. The idea is to log the entrance and exit of any decorated method as INFO level records.

Example:

```
class MyDevice(tango.Device_4Impl):

    @tango.InfoIt()
    def read_Current(self, attr):
        attr.set_value(self._current, 1)
```

All log messages generated by this class have INFO level.
The constructor receives three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs - shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

WarnIt

class tango.**WarnIt**(*show_args=False, show_kwargs=False, show_ret=False*)
A class designed to be a decorator of any method of a *tango.DeviceImpl* subclass. The idea is to log the entrance and exit of any decorated method as WARN level records.

Example:

```
class MyDevice(tango.Device_4Impl):

    @tango.WarnIt()
    def read_Current(self, attr):
        attr.set_value(self._current, 1)
```

All log messages generated by this class have WARN level.
The constructor receives three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs - shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

ErrorIt

class tango.**ErrorIt**(*show_args=False, show_kwargs=False, show_ret=False*)
A class designed to be a decorator of any method of a *tango.DeviceImpl* subclass. The idea is to log the entrance and exit of any decorated method as ERROR level records.

Example:

```
class MyDevice(tango.Device_4Impl):

    @tango.ErrorIt()
    def read_Current(self, attr):
        attr.set_value(self._current, 1)
```

All log messages generated by this class have ERROR level.

The constructor receives three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs - shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

FatalIt

class tango.**FatalIt**(*show_args=False*, *show_kwargs=False*, *show_ret=False*)
A class designed to be a decorator of any method of a *tango.DeviceImpl* subclass. The idea is to log the entrance and exit of any decorated method as FATAL level records.

Example:

```
class MyDevice(tango.Device_4Impl):

    @tango.FatalIt()
    def read_Current(self, attr):
        attr.set_value(self._current, 1)
```

All log messages generated by this class have FATAL level.
The constructor receives three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs - shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

5.3.5 Attribute classes

Attr

class tango.**Attr**
This class represents a Tango writable attribute.

Attribute

class tango.**Attribute**
This class represents a Tango attribute.

get_properties(*self*, *attr_cfg = None*) → AttributeConfig

Get attribute properties.

Parameters

conf the config object to be filled with the attribute configuration. Default is None meaning the method will create internally a new AttributeConfig_5 and return it. Can be AttributeConfig, AttributeConfig_2, AttributeConfig_3, AttributeConfig_5 or MultiAttrProp

Return (AttributeConfig) the config object filled with attribute configuration information

New in PyTango 7.1.4

set_properties(*self*, *attr_cfg*, *dev*) → None

Set attribute properties.

This method sets the attribute properties value with the content of the fileds in the AttributeConfig/ AttributeConfig_3 object

Parameters

conf (AttributeConfig or AttributeConfig_3) the config object.

dev (*DeviceImpl*) the device (not used, maintained for backward compatibility)

New in PyTango 7.1.4

WAttribute

class tango.**WAttribute**
This class represents a Tango writable attribute.

MultiAttribute

class tango.**MultiAttribute**
There is one instance of this class for each device. This class is mainly an aggregate of *Attribute* or *WAttribute* objects. It eases management of multiple attributes

UserDefaultAttrProp

class tango.**UserDefaultAttrProp**
User class to set attribute default properties. This class is used to set attribute default properties. Three levels of attributes properties setting are implemented within Tango. The highest property setting level is the database. Then the user default (set using this UserDefaultAttrProp class) and finally a Tango library default value

set_enum_labels(*self*, *enum_labels*) → None

Set default enumeration labels.

Parameters

enum_labels (seq) list of enumeration labels

New in PyTango 9.2.0

5.3.6 Util

class tango.**Util**
This class is a used to store TANGO device server process data and to provide the user with a set of utilities method.

This class is implemented using the singleton design pattern. Therefore a device server process can have only one instance of this class and its constructor is not public. Example:

```
util = tango.Util.instance()
    print(util.get_host_name())
```

add_Cpp_TgClass (*device_class_name*, *tango_device_class_name*)
Register a new C++ tango class.

If there is a shared library file called MotorClass.so which contains a MotorClass class and a _create_MotorClass_class method. Example:

```
util.add_Cpp_TgClass('MotorClass', 'Motor')
```

Note: the parameter 'device_class_name' must match the shared library name.

Deprecated since version 7.1.2: Use *tango.Util.add_class()* instead.

add_TgClass (*klass_device_class*, *klass_device*, *device_class_name=None*)
Register a new python tango class. Example:

```
util.add_TgClass(MotorClass, Motor)
util.add_TgClass(MotorClass, Motor, 'Motor') # equivalent to previous line
```

Deprecated since version 7.1.2: Use *tango.Util.add_class()* instead.

add_class (*self*, *class<DeviceClass>*, *class<DeviceImpl>*, *language="python"*) → None

Register a new tango class ('python' or 'c++').

If language is 'python' then args must be the same as *tango.Util.add_TgClass()*. Otherwise, args should be the ones in *tango.Util.add_Cpp_TgClass()*. Example:

```
util.add_class(MotorClass, Motor)
util.add_class('CounterClass', 'Counter', language='c++')
```

New in PyTango 7.1.2

create_device (*self*, *klass_name*, *device_name*, *alias=None*, *cb=None*) → None

Creates a new device of the given class in the database, creates a new DeviceImpl for it and calls init_device (just like it is done for existing devices when the DS starts up)

An optional parameter callback is called AFTER the device is registered in the database and BEFORE the init_device for the newly created device is called

Throws tango.DevFailed:

- the device name exists already or
- the given class is not registered for this DS.
- the cb is not a callable

New in PyTango 7.1.2

Parameters

klass_name (str) the device class name

device_name (str) the device name

alias (str) optional alias. Default value is None meaning do not create device alias

cb (callable) a callback that is called AFTER the device is registered in the database and BEFORE the init_device for the newly created device is called. Typically you may want to put device and/or attribute properties in the database here. The callback must receive a parameter: device name (str). Default value is None meaning no callback

Return None

delete_device (*self*, *klass_name*, *device_name*) → None

Deletes an existing device from the database and from this running server

Throws tango.DevFailed:

- the device name doesn't exist in the database
- the device name doesn't exist in this DS.

New in PyTango 7.1.2

Parameters

klass_name (str) the device class name

device_name (str) the device name

Return None

get_class_list (*self*) → seq<DeviceClass>

Returns a list of objects of inheriting from DeviceClass

Parameters None

Return (*seq*) a list of objects of inheriting from DeviceClass

5.4 Database API

class tango.**Database**

Database is the high level Tango object which contains the link to the static database. Database provides methods for all database commands : get_device_property(), put_device_property(), info(), etc.. To create a Database, use the default constructor. Example:

```
db = Database()
```

The constructor uses the TANGO_HOST env. variable to determine which instance of the Database to connect to.

add_server (*self*, *servname*, *dev_info*, *with_dserver=False*) → None

Add a (group of) devices to the database. This is considered as a low level call because it may render the database inconsistent if it is not used properly.

If *with_dserver* parameter is set to False (default), this call will only register the given dev_info(s). You should include in the list of dev_info an entry to the usually hidden **DServer** device.

If *with_dserver* parameter is set to True, the call will add an additional **DServer** device if it is not included in the *dev_info* parameter.

Example using *with_dserver=True*:

```
dev_info1 = DbDevInfo()
dev_info1.name = 'my/own/device'
dev_info1._class = 'MyDevice'
dev_info1.server = 'MyServer/test'
db.add_server(dev_info1.server, dev_info, with_dserver=True)
```

Same example using *with_dserver=False*:

```
        dev_info1 = DbDevInfo()
        dev_info1.name = 'my/own/device'
        dev_info1._class = 'MyDevice'
        dev_info1.server = 'MyServer/test'

        dev_info2 = DbDevInfo()
        dev_info1.name = 'dserver/' + dev_info1.server
        dev_info1._class = 'DServer
        dev_info1.server = dev_info1.server

        dev_info = dev_info1, dev_info2
        db.add_server(dev_info1.server, dev_info)
```

New in version 8.1.7: added *with_dserver* parameter

Parameters

servname (str) server name

dev_info (sequence<DbDevInfo> | DbDevInfos | DbDevInfo) containing the server device(s) information

with_dserver (bool) whether or not to auto create **DServer** device in server

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

delete_class_attribute_property(*self*, *class_name*, *value*) → None

Delete a list of attribute properties for the specified class.

Parameters

class_name (str) class name

propnames can be one of the following:

1. DbData [in] - several property data to be deleted
2. sequence<str> [in]- several property data to be deleted
3. sequence<DbDatum> [in] - several property data to be deleted
4. dict<str, seq<str>> keys are attribute names and value being a list of attribute property names

Return None

Throws *ConnectionFailed*, *CommunicationFailed DevFailed* from device (DB_SQLError)

delete_class_property(*self*, *class_name*, *value*) → None

Delete a the given of properties for the specified class.

Parameters

class_name (str) class name

value can be one of the following:

1. str [in] - single property data to be deleted
2. DbDatum [in] - single property data to be deleted
3. DbData [in] - several property data to be deleted

4. sequence<str> [in]- several property data to be deleted
5. sequence<DbDatum> [in] - several property data to be deleted
6. dict<str, obj> [in] - keys are property names to be deleted (values are ignored)
7. dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

delete_device_attribute_property(*self*, *dev_name*, *value*) → None

Delete a list of attribute properties for the specified device.

Parameters

devname (string) device name

propnames can be one of the following: 1. DbData [in] - several property data to be deleted 2. sequence<str> [in]- several property data to be deleted 3. sequence<DbDatum> [in] - several property data to be deleted 3. dict<str, seq<str>> keys are attribute names and value being a list of attribute property names

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

delete_device_property(*self*, *dev_name*, *value*) → None

Delete a the given of properties for the specified device.

Parameters

dev_name (str) object name

value can be one of the following: 1. str [in] - single property data to be deleted 2. DbDatum [in] - single property data to be deleted 3. DbData [in] - several property data to be deleted 4. sequence<str> [in]- several property data to be deleted 5. sequence<DbDatum> [in] - several property data to be deleted 6. dict<str, obj> [in] - keys are property names to be deleted (values are ignored) 7. dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

delete_property(*self*, *obj_name*, *value*) → None

Delete a the given of properties for the specified object.

Parameters

obj_name (str) object name

value can be one of the following:

1. str [in] - single property data to be deleted
2. DbDatum [in] - single property data to be deleted

3. DbData [in] - several property data to be deleted
4. sequence<string> [in]- several property data to be deleted
5. sequence<DbDatum> [in] - several property data to be deleted
6. dict<str, obj> [in] - keys are property names to be deleted (values are ignored)
7. dict<str, DbDatum> [in] - several DbDatum.name are property names to be deleted (keys are ignored)

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

export_server(*self*, *dev_info*) → None

Export a group of devices to the database.

Parameters

devinfo (sequence<DbDevExportInfo> | DbDevExportInfos | DbDevExportInfo) containing the device(s) to export information

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

get_class_attribute_property(*self*, *class_name*, *value*) → dict<str, dict<str, seq<str>>

Query the database for a list of class attribute properties for the specified class. The method returns all the properties for the specified attributes.

Parameters

class_name (str) class name

propnames can be one of the following:

1. str [in] - single attribute properties to be fetched
2. DbDatum [in] - single attribute properties to be fetched
3. DbData [in,out] - several attribute properties to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several attribute properties to be fetched
5. sequence<DbDatum> [in] - several attribute properties to be fetched
6. dict<str, obj> [in,out] - keys are attribute names In this case the given dict values will be changed to contain the several attribute property values

Return a dictionary which keys are the attribute names the value associated with each key being a another dictionary where keys are property names and value is a sequence of strings being the property value.

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

get_class_property(*self*, *class_name*, *value*) → dict<str, seq<str>>

Query the database for a list of class properties.

Parameters

class_name (str) class name

value can be one of the following:

1. str [in] - single property data to be fetched
2. tango.DbDatum [in] - single property data to be fetched
3. tango.DbData [in,out] - several property data to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several property data to be fetched
5. sequence<DbDatum> [in] - several property data to be fetched
6. dict<str, obj> [in,out] - keys are property names In this case the given dict values will be changed to contain the several property values

Return a dictionary which keys are the property names the value associated with each key being a a sequence of strings being the property value.

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

get_device_attribute_property(*self*, *dev_name*, *value*) → dict<str, dict<str, seq<str>>>

Query the database for a list of device attribute properties for the specified device. The method returns all the properties for the specified attributes.

Parameters

dev_name (string) device name

value can be one of the following:

1. str [in] - single attribute properties to be fetched
2. DbDatum [in] - single attribute properties to be fetched
3. DbData [in,out] - several attribute properties to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several attribute properties to be fetched
5. sequence<DbDatum> [in] - several attribute properties to be fetched
6. dict<str, obj> [in,out] - keys are attribute names In this case the given dict values will be changed to contain the several attribute property values

Return a dictionary which keys are the attribute names the value associated with each key being a another dictionary where keys are property names and value is a DbDatum containing the property value.

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

get_device_property(*self*, *dev_name*, *value*) → dict<str, seq<str>>

Query the database for a list of device properties.

Parameters

dev_name (str) object name

value can be one of the following:

1. str [in] - single property data to be fetched
2. DbDatum [in] - single property data to be fetched
3. DbData [in,out] - several property data to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several property data to be fetched
5. sequence<DbDatum> [in] - several property data to be fetched
6. dict<str, obj> [in,out] - keys are property names In this case the given dict values will be changed to contain the several property values

Return a dictionary which keys are the property names the value associated with each key being a a sequence of strings being the property value.

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

get_device_property_list(*self*, *dev_name*, *wildcard*, *array=None*) → DbData

Query the database for a list of properties defined for the specified device and which match the specified wildcard. If array parameter is given, it must be an object implementing de 'append' method. If given, it is filled with the matching property names. If not given the method returns a new DbDatum containing the matching property names.

New in PyTango 7.0.0

Parameters

dev_name (str) device name

wildcard (str) property name wildcard

array [out] (sequence) (optional) array that will contain the matching property names.

Return if container is None, return is a new DbDatum containing the matching property names. Otherwise returns the given array filled with the property names

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device

get_property(*self*, *obj_name*, *value*) → dict<str, seq<str>>

Query the database for a list of object (i.e non-device) properties.

Parameters

obj_name (str) object name

value can be one of the following:

1. str [in] - single property data to be fetched
2. DbDatum [in] - single property data to be fetched

3. DbData [in,out] - several property data to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several property data to be fetched
5. sequence<DbDatum> [in] - several property data to be fetched
6. dict<str, obj> [in,out] - keys are property names In this case the given dict values will be changed to contain the several property values

Return a dictionary which keys are the property names the value associated with each key being a a sequence of strings being the property value.

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

get_property_forced(*obj_name*, *value*)

get_property(self, obj_name, value) -> dict<str, seq<str>>

Query the database for a list of object (i.e non-device) properties.

Parameters

obj_name (str) object name

value can be one of the following:

1. str [in] - single property data to be fetched
2. DbDatum [in] - single property data to be fetched
3. DbData [in,out] - several property data to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several property data to be fetched
5. sequence<DbDatum> [in] - several property data to be fetched
6. dict<str, obj> [in,out] - keys are property names In this case the given dict values will be changed to contain the several property values

Return a dictionary which keys are the property names the value associated with each key being a a sequence of strings being the property value.

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

put_class_attribute_property(*self*, *class_name*, *value*) → None

Insert or update a list of properties for the specified class.

Parameters

class_name (str) class name

propdata can be one of the following:

1. tango.DbData - several property data to be inserted
2. sequence<DbDatum> - several property data to be inserted

3. dict<str, dict<str, obj>> keys are attribute names and value being another dictionary which keys are the attribute property names and the value associated with each key being:

   3.1 seq<str> 3.2 tango.DbDatum

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

put_class_property(*self*, *class_name*, *value*) → None

Insert or update a list of properties for the specified class.

Parameters

class_name (str) class name

value can be one of the following: 1. DbDatum - single property data to be inserted 2. DbData - several property data to be inserted 3. sequence<DbDatum> - several property data to be inserted 4. dict<str, DbDatum> - keys are property names and value has data to be inserted 5. dict<str, obj> - keys are property names and str(obj) is property value 6. dict<str, seq<str>> - keys are property names and value has data to be inserted

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

put_device_attribute_property(*self*, *dev_name*, *value*) → None

Insert or update a list of properties for the specified device.

Parameters

dev_name (str) device name

value can be one of the following:

1. DbData - several property data to be inserted
2. sequence<DbDatum> - several property data to be inserted
3. dict<str, dict<str, obj>> keys are attribute names and value being another dictionary which keys are the attribute property names and the value associated with each key being:

   3.1 seq<str> 3.2 tango.DbDatum

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

put_device_property(*self*, *dev_name*, *value*) → None

Insert or update a list of properties for the specified device.

Parameters

dev_name (str) object name

value can be one of the following:

1. DbDatum - single property data to be inserted
2. DbData - several property data to be inserted

3. sequence<DbDatum> - several property data to be inserted
4. dict<str, DbDatum> - keys are property names and value has data to be inserted
5. dict<str, obj> - keys are property names and str(obj) is property value
6. dict<str, seq<str>> - keys are property names and value has data to be inserted

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

put_property(*self*, *obj_name*, *value*) → None

Insert or update a list of properties for the specified object.

Parameters

obj_name (str) object name

value can be one of the following:

1. DbDatum - single property data to be inserted
2. DbData - several property data to be inserted
3. sequence<DbDatum> - several property data to be inserted
4. dict<str, DbDatum> - keys are property names and value has data to be inserted
5. dict<str, obj> - keys are property names and str(obj) is property value
6. dict<str, seq<str>> - keys are property names and value has data to be inserted

Return None

Throws *ConnectionFailed*, *CommunicationFailed*, *DevFailed* from device (DB_SQLError)

class tango.**DbDatum**

A single database value which has a name, type, address and value and methods for inserting and extracting C++ native types. This is the fundamental type for specifying database properties. Every property has a name and has one or more values associated with it. A status flag indicates if there is data in the DbDatum object or not. An additional flag allows the user to activate exceptions.

Note: DbDatum is extended to support the python sequence API. This way the DbDatum behaves like a sequence of strings. This allows the user to work with a DbDatum as if it was working with the old list of strings.

New in PyTango 7.0.0

class tango.**DbDevExportInfo**

import info for a device (should be retrived from the database) with the following members:

- name : (str) device name
- ior : (str) CORBA reference of the device
- host : name of the computer hosting the server
- version : (str) version
- pid : process identifier

class tango.**DbDevExportInfo**

import info for a device (should be retrived from the database) with the following members:

- name : (str) device name

- ior : (str) CORBA reference of the device
- host : name of the computer hosting the server
- version : (str) version
- pid : process identifier

class tango.**DbDevImportInfo**
import info for a device (should be retrived from the database) with the following members:
- name : (str) device name
- exported : 1 if device is running, 0 else
- ior : (str)CORBA reference of the device
- version : (str) version

class tango.**DbDevInfo**
A structure containing available information for a device with the following members:
- name : (str) name
- _class : (str) device class
- server : (str) server

class tango.**DbHistory**
A structure containing the modifications of a property. No public members.

class tango.**DbServerInfo**
A structure containing available information for a device server with the following members:
- name : (str) name
- host : (str) host
- mode : (str) mode
- level : (str) level

5.5 Encoded API

This feature is only possible since PyTango 7.1.4

class tango.**EncodedAttribute**

decode_gray16(*da*, *extract_as=<ExtensionMock name='_tango.ExtractAs.Numpy' id='140222368807344'>*)
Decode a 16 bits grayscale image (GRAY16) and returns a 16 bits gray scale image.

param da *DeviceAttribute* that contains the image

type da *DeviceAttribute*

param extract_as defaults to ExtractAs.Numpy

type extract_as ExtractAs

return the decoded data

- **In case String string is choosen as extract method, a tuple is returned:**
  width<int>, height<int>, buffer<str>
- In case Numpy is choosen as extract method, a numpy.ndarray is returned with ndim=2, shape=(height, width) and dtype=numpy.uint16.
- In case Tuple or List are choosen, a tuple<tuple<int>> or list<list<int>> is returned.

Warning: The PyTango calls that return a *DeviceAttribute* (like *DeviceProxy.read_attribute()* or `DeviceProxy.command_inout()`) automatically extract the contents by default. This method requires that the given *DeviceAttribute* is obtained from a call which **DOESN'T** extract the contents. Example:

```
dev = tango.DeviceProxy("a/b/c")
da = dev.read_attribute("my_attr", extract_as=tango.ExtractAs.Nothing)
enc = tango.EncodedAttribute()
data = enc.decode_gray16(da)
```

decode_gray8(*da*, *extract_as=<ExtensionMock name='_tango.ExtractAs.Numpy' id='140222368807344'>*)

Decode a 8 bits grayscale image (JPEG_GRAY8 or GRAY8) and returns a 8 bits gray scale image.

param da *DeviceAttribute* that contains the image

type da *DeviceAttribute*

param extract_as defaults to ExtractAs.Numpy

type extract_as ExtractAs

return the decoded data

- **In case String string is choosen as extract method, a tuple is returned:** width<int>, height<int>, buffer<str>
- In case Numpy is choosen as extract method, a `numpy.ndarray` is returned with ndim=2, shape=(height, width) and dtype=numpy.uint8.
- In case Tuple or List are choosen, a tuple<tuple<int>> or list<list<int>> is returned.

Warning: The PyTango calls that return a *DeviceAttribute* (like *DeviceProxy.read_attribute()* or `DeviceProxy.command_inout()`) automatically extract the contents by default. This method requires that the given *DeviceAttribute* is obtained from a call which **DOESN'T** extract the contents. Example:

```
dev = tango.DeviceProxy("a/b/c")
da = dev.read_attribute("my_attr", extract_as=tango.ExtractAs.Nothing)
enc = tango.EncodedAttribute()
data = enc.decode_gray8(da)
```

decode_rgb32(*da*, *extract_as=<ExtensionMock name='_tango.ExtractAs.Numpy' id='140222368807344'>*)

Decode a color image (JPEG_RGB or RGB24) and returns a 32 bits RGB image.

param da *DeviceAttribute* that contains the image

type da *DeviceAttribute*

param extract_as defaults to ExtractAs.Numpy

type extract_as ExtractAs

return the decoded data

- **In case String string is choosen as extract method, a tuple is returned:** width<int>, height<int>, buffer<str>

- In case Numpy is choosen as extract method, a numpy.ndarray is returned with ndim=2, shape=(height, width) and dtype=numpy.uint32.
- In case Tuple or List are choosen, a tuple<tuple<int>> or list<list<int>> is returned.

Warning: The PyTango calls that return a *DeviceAttribute* (like *DeviceProxy.read_attribute()* or DeviceProxy.command_inout()) automatically extract the contents by default. This method requires that the given *DeviceAttribute* is obtained from a call which **DOESN'T** extract the contents. Example:

```
dev = tango.DeviceProxy("a/b/c")
da = dev.read_attribute("my_attr", extract_as=tango.ExtractAs.Nothing)
enc = tango.EncodedAttribute()
data = enc.decode_rgb32(da)
```

encode_gray16 (*gray16*, *width=0*, *height=0*)
Encode a 16 bit grayscale image (no compression)

param gray16 an object containning image information

type gray16 str or buffer or numpy.ndarray or seq< seq<element> >

param width image width. **MUST** be given if gray16 is a string or if it is a numpy.ndarray with ndims != 2. Otherwise it is calculated internally.

type width int

param height image height. **MUST** be given if gray16 is a string or if it is a numpy.ndarray with ndims != 2. Otherwise it is calculated internally.

type height int

Note: When numpy.ndarray is given:

- gray16 **MUST** be CONTIGUOUS, ALIGNED
- if gray16.ndims != 2, width and height **MUST** be given and gray16.nbytes/2 **MUST** match width*height
- if gray16.ndims == 2, gray16.itemsize **MUST** be 2 (typically, gray16.dtype is one of *numpy.dtype.int16*, *numpy.dtype.uint16*, *numpy.dtype.short* or *numpy.dtype.ushort*)

Example :

```
def read_myattr(self, attr):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.int16)
    data = numpy.array((data,data,data))
    enc.encode_gray16(data)
    attr.set_value(enc)
```

encode_gray8 (*gray8*, *width=0*, *height=0*)
Encode a 8 bit grayscale image (no compression)

param gray8 an object containning image information

type gray8 str or numpy.ndarray or seq< seq<element> >

param width image width. **MUST** be given if gray8 is a string or if it is a `numpy.ndarray` with ndims != 2. Otherwise it is calculated internally.

type width `int`

param height image height. **MUST** be given if gray8 is a string or if it is a `numpy.ndarray` with ndims != 2. Otherwise it is calculated internally.

type height `int`

Note: When `numpy.ndarray` is given:

- gray8 **MUST** be CONTIGUOUS, ALIGNED
- if gray8.ndims != 2, width and height **MUST** be given and gray8.nbytes **MUST** match width*height
- if gray8.ndims == 2, gray8.itemsize **MUST** be 1 (typically, gray8.dtype is one of *numpy.dtype.byte*, *numpy.dtype.ubyte*, *numpy.dtype.int8* or *numpy.dtype.uint8*)

Example :

```
def read_myattr(self, attr):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.byte)
    data = numpy.array((data,data,data))
    enc.encode_gray8(data)
    attr.set_value(enc)
```

encode_jpeg_gray8 (*gray8*, *width=0*, *height=0*, *quality=100.0*)
Encode a 8 bit grayscale image as JPEG format

param gray8 an object containning image information

type gray8 `str` or `numpy.ndarray` or seq< seq<element> >

param width image width. **MUST** be given if gray8 is a string or if it is a `numpy.ndarray` with ndims != 2. Otherwise it is calculated internally.

type width `int`

param height image height. **MUST** be given if gray8 is a string or if it is a `numpy.ndarray` with ndims != 2. Otherwise it is calculated internally.

type height `int`

param quality Quality of JPEG (0=poor quality 100=max quality) (default is 100.0)

type quality `float`

Note: When `numpy.ndarray` is given:

- gray8 **MUST** be CONTIGUOUS, ALIGNED
- if gray8.ndims != 2, width and height **MUST** be given and gray8.nbytes **MUST** match width*height
- if gray8.ndims == 2, gray8.itemsize **MUST** be 1 (typically, gray8.dtype is one of *numpy.dtype.byte*, *numpy.dtype.ubyte*, *numpy.dtype.int8* or *numpy.dtype.uint8*)

Example :

```
def read_myattr(self, attr):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.byte)
    data = numpy.array((data,data,data))
    enc.encode_jpeg_gray8(data)
    attr.set_value(enc)
```

encode_jpeg_rgb24 (*rgb24*, *width=0*, *height=0*, *quality=100.0*)
Encode a 24 bit rgb color image as JPEG format.

param rgb24 an object containning image information

type rgb24 `str` or `numpy.ndarray` or seq< seq<element> >

param width image width. **MUST** be given if rgb24 is a string or if it is a `numpy.ndarray` with ndims != 3. Otherwise it is calculated internally.

type width `int`

param height image height. **MUST** be given if rgb24 is a string or if it is a `numpy.ndarray` with ndims != 3. Otherwise it is calculated internally.

type height `int`

param quality Quality of JPEG (0=poor quality 100=max quality) (default is 100.0)

type quality `float`

Note: When `numpy.ndarray` is given:

- rgb24 **MUST** be CONTIGUOUS, ALIGNED
- if rgb24.ndims != 3, width and height **MUST** be given and rgb24.nbytes/3 **MUST** match width*height
- if rgb24.ndims == 3, rgb24.itemsize **MUST** be 1 (typically, rgb24.dtype is one of *numpy.dtype.byte*, *numpy.dtype.ubyte*, *numpy.dtype.int8* or *numpy.dtype.uint8*) and shape **MUST** be (height, width, 3)

Example :

```
def read_myattr(self, attr):
    enc = tango.EncodedAttribute()
    # create an 'image' where each pixel is R=0x01, G=0x01, B=0x01
    arr = numpy.ones((10,10,3), dtype=numpy.uint8)
    enc.encode_jpeg_rgb24(data)
    attr.set_value(enc)
```

encode_jpeg_rgb32 (*rgb32*, *width=0*, *height=0*, *quality=100.0*)
Encode a 32 bit rgb color image as JPEG format.

param rgb32 an object containning image information

type rgb32 `str` or `numpy.ndarray` or seq< seq<element> >

param width image width. **MUST** be given if rgb32 is a string or if it is a `numpy.ndarray` with ndims != 2. Otherwise it is calculated internally.

type width `int`

param height image height. **MUST** be given if rgb32 is a string or if it is a `numpy.ndarray` with ndims != 2. Otherwise it is calculated internally.

type height `int`

Note: When `numpy.ndarray` is given:

- rgb32 **MUST** be CONTIGUOUS, ALIGNED
- if rgb32.ndims != 2, width and height **MUST** be given and rgb32.nbytes/4 **MUST** match width*height
- if rgb32.ndims == 2, rgb32.itemsize **MUST** be 4 (typically, rgb32.dtype is one of *numpy.dtype.int32*, *numpy.dtype.uint32*)

Example :

```
def read_myattr(self, attr):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.int32)
    data = numpy.array((data,data,data))
    enc.encode_jpeg_rgb32(data)
    attr.set_value(enc)
```

encode_rgb24 (*rgb24*, *width=0*, *height=0*)

Encode a 24 bit color image (no compression)

param rgb24 an object containning image information

type rgb24 `str` or `numpy.ndarray` or seq< seq<element> >

param width image width. **MUST** be given if rgb24 is a string or if it is a `numpy.ndarray` with ndims != 3. Otherwise it is calculated internally.

type width `int`

param height image height. **MUST** be given if rgb24 is a string or if it is a `numpy.ndarray` with ndims != 3. Otherwise it is calculated internally.

type height `int`

Note: When `numpy.ndarray` is given:

- rgb24 **MUST** be CONTIGUOUS, ALIGNED
- if rgb24.ndims != 3, width and height **MUST** be given and rgb24.nbytes/3 **MUST** match width*height
- if rgb24.ndims == 3, rgb24.itemsize **MUST** be 1 (typically, rgb24.dtype is one of *numpy.dtype.byte*, *numpy.dtype.ubyte*, *numpy.dtype.int8* or *numpy.dtype.uint8*) and shape **MUST** be (height, width, 3)

Example :

```
def read_myattr(self, attr):
    enc = tango.EncodedAttribute()
    # create an 'image' where each pixel is R=0x01, G=0x01, B=0x01
    arr = numpy.ones((10,10,3), dtype=numpy.uint8)
    enc.encode_rgb24(data)
    attr.set_value(enc)
```

5.6 The Utilities API

class tango.utils.**EventCallBack**(*format='{date} {dev_name} {name} {type} {value}', fd=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, max_buf=100*)

Useful event callback for test purposes

Usage:

```
>>> dev = tango.DeviceProxy(dev_name)
>>> cb = tango.utils.EventCallBack()
>>> id = dev.subscribe_event("state", tango.EventType.CHANGE_EVENT, cb, [])
2011-04-06 15:33:18.910474 sys/tg_test/1 STATE CHANGE [ATTR_VALID] ON
```

Allowed format keys are:
- date (event timestamp)
- reception_date (event reception timestamp)
- type (event type)
- dev_name (device name)
- name (attribute name)
- value (event value)

New in PyTango 7.1.4

get_events()

Returns the list of events received by this callback

Returns the list of events received by this callback

Return type sequence<obj>

push_event(*evt*)

Internal usage only

tango.utils.**is_pure_str**(*obj*)

Tells if the given object is a python string.

In python 2.x this means any subclass of basestring. In python 3.x this means any subclass of str.

Parameters **obj** (object) – the object to be inspected

Returns True is the given obj is a string or False otherwise

Return type bool

tango.utils.**is_seq**(*obj*)

Tells if the given object is a python sequence.

It will return True for any collections.Sequence (list, tuple, str, bytes, unicode), bytearray and (if numpy is enabled) numpy.ndarray

Parameters **obj** (object) – the object to be inspected

Returns True is the given obj is a sequence or False otherwise

Return type bool

tango.utils.**is_non_str_seq**(*obj*)

Tells if the given object is a python sequence (excluding string sequences).

It will return True for any collections.Sequence (list, tuple (and bytes in python3)), bytearray and (if numpy is enabled) numpy.ndarray

Parameters **obj** (object) – the object to be inspected

Returns True is the given obj is a sequence or False otherwise

Return type bool

tango.utils.**is_integer**(*obj*)
Tells if the given object is a python integer.

It will return True for any int, long (in python 2) and (if numpy is enabled) numpy.integer
Parameters **obj** (object) – the object to be inspected

Returns True is the given obj is a python integer or False otherwise

Return type bool

tango.utils.**is_number**(*obj*)
Tells if the given object is a python number.

It will return True for any numbers.Number and (if numpy is enabled) numpy.number
Parameters **obj** (object) – the object to be inspected

Returns True is the given obj is a python number or False otherwise

Return type bool

tango.utils.**is_bool**(*tg_type*, *inc_array=False*)
Tells if the given tango type is boolean
Parameters

- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is boolean or False otherwise

Return type bool

tango.utils.**is_scalar_type**(*tg_type*)
Tells if the given tango type is a scalar
Parameters **tg_type** (*tango.CmdArgType*) – tango type

Returns True if the given tango type is a scalar or False otherwise

Return type bool

tango.utils.**is_array_type**(*tg_type*)
Tells if the given tango type is an array type
Parameters **tg_type** (*tango.CmdArgType*) – tango type

Returns True if the given tango type is an array type or False otherwise

Return type bool

tango.utils.**is_numerical_type**(*tg_type*, *inc_array=False*)
Tells if the given tango type is numerical
Parameters

- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is a numerical or False otherwise

Return type bool

tango.utils.**is_int_type**(*tg_type*, *inc_array=False*)
Tells if the given tango type is integer
Parameters

- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is integer or False otherwise

Return type bool

tango.utils.**is_float_type**(*tg_type*, *inc_array=False*)
Tells if the given tango type is float
Parameters
- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is float or False otherwise

Return type bool

tango.utils.**is_bool_type**(*tg_type*, *inc_array=False*)
Tells if the given tango type is boolean
Parameters
- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is boolean or False otherwise

Return type bool

tango.utils.**is_bin_type**(*tg_type*, *inc_array=False*)
Tells if the given tango type is binary
Parameters
- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is binary or False otherwise

Return type bool

tango.utils.**is_str_type**(*tg_type*, *inc_array=False*)
Tells if the given tango type is string
Parameters
- **tg_type** (*tango.CmdArgType*) – tango type
- **inc_array** (bool) – (optional, default is False) determines if include array in the list of checked types

Returns True if the given tango type is string or False otherwise

Return type bool

tango.utils.**obj_2_str**(*obj*, *tg_type=None*)
Converts a python object into a string according to the given tango type
Parameters
- **obj** (object) – the object to be converted
- **tg_type** (*tango.CmdArgType*) – tango type

Returns a string representation of the given object

Return type str

tango.utils.**seqStr_2_obj**(*seq*, *tg_type*, *tg_format=None*)
Translates a sequence<str> to a sequence of objects of give type and format
Parameters
- **seq** (*sequence<str>*) – the sequence

- **tg_type** (*tango.CmdArgType*) – tango type
- **tg_format** (*tango.AttrDataFormat*) – (optional, default is None, meaning SCALAR) tango format

Returns a new sequence

tango.utils.**scalar_to_array_type**(*tg_type*)

Gives the array tango type corresponding to the given tango scalar type. Example: giving DevLong will return DevVarLongArray.

Parameters **tg_type** (*tango.CmdArgType*) – tango type

Returns the array tango type for the given scalar tango type

Return type *tango.CmdArgType*

Raises **ValueError** – in case the given dtype is not a tango scalar type

tango.utils.**get_home**()

Find user's home directory if possible. Otherwise raise error.

Returns user's home directory

Return type str

New in PyTango 7.1.4

tango.utils.**requires_pytango**(*min_version=None*, *conflicts=()*, *software_name='Software'*)

Determines if the required PyTango version for the running software is present. If not an exception is thrown. Example usage:

```
from tango import requires_pytango

requires_pytango('7.1', conflicts=['8.1.1'], software='MyDS')
```

Parameters

- **min_version** (None, str, LooseVersion) – minimum PyTango version [default: None, meaning no minimum required]. If a string is given, it must be in the valid version number format (see: LooseVersion)
- **conflicts** (*seq<str|LooseVersion>*) – a sequence of PyTango versions which conflict with the software using it
- **software_name** (*str*) – software name using tango. Used in the exception message

Raises **Exception** – if the required PyTango version is not met

New in PyTango 8.1.4

tango.utils.**requires_tango**(*min_version=None*, *conflicts=()*, *software_name='Software'*)

Determines if the required Tango version for the running software is present. If not an exception is thrown. Example usage:

```
from tango import requires_tango

requires_tango('7.1', conflicts=['8.1.1'], software='MyDS')
```

Parameters

- **min_version** (None, str, LooseVersion) – minimum Tango version [default: None, meaning no minimum required]. If a string is given, it must be in the valid version number format (see: LooseVersion)
- **conflicts** (*seq<str|LooseVersion>*) – a sequence of Tango versions which conflict with the software using it

- **software_name** (*str*) – software name using Tango. Used in the exception message

Raises **Exception** – if the required Tango version is not met

New in PyTango 8.1.4

5.7 Exception API

5.7.1 Exception definition

All the exceptions that can be thrown by the underlying Tango C++ API are available in the PyTango python module. Hence a user can catch one of the following exceptions:

- *DevFailed*
- *ConnectionFailed*
- *CommunicationFailed*
- *WrongNameSyntax*
- *NonDbDevice*
- *WrongData*
- *NonSupportedFeature*
- *AsynCall*
- *AsynReplyNotArrived*
- *EventSystemFailed*
- *NamedDevFailedList*
- *DeviceUnlocked*

When an exception is caught, the sys.exc_info() function returns a tuple of three values that give information about the exception that is currently being handled. The values returned are (type, value, traceback). Since most functions don't need access to the traceback, the best solution is to use something like exctype, value = sys.exc_info()[:2] to extract only the exception type and value. If one of the Tango exceptions is caught, the exctype will be class name of the exception (DevFailed, .. etc) and the value a tuple of dictionary objects all of which containing the following kind of key-value pairs:

- **reason**: a string describing the error type (more readable than the associated error code)
- **desc**: a string describing in plain text the reason of the error.
- **origin**: a string giving the name of the (C++ API) method which thrown the exception
- **severity**: one of the strings WARN, ERR, PANIC giving severity level of the error.

```
import tango

#  How to protect the script from exceptions raised by the Tango
try:
    # Get proxy on a non existing device should throw an exception
    device = tango.DeviceProxy("non/existing/device")
except DevFailed as df:
    print("Failed to create proxy to non/existing/device:\n%s" % df)
```

5.7.2 Throwing exception in a device server

The C++ `tango::Except` class with its most important methods have been wrapped to Python. Therefore, in a Python device server, you have the following methods to throw, re-throw or print a Tango::DevFailed exception :

- `throw_exception()` which is a static method
- `re_throw_exception()` which is also a static method
- `print_exception()` which is also a static method

The following code is an example of a command method requesting a command on a sub-device and re-throwing the exception in case of:

```
1 try:
2     dev.command_inout("SubDevCommand")
3 except tango.DevFailed as df:
4     tango.Except.re_throw_exception(df,
5         "MyClass_CommandFailed",
6         "Sub device command SubdevCommand failed",
7         "Command()")
```

line 2 Send the command to the sub device in a try/catch block

line 4-6 Re-throw the exception and add a new level of information in the exception stack

5.7.3 Exception API

class `tango.`**`Except`**
A containner for the static methods:
- throw_exception
- re_throw_exception
- print_exception
- compare_exception

class `tango.`**`DevError`**
Structure describing any error resulting from a command execution, or an attribute query, with following members:
- reason : (`str`) reason
- severity : (*`ErrSeverity`*) error severty (WARN, ERR, PANIC)
- desc : (`str`) error description
- origin : (`str`) Tango server method in which the error happened

exception `tango.`**`DevFailed`**

exception `tango.`**`ConnectionFailed`**
This exception is thrown when a problem occurs during the connection establishment between the application and the device. The API is stateless. This means that DeviceProxy constructors filter most of the exception except for cases described in the following table.

The desc DevError structure field allows a user to get more precise information. These informations are :

DB_DeviceNotDefined The name of the device not defined in the database

API_CommandFailed The device and command name

API_CantConnectToDevice The device name

API_CorbaException The name of the CORBA exception, its reason, its locality, its completed flag and its minor code

API_CantConnectToDatabase The database server host and its port number

API_DeviceNotExported The device name

exception tango.**CommunicationFailed**

This exception is thrown when a communication problem is detected during the communication between the client application and the device server. It is a two levels Tango::DevError structure. In case of time-out, the DevError structures fields are:

Level	Reason	Desc	Severity
0	API_CorbaException	CORBA exception fields translated into a string	ERR
1	API_DeviceTimedOut	String with time-out value and device name	ERR

For all other communication errors, the DevError structures fields are:

Level	Reason	Desc	Severity
0	API_CorbaException	CORBA exception fields translated into a string	ERR
1	API_CommunicationFailed	String with device, method, command/attribute name	ERR

exception tango.**WrongNameSyntax**

This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_UnsupportedProtocol This error occurs when trying to build a DeviceProxy or an AttributeProxy instance for a device with an unsupported protocol. Refer to the appendix on device naming syntax to get the list of supported database modifier

API_UnsupportedDBaseModifier This error occurs when trying to build a DeviceProxy or an AttributeProxy instance for a device/attribute with a database modifier unsupported. Refer to the appendix on device naming syntax to get the list of supported database modifier

API_WrongDeviceNameSyntax This error occurs for all the other error in device name syntax. It is thrown by the DeviceProxy class constructor.

API_WrongAttributeNameSyntax This error occurs for all the other error in attribute name syntax. It is thrown by the AttributeProxy class constructor.

API_WrongWildcardUsage This error occurs if there is a bad usage of the wildcard character

exception tango.**NonDbDevice**

This exception has only one level of Tango::DevError structure. The reason field is set to API_NonDatabaseDevice. This exception is thrown by the API when using the DeviceProxy or AttributeProxy class database access for non-database device.

exception tango.**WrongData**

This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_EmptyDbDatum This error occurs when trying to extract data from an empty DbDatum object

API_IncompatibleArgumentType This error occurs when trying to extract data with a type different than the type used to send the data

API_EmptyDeviceAttribute This error occurs when trying to extract data from an empty DeviceAttribute object

API_IncompatibleAttrArgumentType This error occurs when trying to extract attribute data with a type different than the type used to send the data

API_EmptyDeviceData This error occurs when trying to extract data from an empty DeviceData object

API_IncompatibleCmdArgumentType This error occurs when trying to extract command data with a type different than the type used to send the data

exception tango.**NonSupportedFeature**

This exception is thrown by the API layer when a request to a feature implemented in Tango device interface release n is requested for a device implementing Tango device interface n-x. There is one possible value for the reason field which is API_UnsupportedFeature.

exception tango.**AsynCall**

This exception is thrown by the API layer when a the asynchronous model id badly used. This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_BadAsynPollId This error occurs when using an asynchronous request identifier which is not valid any more.

API_BadAsyn This error occurs when trying to fire callback when no callback has been previously registered

API_BadAsynReqType This error occurs when trying to get result of an asynchronous request with an asynchronous request identifier returned by a non-coherent asynchronous request (For instance, using the asynchronous request identifier returned by a command_inout_asynch() method with a read_attribute_reply() attribute).

exception tango.**AsynReplyNotArrived**

This exception is thrown by the API layer when:

- a request to get asynchronous reply is made and the reply is not yet arrived
- a blocking wait with timeout for asynchronous reply is made and the timeout expired.

There is one possible value for the reason field which is API_AsynReplyNotArrived.

exception tango.**EventSystemFailed**

This exception is thrown by the API layer when subscribing or unsubscribing from an event failed. This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_NotificationServiceFailed This error occurs when the subscribe_event() method failed trying to access the CORBA notification service

API_EventNotFound This error occurs when you are using an incorrect event_id in the unsubscribe_event() method

API_InvalidArgs This error occurs when NULL pointers are passed to the subscribe or unsubscribe event methods

API_MethodArgument This error occurs when trying to subscribe to an event which has already been subsribed to

API_DSFailedRegisteringEvent This error means that the device server to which the device belongs to failed when it tries to register the event. Most likely, it means that there is no event property defined

API_EventNotFound Occurs when using a wrong event identifier in the unsubscribe_event method

exception tango.**DeviceUnlocked**

This exception is thrown by the API layer when a device locked by the process has been unlocked by an admin client. This exception has two levels of Tango::DevError structure. There is only possible value for the reason field which is

API_DeviceUnlocked The device has been unlocked by another client (administration client)

The first level is the message reported by the Tango kernel from the server side. The second layer is added by the client API layer with informations on which API call generates the exception and device name.

exception tango.**NotAllowed**

exception tango.**NamedDevFailedList**

This exception is only thrown by the DeviceProxy::write_attributes() method. In this case, it is necessary to have a new class of exception to transfer the error stack for several attribute(s) which failed during the writing. Therefore, this exception class contains for each attributes which failed :

- The name of the attribute
- Its index in the vector passed as argumen tof the write_attributes() method
- The error stack

CHAPTER 6

How to

This is a small list of how-tos specific to PyTango. A more general Tango how-to list can be found here.

6.1 Check the default TANGO host

The default TANGO host can be defined using the environment variable `TANGO_HOST` or in a *tangorc* file (see Tango environment variables for complete information)

To check what is the current value that TANGO uses for the default configuration simple do:

```
>>> import tango
>>> tango.ApiUtil.get_env_var("TANGO_HOST")
'homer.simpson.com:10000'
```

6.2 Check TANGO version

There are two library versions you might be interested in checking: The PyTango version:

```
>>> import tango
>>> tango.__version__
'9.2.0'

>>> tango.__version_info__
(9, 2, 0, 'b', 1)
```

... and the Tango C++ library version that PyTango was compiled with:

```
>>> import tango
>>> tango.constants.TgLibVers
'9.2.0'
```

6.3 Report a bug

Bugs can be reported as tickets in TANGO Source forge.

When making a bug report don't forget to select *PyTango* in **Category**.

It is also helpfull if you can put in the ticket description the PyTango information. It can be a dump of:

```
$ python -c "from tango.utils import info; print(info())"
```

6.4 Test the connection to the Device and get it's current state

One of the most basic examples is to get a reference to a device and determine if it is running or not:

```
from tango import DeviceProxy

# Get proxy on the tango_test1 device
print("Creating proxy to TangoTest device...")
tango_test = DeviceProxy("sys/tg_test/1")

# ping it
print(tango_test.ping())

# get the state
print(tango_test.state())
```

6.5 Read and write attributes

Basic read/write attribute operations:

```
from tango import DeviceProxy

# Get proxy on the tango_test1 device
print("Creating proxy to TangoTest device...")
tango_test = DeviceProxy("sys/tg_test/1")

# Read a scalar attribute. This will return a tango.DeviceAttribute
# Member 'value' contains the attribute value
scalar = tango_test.read_attribute("long_scalar")
print("Long_scalar value = {0}".format(scalar.value))

# PyTango provides a shorter way:
scalar = tango_test.long_scalar.value
print("Long_scalar value = {0}".format(scalar))

# Read a spectrum attribute
spectrum = tango_test.read_attribute("double_spectrum")
# ... or, the shorter version:
spectrum = tango_test.double_spectrum

# Write a scalar attribute
scalar_value = 18
tango_test.write_attribute("long_scalar", scalar_value)

#  PyTango provides a shorter way:
tango_test.long_scalar = scalar_value

# Write a spectrum attribute
spectrum_value = [1.2, 3.2, 12.3]
tango_test.write_attribute("double_spectrum", spectrum_value)
# ... or, the shorter version:
tango_test.double_spectrum = spectrum_value

# Write an image attribute
image_value = [ [1, 2], [3, 4] ]
tango_test.write_attribute("long_image", image_value)
# ... or, the shorter version:
tango_test.long_image = image_value
```

Note that if PyTango is compiled with numpy support the values got when reading a spectrum or an image will be numpy arrays. This results in a faster and more memory efficient PyTango. You can also use numpy to specify the values when writing attributes, especially if you know the exact attribute type:

```
import numpy
from tango import DeviceProxy

# Get proxy on the tango_test1 device
print("Creating proxy to TangoTest device...")
tango_test = DeviceProxy("sys/tg_test/1")

data_1d_long = numpy.arange(0, 100, dtype=numpy.int32)

tango_test.long_spectrum = data_1d_long

data_2d_float = numpy.zeros((10,20), dtype=numpy.float64)

tango_test.double_image = data_2d_float
```

6.6 Execute commands

As you can see in the following example, when scalar types are used, the Tango binding automagically manages the data types, and writing scripts is quite easy:

```
from tango import DeviceProxy

# Get proxy on the tango_test1 device
print("Creating proxy to TangoTest device...")
tango_test = DeviceProxy("sys/tg_test/1")

# First use the classical command_inout way to execute the DevString command
# (DevString in this case is a command of the Tango_Test device)

result = tango_test.command_inout("DevString", "First hello to device")
print("Result of execution of DevString command = {0}".format(result))

# the same can be achieved with a helper method
result = tango_test.DevString("Second Hello to device")
print("Result of execution of DevString command = {0}".format(result))

# Please note that argin argument type is automatically managed by python
result = tango_test.DevULong(12456)
print("Result of execution of DevULong command = {0}".format(result))
```

6.7 Execute commands with more complex types

In this case you have to use put your arguments data in the correct python structures:

```
from tango import DeviceProxy

# Get proxy on the tango_test1 device
print("Creating proxy to TangoTest device...")
tango_test = DeviceProxy("sys/tg_test/1")

# The input argument is a DevVarLongStringArray so create the argin
# variable containing an array of longs and an array of strings
```

```
argin = ([1,2,3], ["Hello", "TangoTest device"])

result = tango_test.DevVarLongArray(argin)
print("Result of execution of DevVarLongArray command = {0}".format(result))
```

6.8 Work with Groups

Todo

write this how to

6.9 Handle errors

Todo

write this how to

For now check *Exception API*.

6.10 Registering devices

Here is how to define devices in the Tango DataBase:

```
from tango import Database, DbDevInfo

#  A reference on the DataBase
db = Database()

# The 3 devices name we want to create
# Note: these 3 devices will be served by the same DServer
new_device_name1 = "px1/tdl/mouse1"
new_device_name2 = "px1/tdl/mouse2"
new_device_name3 = "px1/tdl/mouse3"

# Define the Tango Class served by this  DServer
new_device_info_mouse = DbDevInfo()
new_device_info_mouse._class = "Mouse"
new_device_info_mouse.server = "ds_Mouse/server_mouse"

# add the first device
print("Creating device: %s" % new_device_name1)
new_device_info_mouse.name = new_device_name1
db.add_device(new_device_info_mouse)

# add the next device
print("Creating device: %s" % new_device_name2)
new_device_info_mouse.name = new_device_name2
db.add_device(new_device_info_mouse)

# add the third device
print("Creating device: %s" % new_device_name3)
new_device_info_mouse.name = new_device_name3
db.add_device(new_device_info_mouse)
```

6.10.1 Setting up device properties

A more complex example using python subtilities. The following python script example (containing some functions and instructions manipulating a Galil motor axis device server) gives an idea of how the Tango API should be accessed from Python:

```
from tango import DeviceProxy

# connecting to the motor axis device
axis1 = DeviceProxy("microxas/motorisation/galilbox")

# Getting Device Properties
property_names = ["AxisBoxAttachement",
                  "AxisEncoderType",
                  "AxisNumber",
                  "CurrentAcceleration",
                  "CurrentAccuracy",
                  "CurrentBacklash",
                  "CurrentDeceleration",
                  "CurrentDirection",
                  "CurrentMotionAccuracy",
                  "CurrentOvershoot",
                  "CurrentRetry",
                  "CurrentScale",
                  "CurrentSpeed",
                  "CurrentVelocity",
                  "EncoderMotorRatio",
                  "logging_level",
                  "logging_target",
                  "UserEncoderRatio",
                  "UserOffset"]

axis_properties = axis1.get_property(property_names)
for prop in axis_properties.keys():
    print("%s: %s" % (prop, axis_properties[prop][0]))

# Changing Properties
axis_properties["AxisBoxAttachement"] = ["microxas/motorisation/galilbox"]
axis_properties["AxisEncoderType"] = ["1"]
axis_properties["AxisNumber"] = ["6"]
axis1.put_property(axis_properties)
```

6.11 Write a server

Before reading this chapter you should be aware of the TANGO basic concepts. This chapter does not explain what a Tango device or a device server is. This is explained in details in the Tango control system manual

Since version 8.1, PyTango provides a helper module which simplifies the development of a Tango device server. This helper is provided through the *tango.server* module.

Here is a simple example on how to write a *Clock* device server using the high level API

```
import time
from tango.server import run
from tango.server import Device, DeviceMeta
from tango.server import attribute, command, pipe
```

```
5
6
7  class Clock(Device):
8      __metaclass__ = DeviceMeta
9
10     @attribute
11     def time(self):
12         return time.time()
13
14     @command(dtype_in=str, dtype_out=str)
15     def strftime(self, format):
16         return time.strftime(format)
17
18     @pipe
19     def info(self):
20         return ('Information',
21                 dict(manufacturer='Tango',
22                      model='PS2000',
23                      version_number=123))
24
25
26 if __name__ == "__main__":
27     run([Clock])
```

line 2-4 import the necessary symbols

line 7 tango device class definition. A Tango device must inherit from *tango.server.Device*

line 8 mandatory *magic* line. A Tango device must define the metaclass as `tango.server.DeviceClass`. This has to be done due to a limitation on boost-python

line 10-12 definition of the *time* attribute. By default, attributes are double, scalar, read-only. Check the *attribute* for the complete list of attribute options.

line 14-16 the method *strftime* is exported as a Tango command. In receives a string as argument and it returns a string. If a method is to be exported as a Tango command, it must be decorated as such with the *command()* decorator

line 18-23 definition of the *info* pipe. Check the *pipe* for the complete list of pipe options.

line 28 start the Tango run loop. The mandatory argument is a list of python classes that are to be exported as Tango classes. Check *run()* for the complete list of options

Here is a more complete example on how to write a *PowerSupply* device server using the high level API. The example contains:

1. a read-only double scalar attribute called *voltage*
2. a read/write double scalar expert attribute *current*
3. a read-only double image attribute called *noise*
4. a *ramp* command
5. a *host* device property
6. a *port* class property

```
1 from time import time
2 from numpy.random import random_sample
3
4 from tango import AttrQuality, AttrWriteType, DispLevel, run
5 from tango.server import Device, DeviceMeta, attribute, command
6 from tango.server import class_property, device_property
7
8
9 class PowerSupply(Device):
```

```
    __metaclass__ = DeviceMeta

    current = attribute(label="Current", dtype=float,
                        display_level=DispLevel.EXPERT,
                        access=AttrWriteType.READ_WRITE,
                        unit="A", format="8.4f",
                        min_value=0.0, max_value=8.5,
                        min_alarm=0.1, max_alarm=8.4,
                        min_warning=0.5, max_warning=8.0,
                        fget="get_current", fset="set_current",
                        doc="the power supply current")

    noise = attribute(label="Noise", dtype=((float,),),
                      max_dim_x=1024, max_dim_y=1024,
                      fget="get_noise")

    host = device_property(dtype=str)
    port = class_property(dtype=int, default_value=9788)

    @attribute
    def voltage(self):
        self.info_stream("get voltage(%s, %d)" % (self.host, self.port))
        return 10.0

    def get_current(self):
        return 2.3456, time(), AttrQuality.ATTR_WARNING

    def set_current(self, current):
        print("Current set to %f" % current)

    def get_noise(self):
        return random_sample((1024, 1024))

    @command(dtype_in=float)
    def ramp(self, value):
        print("Ramping up...")


if __name__ == "__main__":
    run([PowerSupply])
```

Note: the `__metaclass__` statement is mandatory due to a limitation in the *boost-python* library used by PyTango.

If you are using python 3 you can write instead:

```
class PowerSupply(Device, metaclass=DeviceMeta)
    pass
```

6.12 Server logging

This chapter instructs you on how to use the tango logging API (log4tango) to create tango log messages on your device server.

The logging system explained here is the Tango Logging Service (TLS). For detailed information on how this logging system works please check:

- 3.5 The tango logging service

- 9.3 The tango logging service

The easiest way to start seeing log messages on your device server console is by starting it with the verbose option. Example:

```
python PyDsExp.py PyDs1 -v4
```

This activates the console tango logging target and filters messages with importance level DEBUG or more. The links above provided detailed information on how to configure log levels and log targets. In this document we will focus on how to write log messages on your device server.

6.12.1 Basic logging

The most basic way to write a log message on your device is to use the *Device* logging related methods:

- `debug_stream()`
- `info_stream()`
- `warn_stream()`
- `error_stream()`
- `fatal_stream()`

Example:

```
def read_voltage(self):
    self.info_stream("read voltage attribute")
    # ...
    return voltage_value
```

This will print a message like:

```
1282206864 [-1215867200] INFO test/power_supply/1 read voltage attribute
```

every time a client asks to read the *voltage* attribute value.

The logging methods support argument list feature (since PyTango 8.1). Example:

```
def read_voltage(self):
    self.info_stream("read_voltage(%s, %d)", self.host, self.port)
    # ...
    return voltage_value
```

6.12.2 Logging with print statement

This feature is only possible since PyTango 7.1.3

It is possible to use the print statement to log messages into the tango logging system. This is achieved by using the python's print extend form sometimes refered to as *print chevron*.

Same example as above, but now using *print chevron*:

```
def read_voltage(self, the_att):
    print >>self.log_info, "read voltage attribute"
    # ...
    return voltage_value
```

Or using the python 3k print function:

```
def read_Long_attr(self, the_att):
    print("read voltage attribute", file=self.log_info)
    # ...
    return voltage_value
```

6.12.3 Logging with decorators

This feature is only possible since PyTango 7.1.3

PyTango provides a set of decorators that place automatic log messages when you enter and when you leave a python method. For example:

```
@tango.DebugIt()
def read_Long_attr(self, the_att):
    the_att.set_value(self.attr_long)
```

will generate a pair of log messages each time a client asks for the 'Long_attr' value. Your output would look something like:

```
1282208997 [-1215965504] DEBUG test/pydsexp/1 -> read_Long_attr()
1282208997 [-1215965504] DEBUG test/pydsexp/1 <- read_Long_attr()
```

Decorators exist for all tango log levels:

- *tango.DebugIt*
- *tango.InfoIt*
- *tango.WarnIt*
- *tango.ErrorIt*
- *tango.FatalIt*

The decorators receive three optional arguments:

- show_args - shows method arguments in log message (defaults to False)
- show_kwargs shows keyword method arguments in log message (defaults to False)
- show_ret - shows return value in log message (defaults to False)

Example:

```
@tango.DebugIt(show_args=True, show_ret=True)
def IOLong(self, in_data):
    return in_data * 2
```

will output something like:

```
1282221947 [-1261438096] DEBUG test/pydsexp/1 -> IOLong(23)
1282221947 [-1261438096] DEBUG test/pydsexp/1 46 <- IOLong()
```

6.13 Multiple device classes (Python and C++) in a server

Within the same python interpreter, it is possible to mix several Tango classes. Let's say two of your colleagues programmed two separate Tango classes in two separated python files: A `PLC` class in a `PLC.py`:

```
# PLC.py

from tango.server import Device, DeviceMeta, run

class PLC(Device):
    __metaclass__ = DeviceMeta

    # bla, bla my PLC code

if __name__ == "__main__":
    run([PLC])
```

... and a `IRMirror` in a `IRMirror.py`:

```
# IRMirror.py

from tango.server import Device, DeviceMeta, run

class IRMirror(Device):
    __metaclass__ = DeviceMeta

    # bla, bla my IRMirror code

if __name__ == "__main__":
    run([IRMirror])
```

You want to create a Tango server called *PLCMirror* that is able to contain devices from both PLC and IRMirror classes. All you have to do is write a `PLCMirror.py` containing the code:

```
# PLCMirror.py

from tango.server import run
from PLC import PLC
from IRMirror import IRMirror

run([PLC, IRMirror])
```

It is also possible to add C++ Tango class in a Python device server as soon as:

1. The Tango class is in a shared library
2. It exist a C function to create the Tango class

For a Tango class called MyTgClass, the shared library has to be called MyTgClass.so and has to be in a directory listed in the LD_LIBRARY_PATH environment variable. The C function creating the Tango class has to be called _create_MyTgClass_class() and has to take one parameter of type "char *" which is the Tango class name. Here is an example of the main function of the same device server than before but with one C++ Tango class called SerialLine:

```
import tango
import sys

if __name__ == '__main__':
    py = tango.Util(sys.argv)
    util.add_class('SerialLine', 'SerialLine', language="c++")
    util.add_class(PLCClass, PLC, 'PLC')
    util.add_class(IRMirrorClass, IRMirror, 'IRMirror')

    U = tango.Util.instance()
    U.server_init()
    U.server_run()
```

Line 6 The C++ class is registered in the device server

Line 7 and 8 The two Python classes are registered in the device server

6.14 Create attributes dynamically

It is also possible to create dynamic attributes within a Python device server. There are several ways to create dynamic attributes. One of the way, is to create all the devices within a loop, then to create the dynamic attributes and finally to make all the devices available for the external world. In C++ device server, this is typically done within the <Device>Class::device_factory() method. In Python device server, this method is generic and the user does not have one. Nevertheless, this generic device_factory method calls a method named dyn_attr() allowing the user to create his dynamic attributes. It is simply necessary to re-define this method within your <Device>Class and to create the dynamic attribute within this method:

```
dyn_attr(self, dev_list)
```

where dev_list is a list containing all the devices created by the generic device_factory() method.

There is another point to be noted regarding dynamic attribute within Python device server. The Tango Python device server core checks that for each attribute it exists methods named <attribute_name>_read and/or <attribute_name>_write and/or is_<attribute_name>_allowed. Using dynamic attribute, it is not possible to define these methods because attributes name and number are known only at run-time. To address this issue, the Device_3Impl::add_attribute() method has a diferent signature for Python device server which is:

```
add_attribute(self, attr, r_meth = None, w_meth = None,
is_allo_meth = None)
```

attr is an instance of the Attr class, r_meth is the method which has to be executed with the attribute is read, w_meth is the method to be executed when the attribute is written and is_allo_meth is the method to be executed to implement the attribute state machine. The method passed here as argument as to be class method and not object method. Which argument you have to use depends on the type of the attribute (A WRITE attribute does not need a read method). Note, that depending on the number of argument you pass to this method, you may have to use Python keyword argument. The necessary methods required by the Tango Python device server core will be created automatically as a forward to the methods given as arguments.

Here is an example of a device which has a TANGO command called *createFloatAttribute*. When called, this command creates a new scalar floating point attribute with the specified name:

```
1  from tango import Util, Attr
2  from tango.server import DeviceMeta, Device, command
3
4  class MyDevice(Device):
5      __metaclass__ = DeviceMeta
6
7      @command(dtype_in=str)
8      def CreateFloatAttribute(self, attr_name):
9          attr = Attr(attr_name, tango.DevDouble)
10         self.add_attribute(attr, self.read_General, self.write_General)
11
12     def read_General(self, attr):
13         self.info_stream("Reading attribute %s", attr.get_name())
14         attr.set_value(99.99)
15
16     def write_General(self, attr):
17         self.info_stream("Writting attribute %s", attr.get_name())
```

6.15 Create/Delete devices dynamically

This feature is only possible since PyTango 7.1.2

Starting from PyTango 7.1.2 it is possible to create devices in a device server "en caliente". This means that you can create a command in your "management device" of a device server that creates devices of (possibly) several other tango classes. There are two ways to create a new device which are described below.

Tango imposes a limitation: the tango class(es) of the device(s) that is(are) to be created must have been registered before the server starts. If you use the high level API, the tango class(es) must be listed in the call to *run()*. If you use the lower level server API, it must be done using individual calls to *add_class()*.

6.15.1 Dynamic device from a known tango class name

If you know the tango class name but you don't have access to the *tango.DeviceClass* (or you are too lazy to search how to get it ;-) the way to do it is call *create_device()* / *delete_device()*. Here is an example of implementing a tango command on one of your devices that creates a device of some arbitrary class (the example assumes the tango commands 'CreateDevice' and 'DeleteDevice' receive a parameter of type DevVarStringArray with two strings. No error processing was done on the code for simplicity sake):

```
from tango import Util
from tango.server import DeviceMeta, Device, command

class MyDevice(Device):
    __metaclass__ = DeviceMeta

    @command(dtype_in=[str])
    def CreateDevice(self, pars):
        klass_name, dev_name = pars
        util = Util.instance()
        util.create_device(klass_name, dev_name, alias=None, cb=None)

    @command(dtype_in=[str])
    def DeleteDevice(self, pars):
        klass_name, dev_name = pars
        util = Util.instance()
        util.delete_device(klass_name, dev_name)
```

An optional callback can be registered that will be executed after the device is registed in the tango database but before the actual device object is created and its init_device method is called. It can be used, for example, to initialize some device properties.

6.15.2 Dynamic device from a known tango class

If you already have access to the *DeviceClass* object that corresponds to the tango class of the device to be created you can call directly the *create_device()* / *delete_device()*. For example, if you wish to create a clone of your device, you can create a tango command called *Clone*:

```
class MyDevice(tango.Device_4Impl):

    def fill_new_device_properties(self, dev_name):
        prop_names = db.get_device_property_list(self.get_name(), "*")
        prop_values = db.get_device_property(self.get_name(), prop_names.value_string)
        db.put_device_property(dev_name, prop_values)

```

```
8           # do the same for attributes...
9           ...
10
11      def Clone(self, dev_name):
12          klass = self.get_device_class()
13          klass.create_device(dev_name, alias=None, cb=self.fill_new_device_properties)
14
15      def DeleteSibling(self, dev_name):
16          klass = self.get_device_class()
17          klass.delete_device(dev_name)
```

Note that the cb parameter is optional. In the example it is given for demonstration purposes only.

6.16 Write a server (original API)

This chapter describes how to develop a PyTango device server using the original PyTango server API. This API mimics the C++ API and is considered low level. You should write a server using this API if you are using code generated by Pogo tool or if for some reason the high level API helper doesn't provide a feature you need (in that case think of writing a mail to tango mailing list explaining what you cannot do).

6.16.1 The main part of a Python device server

The rule of this part of a Tango device server is to:

- Create the *Util* object passing it the Python interpreter command line arguments
- Add to this object the list of Tango class(es) which have to be hosted by this interpreter
- Initialize the device server
- Run the device server loop

The following is a typical code for this main function:

```
1 if __name__ == '__main__':
2     util = tango.Util(sys.argv)
3     util.add_class(PyDsExpClass, PyDsExp)
4
5     U = tango.Util.instance()
6     U.server_init()
7     U.server_run()
```

Line 2 Create the Util object passing it the interpreter command line arguments

Line 3 Add the Tango class *PyDsExp* to the device server. The *Util.add_class()* method of the Util class has two arguments which are the Tango class PyDsExpClass instance and the Tango PyDsExp instance. This *Util.add_class()* method is only available since version 7.1.2. If you are using an older version please use *Util.add_TgClass()* instead.

Line 7 Initialize the Tango device server

Line 8 Run the device server loop

6.16.2 The PyDsExpClass class in Python

The rule of this class is to :

- Host and manage data you have only once for the Tango class whatever devices of this class will be created

- Define Tango class command(s)
- Define Tango class attribute(s)

In our example, the code of this Python class looks like:

```
1  class PyDsExpClass(tango.DeviceClass):
2
3      cmd_list = { 'IOLong' : [ [ tango.ArgType.DevLong, "Number" ],
4                                [ tango.ArgType.DevLong, "Number * 2" ] ],
5                   'IOStringArray' : [ [ tango.ArgType.DevVarStringArray, "Array of string" ],
6                                       [ tango.ArgType.DevVarStringArray, "This reversed array"] ],
7      }
8
9      attr_list = { 'Long_attr' : [ [ tango.ArgType.DevLong ,
10                                     tango.AttrDataFormat.SCALAR ,
11                                     tango.AttrWriteType.READ],
12                                   { 'min alarm' : 1000, 'max alarm' : 1500 } ],
13
14                   'Short_attr_rw' : [ [ tango.ArgType.DevShort,
15                                         tango.AttrDataFormat.SCALAR,
16                                         tango.AttrWriteType.READ_WRITE ] ]
17     }
```

Line 1 The PyDsExpClass class has to inherit from the *DeviceClass* class

Line 3 to 7 Definition of the cmd_list `dict` defining commands. The *IOLong* command is defined at lines 3 and 4. The *IOStringArray* command is defined in lines 5 and 6

Line 9 to 17 Definition of the attr_list `dict` defining attributes. The *Long_attr* attribute is defined at lines 9 to 12 and the *Short_attr_rw* attribute is defined at lines 14 to 16

If you have something specific to do in the class constructor like initializing some specific data member, you will have to code a class constructor. An example of such a contructor is

```
1  def __init__(self, name):
2      tango.DeviceClass.__init__(self, name)
3      self.set_type("TestDevice")
```

The device type is set at line 3.

6.16.3 Defining commands

As shown in the previous example, commands have to be defined in a `dict` called *cmd_list* as a data member of the xxxClass class of the Tango class. This `dict` has one element per command. The element key is the command name. The element value is a python list which defines the command. The generic form of a command definition is:

```
'cmd_name' :  [ [in_type, <"In desc">], [out_type, <"Out desc">],
<{opt parameters}>]
```

The first element of the value list is itself a list with the command input data type (one of the `tango.ArgType` pseudo enumeration value) and optionally a string describing this input argument. The second element of the value list is also a list with the command output data type (one of the `tango.ArgType` pseudo enumeration value) and optionaly a string describing it. These two elements are mandatory. The third list element is optional and allows additional command definition. The authorized element for this `dict` are summarized in the following array:

key	Value	Definition
"display level"	DispLevel enum value	The command display level
"polling period"	Any number	The command polling period (mS)
"default command"	True or False	To define that it is the default command

6.16.4 Defining attributes

As shown in the previous example, attributes have to be defined in a `dict` called **attr_list** as a data member of the xxxClass class of the Tango class. This `dict` has one element per attribute. The element key is the attribute name. The element value is a python `list` which defines the attribute. The generic form of an attribute definition is:

```
'attr_name' :  [ [mandatory parameters], <{opt parameters}>]
```

For any kind of attributes, the mandatory parameters are:

```
[attr data type, attr data format, attr data R/W type]
```

The attribute data type is one of the possible value for attributes of the `tango.ArgType` pseudo enunmeration. The attribute data format is one of the possible value of the *`tango.AttrDataFormat`* pseudo enumeration and the attribute R/W type is one of the possible value of the *`tango.AttrWriteType`* pseudo enumeration. For spectrum attribute, you have to add the maximum X size (a number). For image attribute, you have to add the maximun X and Y dimension (two numbers). The authorized elements for the `dict` defining optional parameters are summarized in the following array:

key	value	definition
"display level"	tango.DispLevel enum value	The attribute display level
"polling period"	Any number	The attribute polling period (mS)
"memorized"	"true" or "true_without_hard_applied"	Define if and how the att. is memorized
"label"	A string	The attribute label
"description"	A string	The attribute description
"unit"	A string	The attribute unit
"standard unit"	A number	The attribute standard unit
"display unit"	A string	The attribute display unit
"format"	A string	The attribute display format
"max value"	A number	The attribute max value
"min value"	A number	The attribute min value
"max alarm"	A number	The attribute max alarm
"min alarm"	A number	The attribute min alarm
"min warning"	A number	The attribute min warning
"max warning"	A number	The attribute max warning
"delta time"	A number	The attribute RDS alarm delta time
"delta val"	A number	The attribute RDS alarm delta val

6.16.5 The PyDsExp class in Python

The rule of this class is to implement methods executed by commands and attributes. In our example, the code of this class looks like:

```
class PyDsExp(tango.Device_4Impl):

    def __init__(self,cl,name):
        tango.Device_4Impl.__init__(self, cl, name)
        self.info_stream('In PyDsExp.__init__')
        PyDsExp.init_device(self)

```

```
    def init_device(self):
        self.info_stream('In Python init_device method')
        self.set_state(tango.DevState.ON)
        self.attr_short_rw = 66
        self.attr_long = 1246

    #------------------------------------------------------------------

    def delete_device(self):
        self.info_stream('PyDsExp.delete_device')

    #------------------------------------------------------------------
    # COMMANDS
    #------------------------------------------------------------------

    def is_IOLong_allowed(self):
        return self.get_state() == tango.DevState.ON

    def IOLong(self, in_data):
        self.info_stream('IOLong', in_data)
        in_data = in_data * 2
        self.info_stream('IOLong returns', in_data)
        return in_data

    #------------------------------------------------------------------

    def is_IOStringArray_allowed(self):
        return self.get_state() == tango.DevState.ON

    def IOStringArray(self, in_data):
        l = range(len(in_data)-1, -1, -1)
        out_index=0
        out_data=[]
        for i in l:
            self.info_stream('IOStringArray <-', in_data[out_index])
            out_data.append(in_data[i])
            self.info_stream('IOStringArray ->',out_data[out_index])
            out_index += 1
        self.y = out_data
        return out_data

    #------------------------------------------------------------------
    # ATTRIBUTES
    #------------------------------------------------------------------

    def read_attr_hardware(self, data):
        self.info_stream('In read_attr_hardware')

    def read_Long_attr(self, the_att):
        self.info_stream("read_Long_attr")

        the_att.set_value(self.attr_long)

    def is_Long_attr_allowed(self, req_type):
        return self.get_state() in (tango.DevState.ON,)

    def read_Short_attr_rw(self, the_att):
        self.info_stream("read_Short_attr_rw")

        the_att.set_value(self.attr_short_rw)

    def write_Short_attr_rw(self, the_att):
        self.info_stream("write_Short_attr_rw")
```

```
71
72          self.attr_short_rw = the_att.get_write_value()
73
74      def is_Short_attr_rw_allowed(self, req_type):
75          return self.get_state() in (tango.DevState.ON,)
```

Line 1 The PyDsExp class has to inherit from the tango.Device_4Impl

Line 3 to 6 PyDsExp class constructor. Note that at line 6, it calls the *init_device()* method

Line 8 to 12 The init_device() method. It sets the device state (line 9) and initialises some data members

Line 16 to 17 The delete_device() method. This method is not mandatory. You define it only if you have to do something specific before the device is destroyed

Line 23 to 30 The two methods for the *IOLong* command. The first method is called *is_IOLong_allowed()* and it is the command is_allowed method (line 23 to 24). The second method has the same name than the command name. It is the method which executes the command. The command input data type is a Tango long and therefore, this method receives a python integer.

Line 34 to 47 The two methods for the *IOStringArray* command. The first method is its is_allowed method (Line 34 to 35). The second one is the command execution method (Line 37 to 47). The command input data type is a string array. Therefore, the method receives the array in a python list of python strings.

Line 53 to 54 The *read_attr_hardware()* method. Its argument is a Python sequence of Python integer.

Line 56 to 59 The method executed when the *Long_attr* attribute is read. Note that before PyTango 7 it sets the attribute value with the tango.set_attribute_value function. Now the same can be done using the set_value of the attribute object

Line 61 to 62 The is_allowed method for the *Long_attr* attribute. This is an optional method that is called when the attribute is read or written. Not defining it has the same effect as always returning True. The parameter req_type is of type `AttReqtype` which tells if the method is called due to a read or write request. Since this is a read-only attribute, the method will only be called for read requests, obviously.

Line 64 to 67 The method executed when the *Short_attr_rw* attribute is read.

Line 69 to 72 The method executed when the Short_attr_rw attribute is written. Note that before PyTango 7 it gets the attribute value with a call to the Attribute method *get_write_value* with a list as argument. Now the write value can be obtained as the return value of the *get_write_value* call. And in case it is a scalar there is no more the need to extract it from the list.

Line 74 to 75 The is_allowed method for the *Short_attr_rw* attribute. This is an optional method that is called when the attribute is read or written. Not defining it has the same effect as always returning True. The parameter req_type is of type `AttReqtype` which tells if the method is called due to a read or write request.

General methods

The following array summarizes how the general methods we have in a Tango device server are implemented in Python.

Name	Input par (with "self")	return value	mandatory
init_device	None	None	Yes
delete_device	None	None	No
always_executed_hook	None	None	No
signal_handler	`int`	None	No
read_attr_hardware	sequence<`int`>	None	No

Implementing a command

Commands are defined as described above. Nevertheless, some methods implementing them have to be written. These methods names are fixed and depend on command name. They have to be called:

- `is_<Cmd_name>_allowed(self)`
- `<Cmd_name>(self, arg)`

For instance, with a command called *MyCmd*, its is_allowed method has to be called *is_MyCmd_allowed* and its execution method has to be called simply *MyCmd*. The following array gives some more info on these methods.

Name	Input par (with "self")	return value	mandatory
is_<Cmd_name>_allowed	None	Python boolean	No
Cmd_name	Depends on cmd type	Depends on cmd type	Yes

Please check *Data types* chapter to understand the data types that can be used in command parameters and return values.

The following code is an example of how you write code executed when a client calls a command named IOLong:

```
1  def is_IOLong_allowed(self):
2      self.debug_stream("in is_IOLong_allowed")
3      return self.get_state() == tango.DevState.ON
4
5  def IOLong(self, in_data):
6      self.info_stream('IOLong', in_data)
7      in_data = in_data * 2
8      self.info_stream('IOLong returns', in_data)
9      return in_data
```

Line 1-3 the is_IOLong_allowed method determines in which conditions the command 'IOLong' can be executed. In this case, the command can only be executed if the device is in 'ON' state.

Line 6 write a log message to the tango INFO stream (click *here* for more information about PyTango log system).

Line 7 does something with the input parameter

Line 8 write another log message to the tango INFO stream (click *here* for more information about PyTango log system).

Line 9 return the output of executing the tango command

Implementing an attribute

Attributes are defined as described in chapter 5.3.2. Nevertheless, some methods implementing them have to be written. These methods names are fixed and depend on attribute name. They have to be called:

- `is_<Attr_name>_allowed(self, req_type)`
- `read_<Attr_name>(self, attr)`
- `write_<Attr_name>(self, attr)`

For instance, with an attribute called *MyAttr*, its is_allowed method has to be called *is_MyAttr_allowed*, its read method has to be called *read_MyAttr* and its write method has to be called *write_MyAttr*. The *attr* parameter is an instance of `Attr`. Unlike the commands, the is_allowed method for attributes receives a parameter of type `AttReqtype`.

Please check *Data types* chapter to understand the data types that can be used in attribute.

The following code is an example of how you write code executed when a client read an attribute which is called *Long_attr*:

```
1 def read_Long_attr(self, the_att):
2     self.info_stream("read attribute name Long_attr")
3     the_att.set_value(self.attr_long)
```

Line 1 Method declaration with "the_att" being an instance of the Attribute class representing the Long_attr attribute

Line 2 write a log message to the tango INFO stream (click *here* for more information about PyTango log system).

Line 3 Set the attribute value using the method set_value() with the attribute value as parameter.

The following code is an example of how you write code executed when a client write the Short_attr_rw attribute:

```
1 def write_Short_attr_rw(self,the_att):
2     self.info_stream("In write_Short_attr_rw for attribute ",the_att.get_name())
3     self.attr_short_rw = the_att.get_write_value(data)
```

Line 1 Method declaration with "the_att" being an instance of the Attribute class representing the Short_attr_rw attribute

Line 2 write a log message to the tango INFO stream (click *here* for more information about PyTango log system).

Line 3 Get the value sent by the client using the method get_write_value() and store the value written in the device object. Our attribute is a scalar short attribute so the return value is an int

CHAPTER 7

FAQ

Answers to general Tango questions can be found in the general tango tutorial.

Please also check the general tango how to.

How can I report an issue?

Bug reports are very valuable for the community.

Please open a new issue on the github issue page.

How can I contribute to PyTango and the documentation?

Contribution are always welcome!

You can open pull requests on the github PR page.

I got a libbost_python error when I try to import tango module...

For instance:

```
>>> import tango
ImportError: libboost_python.so.1.53.0: cannot open shared object file: No such file or directory
```

You must check that you have the correct boost python installed on your computer. To see which boost python file PyTango needs, type:

```
$ ldd /usr/lib64/python2.7/site-packages/tango/_tango.so
    linux-vdso.so.1 =>  (0x00007ffea7562000)
    libtango.so.9 => /lib64/libtango.so.9 (0x00007fac04011000)
    libomniORB4.so.1 => /lib64/libomniORB4.so.1 (0x00007fac03c62000)
    libboost_python.so.1.53.0 => not found
    [...]
```

I have more questions, where can I ask?

The Tango forum is a good place to get some support. Meet us in the Python section.

CHAPTER 8

PyTango Enhancement Proposals

8.1 TEP 1 - Device Server High Level API

TEP:	1
Title:	Device Server High Level API
Version:	2.2.0
Last-Modified:	10-Sep-2014
Author:	Tiago Coutinho <tcoutinho@cells.es>
Status:	Active
Type:	Standards Track
Content-Type:	text/x-rst
Created:	17-Oct-2012

8.1.1 Abstract

This TEP aims to define a new high level API for writting device servers.

8.1.2 Rationale

The code for Tango device servers written in Python often obey a pattern. It would be nice if non tango experts could create tango device servers without having to code some obscure tango related code. It would also be nice if the tango programming interface would be more pythonic. The final goal is to make writting tango device servers as easy as:

```
class Motor(Device):
    __metaclass__ = DeviceMeta

    position = attribute()

    def read_position(self):
        return 2.3

    @command()
    def move(self, position):
        pass

if __name__ == "__main__":
    server_run((Motor,))
```

8.1.3 Places to simplify

After looking at most python device servers one can see some patterns:

At <*Device*> class level:

1. <Device> always inherits from latest available DeviceImpl from pogo version
2. **constructor always does the same:**
   (a) calls super constructor
   (b) debug message
   (c) calls init_device
3. all methods have debug_stream as first instruction
4. init_device does additionaly get_device_properties()
5. *read attribute* methods follow the pattern:

```
def read_Attr(self, attr):
  self.debug_stream()
  value = get_value_from_hardware()
  attr.set_value(value)
```

6. *write attribute* methods follow the pattern:

```
def write_Attr(self, attr):
  self.debug_stream()
  w_value = attr.get_write_value()
  apply_value_to_hardware(w_value)
```

At <*Device*>*Class* class level:

1. A <Device>Class class exists for every <DeviceName> class
2. The <Device>Class class only contains attributes, commands and properties descriptions (no logic)
3. The attr_list description always follows the same (non explicit) pattern (and so does cmd_list, class_property_list, device_property_list)
4. the syntax for attr_list, cmd_list, etc is far from understandable

At *main()* level:

1. **The main() method always does the same:**
   (a) create *Util*
   (b) register tango class
   (c) when registering a python class to become a tango class, 99.9% of times the python class name is the same as the tango class name (example: Motor is registered as tango class "Motor")
   (d) call *server_init()*
   (e) call *server_run()*

8.1.4 High level API

The goals of the high level API are:

Maintain all features of low-level API available from high-level API

Everything that was done with the low-level API must also be possible to do with the new API.

All tango features should be available by direct usage of the new simplified, cleaner high-level API and through direct access to the low-level API.

Automatic inheritance from the latest** `DeviceImpl`

Currently Devices need to inherit from a direct Tango device implementation (*DeviceImpl*, or *Device_2Impl*, *Device_3Impl*, *Device_4Impl*, etc) according to the tango version being used during the development.

In order to keep the code up to date with tango, every time a new Tango IDL is released, the code of **every** device server needs to be manually updated to ihnerit from the newest tango version.

By inheriting from a new high-level *Device* (which itself automatically *decides* from which DeviceImpl version it should inherit), the device servers are always up to date with the latest tango release without need for manual intervention (see *tango.server*).

Low-level way:

```
class Motor(PyTango.Device_4Impl):
    pass
```

High-level way:

```
class Motor(PyTango.server.Device):
    pass
```

Default implementation of `Device` constructor

99% of the different device classes which inherit from low level *DeviceImpl* only implement *__init__* to call their *init_device* (see *tango.server*).

Device already calls init_device.

Low-level way:

```
class Motor(PyTango.Device_4Impl):

    def __init__(self, dev_class, name):
        PyTango.Device_4Impl.__init__(self, dev_class, name)
        self.init_device()
```

High-level way:

```
class Motor(PyTango.server.Device):

    # Nothing to be done!

    pass
```

Default implementation of `init_device()`

99% of different device classes which inherit from low level *DeviceImpl* have an implementation of *init_device* which *at least* calls *get_device_properties()* (see *tango.server*).

`init_device()` already calls `get_device_properties()`.

Low-level way:

```
class Motor(PyTango.Device_4Impl):

    def init_device(self):
        self.get_device_properties()
```

High-level way:

```
class Motor(PyTango.server.Device):
    # Nothing to be done!
    pass
```

Remove the need to code `DeviceClass`

99% of different device servers only need to implement their own subclass of *DeviceClass* to register the attribute, commands, device and class properties by using the corresponding `attr_list`, `cmd_list`, `device_property_list` and `class_property_list`.

With the high-level API we completely remove the need to code the *DeviceClass* by registering attribute, commands, device and class properties in the *Device* with a more pythonic API (see *tango.server*)

1. Hide *<Device>Class* class completely
2. simplify *main()*

Low-level way:

```
class Motor(PyTango.Device_4Impl):

    def read_Position(self, attr):
        pass

class MotorClass(PyTango.DeviceClass):

    class_property_list = { }
    device_property_list = { }
    cmd_list = { }

    attr_list = {
        'Position':
            [[PyTango.DevDouble,
            PyTango.SCALAR,
            PyTango.READ]],
        }

    def __init__(self, name):
        PyTango.DeviceClass.__init__(self, name)
        self.set_type(name)
```

High-level way:

```
class Motor(PyTango.server.Device):

    position = PyTango.server.attribute(dtype=float, )

    def read_position(self):
        pass
```

Pythonic read/write attribute

With the low level API, it feels strange for a non tango programmer to have to write:

```
def read_Position(self, attr):
    # ...
    attr.set_value(new_position)

def read_Position(self, attr):
    # ...
    attr.set_value_date_quality(new_position, time.time(), AttrQuality.CHANGING)
```

A more pythonic away would be:

```
def read_position(self):
    # ...
    self.position = new_position

def read_position(self):
    # ...
    self.position = new_position, time.time(), AttrQuality.CHANGING
```

Or even:

```
def read_position(self):
    # ...
    return new_position

def read_position(self):
    # ...
    return new_position, time.time(), AttrQuality.CHANGING
```

Simplify *main()*

the typical *main()* method could be greatly simplified. initializing tango, registering tango classes, initializing and running the server loop and managing errors could all be done with the single function call to `server_run()`

Low-level way:

```
def main():
    try:
        py = PyTango.Util(sys.argv)
        py.add_class(MotorClass,Motor,'Motor')

        U = PyTango.Util.instance()
        U.server_init()
        U.server_run()

    except PyTango.DevFailed,e:
        print '-------> Received a DevFailed exception:',e
    except Exception,e:
        print '-------> An unforeseen exception occured....',e
```

High-level way:

```
def main():
    classes = Motor,
    PyTango.server_run(classes)
```

8.1.5 In practice

Currently, a pogo generated device server code for a Motor having a double attribute *position* would look like this:

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-


##############################################################################
## license :
##============================================================================
##
## File :        Motor.py
##
## Project :
##
## $Author :      t$
##
## $Revision :    $
##
## $Date :        $
##
## $HeadUrl :     $
##============================================================================
##            This file is generated by POGO
##    (Program Obviously used to Generate tango Object)
##
##        (c) - Software Engineering Group - ESRF
##############################################################################

"""""""

__all__ = ["Motor", "MotorClass", "main"]

__docformat__ = 'restructuredtext'

import PyTango
import sys
# Add additional import
#----- PROTECTED REGION ID(Motor.additionnal_import) ENABLED START -----#

#----- PROTECTED REGION END -----#  //      Motor.additionnal_import

##############################################################################
## Device States Description
##
## No states for this device
##############################################################################

class Motor (PyTango.Device_4Impl):

#--------- Add you global variables here --------------------------
#----- PROTECTED REGION ID(Motor.global_variables) ENABLED START -----#

#----- PROTECTED REGION END -----#  //      Motor.global_variables
#------------------------------------------------------------------
#    Device constructor
#------------------------------------------------------------------
    def __init__(self,cl, name):
        PyTango.Device_4Impl.__init__(self,cl,name)
        self.debug_stream("In " + self.get_name() + ".__init__()")
        Motor.init_device(self)
```

```
#------------------------------------------------------------------
#    Device destructor
#------------------------------------------------------------------
    def delete_device(self):
        self.debug_stream("In " + self.get_name() + ".delete_device()")
        #----- PROTECTED REGION ID(Motor.delete_device) ENABLED START -----#

        #----- PROTECTED REGION END -----#  //      Motor.delete_device

#------------------------------------------------------------------
#    Device initialization
#------------------------------------------------------------------
    def init_device(self):
        self.debug_stream("In " + self.get_name() + ".init_device()")
        self.get_device_properties(self.get_device_class())
        self.attr_Position_read = 0.0
        #----- PROTECTED REGION ID(Motor.init_device) ENABLED START -----#

        #----- PROTECTED REGION END -----#  //      Motor.init_device

#------------------------------------------------------------------
#    Always excuted hook method
#------------------------------------------------------------------
    def always_executed_hook(self):
        self.debug_stream("In " + self.get_name() + ".always_excuted_hook()")
        #----- PROTECTED REGION ID(Motor.always_executed_hook) ENABLED START -----#

        #----- PROTECTED REGION END -----#  //      Motor.always_executed_hook

#==================================================================
#
#    Motor read/write attribute methods
#
#==================================================================

#------------------------------------------------------------------
#    Read Position attribute
#------------------------------------------------------------------
    def read_Position(self, attr):
        self.debug_stream("In " + self.get_name() + ".read_Position()")
        #----- PROTECTED REGION ID(Motor.Position_read) ENABLED START -----#
        self.attr_Position_read = 1.0
        #----- PROTECTED REGION END -----#  //      Motor.Position_read
        attr.set_value(self.attr_Position_read)

#------------------------------------------------------------------
#    Read Attribute Hardware
#------------------------------------------------------------------
    def read_attr_hardware(self, data):
        self.debug_stream("In " + self.get_name() + ".read_attr_hardware()")
        #----- PROTECTED REGION ID(Motor.read_attr_hardware) ENABLED START -----#

        #----- PROTECTED REGION END -----#  //      Motor.read_attr_hardware


#==================================================================
#
#    Motor command methods
#
#==================================================================
```

```
#==================================================================
#
#    MotorClass class definition
#
#==================================================================
class MotorClass(PyTango.DeviceClass):

    #    Class Properties
    class_property_list = {
        }


    #    Device Properties
    device_property_list = {
        }


    #    Command definitions
    cmd_list = {
        }


    #    Attribute definitions
    attr_list = {
        'Position':
            [[PyTango.DevDouble,
            PyTango.SCALAR,
            PyTango.READ]],
        }


#------------------------------------------------------------------
#    MotorClass Constructor
#------------------------------------------------------------------
    def __init__(self, name):
        PyTango.DeviceClass.__init__(self, name)
        self.set_type(name);
        print "In Motor Class  constructor"

#==================================================================
#
#    Motor class main method
#
#==================================================================
def main():
    try:
        py = PyTango.Util(sys.argv)
        py.add_class(MotorClass,Motor,'Motor')

        U = PyTango.Util.instance()
        U.server_init()
        U.server_run()

    except PyTango.DevFailed,e:
        print '-------> Received a DevFailed exception:',e
    except Exception,e:
        print '-------> An unforeseen exception occured....',e

if __name__ == '__main__':
    main()
```

To make things more fair, let's analyse the stripified version of the code instead:

```
import PyTango
import sys

class Motor (PyTango.Device_4Impl):

    def __init__(self,cl, name):
        PyTango.Device_4Impl.__init__(self,cl,name)
        self.debug_stream("In " + self.get_name() + ".__init__()")
        Motor.init_device(self)

    def delete_device(self):
        self.debug_stream("In " + self.get_name() + ".delete_device()")

    def init_device(self):
        self.debug_stream("In " + self.get_name() + ".init_device()")
        self.get_device_properties(self.get_device_class())
        self.attr_Position_read = 0.0

    def always_executed_hook(self):
        self.debug_stream("In " + self.get_name() + ".always_excuted_hook()")

    def read_Position(self, attr):
        self.debug_stream("In " + self.get_name() + ".read_Position()")
        self.attr_Position_read = 1.0
        attr.set_value(self.attr_Position_read)

    def read_attr_hardware(self, data):
        self.debug_stream("In " + self.get_name() + ".read_attr_hardware()")


class MotorClass(PyTango.DeviceClass):

    class_property_list = {
        }


    device_property_list = {
        }


    cmd_list = {
        }


    attr_list = {
        'Position':
            [[PyTango.DevDouble,
            PyTango.SCALAR,
            PyTango.READ]],
        }

    def __init__(self, name):
        PyTango.DeviceClass.__init__(self, name)
        self.set_type(name);
        print "In Motor Class  constructor"


def main():
    try:
        py = PyTango.Util(sys.argv)
        py.add_class(MotorClass,Motor,'Motor')
```

```
        U = PyTango.Util.instance()
        U.server_init()
        U.server_run()

    except PyTango.DevFailed,e:
        print '-------> Received a DevFailed exception:',e
    except Exception,e:
        print '-------> An unforeseen exception occured....',e

if __name__ == '__main__':
    main()
```

And the equivalent HLAPI version of the code would be:

```
#!/usr/bin/env python

from PyTango import DebugIt, server_run
from PyTango.server import Device, DeviceMeta, attribute


class Motor(Device):
    __metaclass__ = DeviceMeta

    position = attribute()

    @DebugIt()
    def read_position(self):
        return 1.0

def main():
    server_run((Motor,))

if __name__ == "__main__":
    main()
```

8.1.6 References

tango.server

8.1.7 Changes

from 2.1.0 to 2.2.0

Changed module name from *hlapi* to *server*

from 2.0.0 to 2.1.0

Changed module name from *api2* to *hlapi* (High Level API)

From 1.0.0 to 2.0.0

- **API Changes**
  - changed Attr to attribute
  - changed Cmd to command
  - changed Prop to device_property

  - changed ClassProp to class_property
- Included command and properties in the example
- Added references to API documentation

8.1.8 Copyright

This document has been placed in the public domain.

8.2 TEP 2 - Tango database serverless

TEP:	2
Title:	Tango database serverless
Version:	1.0.0
Last-Modified:	17-Oct-2012
Author:	Tiago Coutinho <tcoutinho@cells.es>
Status:	Active
Type:	Standards Track
Content-Type:	text/x-rst
Created:	17-Oct-2012
Post-History:	17-Oct-2012

8.2.1 Abstract

This TEP aims to define a python DataBaseds which doesn't need a database server behind. It would make tango easier to try out by anyone and it could greatly simplify tango installation on small environments (like small, independent laboratories).

8.2.2 Motivation

I was given a openSUSE laptop so that I could do the presentation for the tango meeting held in FRMII on October 2012. Since I planned to do a demonstration as part of the presentation I installed all mysql libraries, omniorb, tango and pytango on this laptop.

During the flight to Munich I realized tango was not working because of a strange mysql server configuration done by the openSUSE distribution. I am not a mysql expert and I couldn't google for a solution. Also it made me angry to have to install all the mysql crap (libmysqlclient, mysqld, mysql-administrator, bla, bla) just to have a demo running.

At the time of writting the first version of this TEP I still didn't solve the problem! Shame on me!

Also at the same tango meetting during the tango archiving discussions I heard fake whispers or changing the tango archiving from MySQL/Oracle to NoSQL.

I started thinking if it could be possible to have an alternative implementation of DataBaseds without the need for a mysql server.

8.2.3 Requisites

- no dependencies on external packages
- no need for a separate database server process (at least, by default)
- no need to execute post install scripts to fill database

8.2.4 Step 1 - Gather database information

It turns out that python has a Database API specification (**PEP 249**). Python distribution comes natively (>= 2.6) with not one but several persistency options (Data Persistence):

module	Native	Platforms	API	Database	Description
Native python 2.x					
pickle	Yes	all	dump/load	file	python serialization/marchalling module
shelve	Yes	all	dict	file	high level persistent, dictionary-like object
marshal	Yes	all	dump/load	file	Internal Python object serialization
anydbm	Yes	all	dict	file	Generic access to DBM-style databases. Wrapper for dbhash, gdbm, dbm or dumbdbm
dbm	Yes	all	dict	file	Simple "database" interface
gdbm	Yes	unix	dict	file	GNU's reinterpretation of dbm
dbhash	Yes	unix?	dict	file	DBM-style interface to the BSD database library (needs bsddb). **Removed in python 3**
bsddb	Yes	unix?	dict	file	Interface to Berkeley DB library. **Removed in python 3**
dumbdbm	Yes	all	dict	file	Portable DBM implementation
sqlite3	Yes	all	DBAPI2	file, memory	DB-API 2.0 interface for SQLite databases
Native Python 3.x					
pickle	Yes	all	dump/load	file	python serialization/marchalling module
shelve	Yes	all	dict	file	high level persistent, dictionary-like object
marshal	Yes	all	dump/load	file	Internal Python object serialization
dbm	Yes	all	dict	file	Interfaces to Unix "databases". Wrapper for dbm.gnu, dbm.ndbm, dbm.dumb
dbm.gnu	Yes	unix	dict	file	GNU's reinterpretation of dbm
dbm.ndbm	Yes	unix	dict	file	Interface based on ndbm
dbm.dumb	Yes	all	dict	file	Portable DBM implementation
sqlite3	Yes	all	DBAPI2	file, memory	DB-API 2.0 interface for SQLite databases

third-party DBAPI2

- pyodbc
- mxODBC
- kinterbasdb
- mxODBC Connect
- MySQLdb
- psycopg
- pyPgSQL
- PySQLite
- adodbapi
- pymssql
- sapdbapi
- ibm_db
- InformixDB

third-party NOSQL

(these may or not have python DBAPI2 interface)

- CouchDB - couchdb.client
- MongoDB - pymongo - NoSQL database
- Cassandra - pycassa

third-party database abstraction layer

- SQLAlchemy - `sqlalchemy` - Python SQL toolkit and Object Relational Mapper

8.2.5 Step 2 - Which module to use?

herrrr... wrong question!

The first decision I thought it should made is which python module better suites the needs of this TEP. Then I realized I would fall into the same trap as the C++ DataBaseds: hard link the server to a specific database implementation (in their case MySQL).

I took a closer look at the tables above and I noticed that python persistent modules come in two flavors: dict and DBAPI2. So naturally the decision I thought it had to be made was: *which flavor to use?*

But then I realized both flavors could be used if we properly design the python DataBaseds.

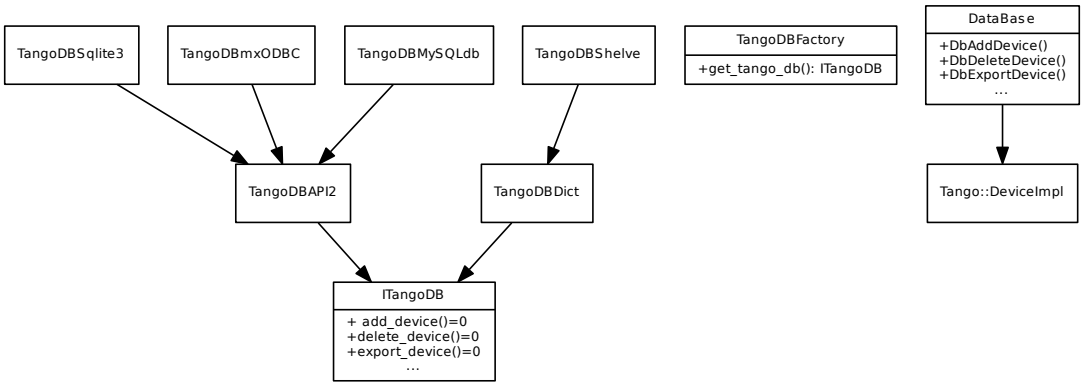
8.2.6 Step 3 - Architecture

If you step back for a moment and look at the big picture you will see that what we need is really just a mapping between the Tango DataBase set of attributes and commands (I will call this *Tango Device DataBase API*) and the python database API oriented to tango (I will call this TDB interface).

The TDB interface should be represented by the `ITangoDB`. Concrete databases should implement this interface (example, DBAPI2 interface should be represented by a class `TangoDBAPI2` implementing `ITangoDB`).

Connection to a concrete ITangoDB should be done through a factory: `TangoDBFactory`

The Tango DataBase device should have no logic. Through basic configuration it should be able to ask the `TangoDBFactory` for a concrete `ITangoDB`. The code of every command and attribute should be simple forward to the `ITangoDB` object (a part of some parameter translation and error handling).



8.2.7 Step 4 - The python DataBaseds

If we can make a python device server which has the same set of attributes and commands has the existing C++ DataBase (and of course the same semantic behavior), the tango DS and tango clients will never know the difference (BTW, that's one of the beauties of tango).

The C++ DataBase consists of around 80 commands and 1 mandatory attribute (the others are used for profiling) so making a python Tango DataBase device from scratch is out of the question.

Fortunately, C++ DataBase is one of the few device servers that were developed since the beginning with pogo and were successfully adapted to pogo 8. This means there is a precious `DataBase.xmi`

available which can be loaded to pogo and saved as a python version. The result of doing this can be found here `here` (this file was generated with a beta version of the pogo 8.1 python code generator so it may contain errors).

8.2.8 Step 5 - Default database implementation

The decision to which database implementation should be used should obey the following rules:

1. should not require an extra database server process
2. should be a native python module
3. should implement python DBAPI2

It came to my attention the `sqlite3` module would be perfect as a default database implementation. This module comes with python since version 2.5 and is available in all platforms. It implements the DBAPI2 interface and can store persistently in a common OS file or even in memory.

There are many free scripts on the web to translate a mysql database to sqlite3 so one can use an existing mysql tango database and directly use it with the python DataBaseds with sqlite3 implementation.

8.2.9 Development

The development is being done in PyTango SVN trunk in the `tango.databaseds` module.

You can checkout with:

```
$ svn co https://tango-cs.svn.sourceforge.net/svnroot/tango-cs/bindings/PyTango/trunk PyTango-tru
```

8.2.10 Disadvantages

A serverless, file based, database has some disadvantages when compared to the mysql solution:

- Not possible to distribute load between Tango DataBase DS and database server (example: run the Tango DS in one machine and the database server in another)
- Not possible to have two Tango DataBase DS pointing to the same database
- Harder to upgrade to newer version of sql tables (specially if using dict based database)

Bare in mind the purpose of this TED is to simplify the process of trying tango and to ease installation and configuration on small environments (like small, independent laboratories).

8.2.11 References

- http://wiki.python.org/moin/DbApiCheatSheet
- http://wiki.python.org/moin/DbApiModuleComparison
- http://wiki.python.org/moin/DatabaseProgramming
- http://wiki.python.org/moin/DbApiFaq
- **PEP 249**
- http://wiki.python.org/moin/ExtendingTheDbApi
- http://wiki.python.org/moin/DbApi3

CHAPTER 9

History of changes

Contributers T. Coutinho

Last Update January 30, 2017

9.1 Document revisions

Date	Revision	Description	Author
18/07/03	1.0	Initial Version	M. Ounsy
06/10/03	2.0	Extension of the "Getting Started" paragraph	A. Buteau/M. Ou
14/10/03	3.0	Added Exception Handling paragraph	M. Ounsy
13/06/05	4.0	Ported to Latex, added events, AttributeProxy and ApiUtil	V. Forchì
13/06/05	4.1	fixed bug with python 2.5 and and state events new Database constructor	V. Forchì
15/01/06	5.0	Added Device Server classes	E.Taurel
15/03/07	6.0	Added AttrInfoEx, AttributeConfig events, 64bits, write_attribute	T. Coutinho
21/03/07	6.1	Added groups	T. Coutinho
15/06/07	6.2	Added dynamic attributes doc	E. Taurel
06/05/08	7.0	Update to Tango 6.1. Added DB methods, version info	T. Coutinho
10/07/09	8.0	Update to Tango 7. Major refactoring. Migrated doc	T. Coutinho/R. S
24/07/09	8.1	Added migration info, added missing API doc	T. Coutinho/R. S
21/09/09	8.2	Added migration info, release of 7.0.0beta2	T. Coutinho/R. S
12/11/09	8.3	Update to Tango 7.1.	T. Coutinho/R. S
??/12/09	8.4	Update to PyTango 7.1.0 rc1	T. Coutinho/R. S
19/02/10	8.5	Update to PyTango 7.1.1	T. Coutinho/R. S
06/08/10	8.6	Update to PyTango 7.1.2	T. Coutinho
05/11/10	8.7	Update to PyTango 7.1.3	T. Coutinho
08/04/11	8.8	Update to PyTango 7.1.4	T. Coutinho
13/04/11	8.9	Update to PyTango 7.1.5	T. Coutinho
14/04/11	8.10	Update to PyTango 7.1.6	T. Coutinho
15/04/11	8.11	Update to PyTango 7.2.0	T. Coutinho
12/12/11	8.12	Update to PyTango 7.2.2	T. Coutinho
24/04/12	8.13	Update to PyTango 7.2.3	T. Coutinho
21/09/12	8.14	Update to PyTango 8.0.0	T. Coutinho
10/10/12	8.15	Update to PyTango 8.0.2	T. Coutinho
20/05/13	8.16	Update to PyTango 8.0.3	T. Coutinho
28/08/13	8.13	Update to PyTango 7.2.4	T. Coutinho
27/11/13	8.18	Update to PyTango 8.1.1	T. Coutinho
16/05/14	8.19	Update to PyTango 8.1.2	T. Coutinho
30/09/14	8.20	Update to PyTango 8.1.4	T. Coutinho
01/10/14	8.21	Update to PyTango 8.1.5	T. Coutinho

Continued on next p

Table 9.1 – continued from previous page

Date	Revision	Description	Author
05/02/15	8.22	Update to PyTango 8.1.6	T. Coutinho
03/02/16	8.23	Update to PyTango 8.1.8	T. Coutinho
12/08/16	8.24	Update to PyTango 8.1.9	V. Michel
26/02/16	9.2	Update to PyTango 9.2.0a	T. Coutinho
15/08/16	9.3	Update to PyTango 9.2.0	V. Michel

9.2 Version history

version	Changes
9.2.0	9.2.0 release. Features: • Issue #37: Add display_level and polling_period as optional arguments to command decorator Bug fixes: • Fix cache problem when using *DeviceProxy* through an *AttributeProxy* • Fix compilation on several platforms • Issue #19: Defining new members in DeviceProxy has side effects • Fixed bug in *beacon.add_device* • Fix for *get_device_list* if server_name is '*' • Fix *get_device_attribute_property2* if *prop_attr* is not *None* • Accept *StdStringVector* in *put_device_property* • Map 'int' to DevLong64 and 'uint' to DevULong64 • Issue #22: Fix push_data_ready_event() deadlock • Issue #28: Fix compilation error for constants.cpp • Issue #21: Fix Group.get_device method • Issue #33: Fix internal server documentation **Changes:** • Move ITango to another project • Use *setuptools* instead of *distutils* • Add *six* as a requirement • Refactor directory structure • Rename *PyTango* module to *tango* (*import PyTango* still works for backward compatibility) • Add a ReST readme for GitHub and PyPI **ITango changes (moved to another project):** • Fix itango event logger for python 3 • Avoid deprecation warning with IPython 4.x • Use entry points instead of scripts
9.2.0a	9.2 alpha release. Missing: • writtable pipes (client and server) • dynamic commands (server)
	• device interface change event (client and server) • pipe event (client and server) Bug fixes:

Last update: January 30, 2017

t

A

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W