PyTango Documentation

Release 9.3.6

PyTango team

Sep 29, 2022

CONTENTS

1 Getting started 1
1.1 Installing . . . 1
1.2 Compiling . . . 2
1.3 Testing . . . 2

2 Quick tour 3
2.1 Fundamental TANGO concepts . . . 3
2.2 Minimum setup . . . 4
2.3 Client . . . 4
2.4 Server . . . 6

3 ITango 11

4 Green mode 13
4.1 Client green modes . . . 13
4.2 Server green modes . . . 19

5 PyTango API 21
5.1 Data types . . . 21
5.2 Client API . . . 29
5.3 Server API . . . 109
5.4 Database API . . . 185
5.5 Encoded API . . . 219
5.6 The Utilities API . . . 226
5.7 Exception API . . . 231

6 How to 239
6.1 How to contribute . . . 239
6.2 Check the default TANGO host . . . 239
6.3 Check TANGO version . . . 239
6.4 Start server from command line . . . 240
6.5 Report a bug . . . 240
6.6 Test the connection to the Device and get it's current state . . . 241
6.7 Read and write attributes . . . 241
6.8 Execute commands . . . 242
6.9 Execute commands with more complex types . . . 243
6.10 Work with Groups . . . 243
6.11 Handle errors . . . 243
6.12 Registering devices . . . 243
6.13 Using clients with multiprocessing . . . 245
6.14 Using clients with multithreading . . . 246
6.15 Write a server . . . 247
6.16 Server logging . . . 249
6.17 Multiple device classes (Python and C++) in a server . . . 251
6.18 Create attributes dynamically . . . 252

6.19 Create/Delete devices dynamically . . . 255
6.20 Write a server (original API) . . . 256

7 How to Contribute **265**
7.1 Workflow . . . 265
7.2 reStructuredText and Sphinx . . . 265
7.3 Source code standard . . . 266
7.4 Using Docker for development . . . 266
7.5 Releasing a new version . . . 266

8 Testing PyTango Devices **269**
8.1 Approaches to testing Tango devices . . . 269
8.2 Device Test Context Classes API . . . 274
8.3 Mocking clients for Testing . . . 278

9 FAQ **285**

10 PyTango Enhancement Proposals **287**
10.1 TEP 1 - Device Server High Level API . . . 287
10.2 TEP 2 - Tango database serverless . . . 298

11 History of changes **303**
11.1 Document revisions . . . 303
11.2 Version history . . . 304

12 Indexes **305**

Python Module Index **307**

Index **309**

CHAPTER

ONE

GETTING STARTED

1.1 Installing

1.1.1 Linux

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

- for Python 2.7:

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

- for Python 3.X:

```
$ sudo apt-get install python3-tango
```

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://gitlab.com/tango-controls/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)"
9.3.4
```

Next steps: Check out the *Quick tour*.

CHAPTER

TWO

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)
```

(continues on next page)

(continued from previous page)

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

In our example we will write a fake power supply device server. 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 `power_supply.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, attribute, command, pipe, device_property


class PowerSupply(Device):

    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,
```

(continues on next page)

(continued from previous page)

```
                        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")
```

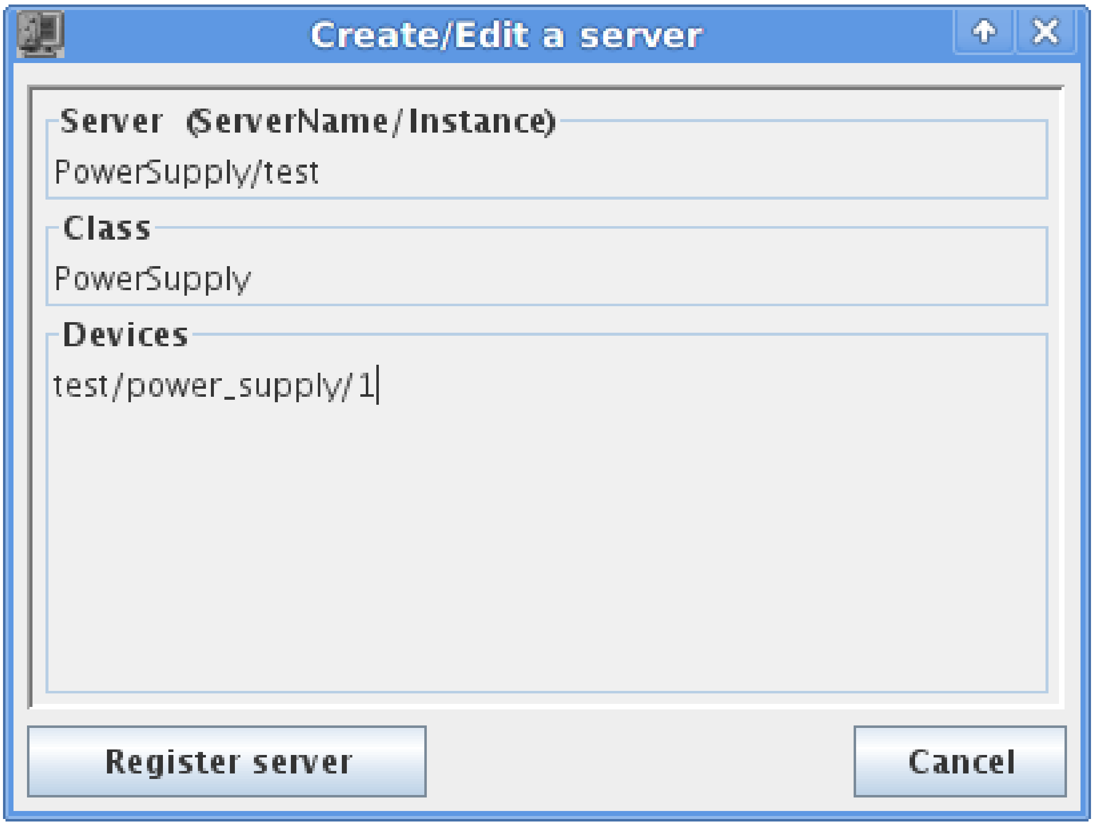
(continues on next page)

(continued from previous page)

```
    def Ramp(self, target_current):
        # should do the ramping
        return True


if __name__ == "__main__":
    PowerSupply.run_server()
```

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 = "PowerSupply/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 power_supply.py test
Ready to accept request
```

Now you can access it from a python console:

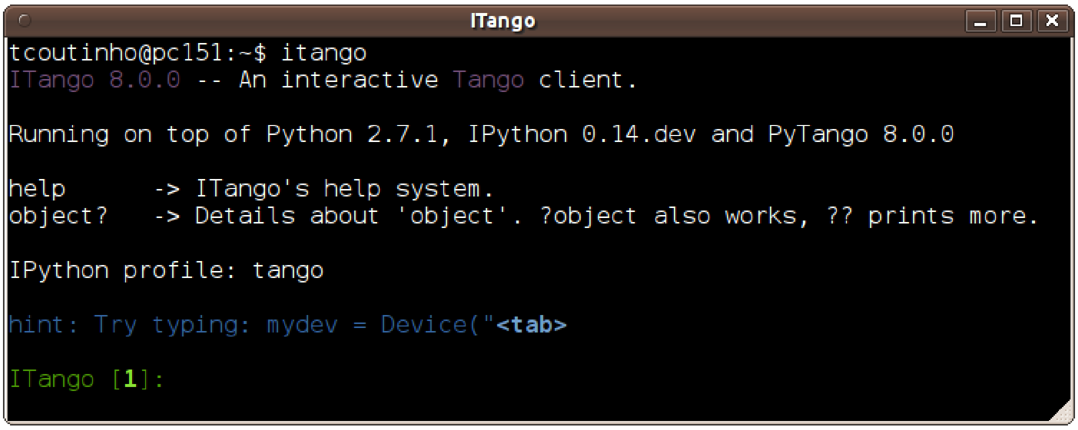
```
>>> import tango

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

>>> power_supply.current = 2.3

>>> power_supply.current
2.3

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

>>> power_supply.state()
ON
```

Note: In this example, the name of the server and the name of the tango class are the same: *PowerSupply*. This pattern is enforced by the `run_server()` method. However, it is possible to run several tango classes in the same server. In this case, the server name would typically be the name of server file. See the `run()` function for further information.

CHAPTER THREE

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 GitLab
- documentation on pythonhosted

CHAPTER

FOUR

GREEN MODE

PyTango supports cooperative green Tango objects. Since version 8.1 two *green* modes have been added: `Futures` and `Gevent`. In version 9.2.0 another one has been added: `Asyncio`.

Note: The preferred mode to use for new projects is `Asyncio`. Support for this mode will take priority over the others.

The `Futures` uses the standard python module `concurrent.futures`. The `Gevent` mode uses the well known gevent library. The newest, `Asyncio` mode, uses asyncio - a Python library for asynchronous programming (it's featured as a part of a standard Python distribution since version 3.5 of Python; it's available on PyPI for older ones).

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

4.1 Client green modes

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
```

4.1.1 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]
```

4.1.2 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).

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
```

(continues on next page)

(continued from previous page)

```
>>> 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`

4.1.3 asyncio mode

Asyncio mode is similar to gevent but it uses explicit coroutines. You can compare gevent and asyncio examples.

```
import asyncio
from tango.asyncio import DeviceProxy

async def asyncio_example():
    dev = await DeviceProxy("sys/tg_test/1")
    print(dev.get_green_mode())
```

(continues on next page)

(continued from previous page)

```python

    print(await dev.state())

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio_example())
loop.close()
```

Below you can find a TCP server example, which runs in an asynchronous mode and waits for a device's attribute name from a TCP client, then asks the device for a value and replies to the TCP client.

```python
"""A simple TCP server for Tango attributes.

It runs on all interfaces on port 8888:

   $ python tango_tcp_server.py
   Serving on 0.0.0.0 port 8888

It can be accessed using netcat:

   $ ncat localhost 8888
   >>> sys/tg_test/1/ampli
   0.0
   >>> sys/tg_test/1/state
   RUNNING
   >>> sys/tg_test/1/nope
   DevFailed[
   DevError[
        desc = Attribute nope is not supported by device sys/tg_test/1
      origin = AttributeProxy::real_constructor()
      reason = API_UnsupportedAttribute
    severity = ERR]
    ]
   >>> ...
"""

import asyncio
from tango.asyncio import AttributeProxy


async def handle_echo(reader, writer):
    # Write the cursor
    writer.write(b'>>> ')
    # Loop over client request
    async for line in reader:
        request = line.decode().strip()
        # Get attribute value using asyncio green mode
        try:
            proxy = await AttributeProxy(request)
            attr_value = await proxy.read()
            reply = str(attr_value.value)
        # Catch exception if something goes wrong
        except Exception as exc:
            reply = str(exc)
        # Reply to client
        writer.write(reply.encode() + b'\n' + b'>>> ')
    # Close communication
```

(continues on next page)

(continued from previous page)

```python
    writer.close()


async def start_serving():
    server = await asyncio.start_server(handle_echo, '0.0.0.0', 8888)
    print('Serving on {} port {}'.format(*server.sockets[0].getsockname()))
    return server


async def stop_serving(server):
    server.close()
    await server.wait_closed()


def main():
    # Start the server
    loop = asyncio.get_event_loop()
    server = loop.run_until_complete(start_serving())
    # Serve requests until Ctrl+C is pressed
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    # Close the server
    loop.run_until_complete(stop_serving(server))
    loop.close()


if __name__ == '__main__':
    main()
```

4.2 Server green modes

PyTango server API from version 9.2.0 supports two green modes: `Gevent` and `Asyncio`. Both can be used in writing new device servers in an asynchronous way.

Note: If your device server has multiple devices they must all use the same green mode.

4.2.1 gevent mode

This mode lets you convert your existing devices to asynchronous devices easily. You just add *green_mode = tango.GreenMode.Gevent* line to your device class. Consider this example:

```python
class GeventDevice(Device):
    green_mode = tango.GreenMode.Gevent
```

Every method in your device class will be treated as a coroutine implicitly. This can be beneficial, but also potentially dangerous as it is a lot harder to debug. You should use this green mode with care. `Gevent` green mode is useful when you don't want to change too much in your existing code (or you don't feel comfortable with writing syntax of asynchronous calls).

Another thing to keep in mind is that when using `Gevent` green mode is that the Tango monitor lock is disabled, so the client requests can be processed concurrently.

Greenlets can also be used to spawn tasks in the background.

4.2.2 asyncio mode

The way asyncio green mode on the server side works is it redirects all user code to an event loop. This means that all user methods become coroutines, so in Python > 3.5 you should define them with *async* keyword. In Python < 3.5, you should use a *@coroutine* decorator. This also means that in order to convert existing code of your devices to `Asyncio` green mode you will have to introduce at least those changes. But, of course, to truly benefit from this green mode (and asynchronous approach in general), you should introduce more far-fetched changes!

The main benefit of asynchronous programing approach is that it lets you control precisely when code is run sequentially without interruptions and when control can be given back to the event loop. It's especially useful if you want to perform some long operations and don't want to prevent clients from accessing other parts of your device (attributes, in particular). This means that in `Asyncio` green mode there is no monitor lock!

The example below shows how asyncio can be used to write an asynchronous Tango device:

```python
"""Demo Tango Device Server using asyncio green mode"""

import asyncio
from tango import DevState, GreenMode
from tango.server import Device, command, attribute


class AsyncioDevice(Device):
    green_mode = GreenMode.Asyncio

    async def init_device(self):
        await super().init_device()
        self.set_state(DevState.ON)

    @command
    async def long_running_command(self):
        self.set_state(DevState.OPEN)
        await asyncio.sleep(2)
        self.set_state(DevState.CLOSE)

    @command
    async def background_task_command(self):
        loop = asyncio.get_event_loop()
        future = loop.create_task(self.coroutine_target())

    async def coroutine_target(self):
        self.set_state(DevState.INSERT)
        await asyncio.sleep(15)
        self.set_state(DevState.EXTRACT)

    @attribute
    async def test_attribute(self):
        await asyncio.sleep(2)
        return 42


if __name__ == '__main__':
    AsyncioDevice.run_server()
```

CHAPTER

FIVE

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*))
DEV_ENUM (*New in PyTango 9.0*)	• `int` (for value) • `list` <`str`> (for enum_labels) **Note: direct attribute access via DeviceProxy will return enumerated** type `enum.IntEnum`. This type uses the package enum34.	• `int` (for value) • `list` <`str`> (for enum_labels) **Note: direct attribute access via DeviceProxy will return enumerated** type `enum.IntEnum`. Python < 3.4, uses the package enum34. Python >= 3.4, uses standard package enum.

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`

continues on next page

Table 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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	List	`list <int>`	`list <int>`
	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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`)

continues on next page

Table 1 – continued from previous page

Tango data type	ExtractAs	Data type (Python 2.x)	Data type (Python 3.x) (*New in PyTango 8.0*)
	Bytes	`bytes` (which is in fact equal to `str`)	`bytes`
	ByteArray	`bytearray`	`bytearray`
	String	`str`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`
	String	`str`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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`	`str` (decoded with *latin-1*, aka *ISO-8859-1*)
	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*)

continues on next page

Table 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_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*)
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)
)
```

(continues on next page)

(continued from previous page)

```
# 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.int32(111)},
                                                                      {'name
↪': '1_1_2DE', 'value': [3.33]}))
                                      })
  )},
  {'name': '2DE', 'value': (3,4,5,6), 'dtype': ('int32',) },
 )
)
```

5.1.2 DevEnum pythonic usage

When using regular tango DeviceProxy and AttributeProxy DevEnum is treated just like in cpp tango (see enumerated attributes for more info). However, since PyTango >= 9.2.5 there is a more pythonic way of using DevEnum data types if you use the *high level API*, both in server and client side.

In server side you can use python `enum.IntEnum` class to deal with DevEnum attributes:

```
import time
from enum import IntEnum
from tango import AttrWriteType
from tango.server import Device, attribute, command


class Noon(IntEnum):
    AM = 0  # DevEnum's must start at 0
    PM = 1  # and increment by 1


class DisplayType(IntEnum):
    ANALOG = 0  # DevEnum's must start at 0
    DIGITAL = 1  # and increment by 1


class Clock(Device):

    display_type = DisplayType(0)

    @attribute(dtype=float)
    def time(self):
        return time.time()

    gmtime = attribute(dtype=(int,), max_dim_x=9)

    def read_gmtime(self):
        return time.gmtime()

    @attribute(dtype=Noon)
    def noon(self):
        time_struct = time.gmtime(time.time())
        return Noon.AM if time_struct.tm_hour < 12 else Noon.PM

    display = attribute(dtype=DisplayType, access=AttrWriteType.READ_WRITE)

    def read_display(self):
        return self.display_type

    def write_display(self, display_type):
        self.display_type = display_type

    @command(dtype_in=float, dtype_out=str)
    def ctime(self, seconds):
        """
        Convert a time in seconds since the Epoch to a string in local␣
↪time.
        This is equivalent to asctime(localtime(seconds)). When the time␣
↪tuple
        is not present, current time as returned by localtime() is used.
```

(continues on next page)

(continued from previous page)

```
        """
        return time.ctime(seconds)

    @command(dtype_in=(int,), dtype_out=float)
    def mktime(self, tupl):
        return time.mktime(tuple(tupl))


if __name__ == "__main__":
    Clock.run_server()
```

On the client side you can also use a pythonic approach for using DevEnum attributes:

```
import sys
import PyTango

if len(sys.argv) != 2:
    print("must provide one and only one clock device name")
    sys.exit(1)

clock = PyTango.DeviceProxy(sys.argv[1])
t = clock.time
gmt = clock.gmtime
noon = clock.noon
display = clock.display
print(t)
print(gmt)
print(noon, noon.name, noon.value)
if noon == noon.AM:
    print('Good morning!')
print(clock.ctime(t))
print(clock.mktime(gmt))
print(display, display.name, display.value)
clock.display = display.ANALOG
clock.display = 'DIGITAL'  # you can use a valid string to set the value
print(clock.display, clock.display.name, clock.display.value)
display_type = type(display)  # or even create your own IntEnum type
analog = display_type(0)
clock.display = analog
print(clock.display, clock.display.name, clock.display.value)
clock.display = clock.display.DIGITAL
print(clock.display, clock.display.name, clock.display.value)
```

5.2 Client API

5.2.1 DeviceProxy

class tango.**DeviceProxy**(**args*, ***kwargs*)

Bases: `Connection`

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.

add_logging_target(*self*, *target_type_target_name*) → None

Adds a new logging target to the device.

The target_type_target_name input parameter must follow the format: target_type::target_name. Supported target types are: console, file and device. For a device target, the target_name part of the target_type_target_name parameter must contain the name of a log consumer device (as defined in A.8). For a file target, target_name is the full path to the file to log to. If omitted, the device's name is used to build the file name (which is something like domain_family_member.log). Finally, the target_name part of the target_type_target_name input parameter is ignored in case of a console target and can be omitted.

Parameters

target_type_target_name
(`str`) logging target

Return
None

Throws
DevFailed from device

New in PyTango 7.0.0

adm_name (*self*) → str

Return the name of the corresponding administrator device. This is useful if you need to send an administration command to the device server, e.g restart it

New in PyTango 3.0.4

alias (*self*) → str

Return the device alias if one is defined. Otherwise, throws exception.

Return
(`str`) device alias

attribute_history (*self, attr_name, depth, extract_as=ExtractAs.Numpy*) → sequence<DeviceAttributeHistory>

Retrieve attribute history from the attribute polling buffer. See chapter on Advanced Feature for all details regarding polling

Parameters

attr_name
(`str`) Attribute name.

depth
(`int`) The wanted history depth.

extract_as
(`ExtractAs`)

Return
This method returns a vector of DeviceAttributeHistory types.

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

attribute_list_query (*self*) → sequence<AttributeInfo>

Query the device for info on all attributes. This method returns a sequence of tango.AttributeInfo.

Parameters
None

Return
(sequence<*AttributeInfo*>) containing the attributes configuration

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

attribute_list_query_ex (*self*) → sequence<AttributeInfoEx>

Query the device for info on all attributes. This method returns a sequence of tango.AttributeInfoEx.

Parameters
None

Return
(sequence<*AttributeInfoEx*>) containing the attributes configuration

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

attribute_query (*self, attr_name*) → *AttributeInfoEx*

Query the device for information about a single attribute.

Parameters

attr_name
(str) the attribute name

Return
(*AttributeInfoEx*) containing the attribute configuration

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

black_box (*self, n*) → sequence<str>

Get the last commands executed on the device server

Parameters

n
n number of commands to get

Return
(sequence<str>) sequence of strings containing the date, time, command and from which client computer the command was executed

Example

```
print(black_box(4))
```

cancel_all_polling_asynch_request (*self*) → None

Cancel all running asynchronous request

This is a client side call. Obviously, the calls cannot be aborted while it is running in the device.

Parameters
None

Return
None

New in PyTango 7.0.0

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

Cancel a running asynchronous request

This is a client side call. Obviously, the call cannot be aborted while it is running in the device.

Parameters

id
The asynchronous call identifier

Return
None

New in PyTango 7.0.0

command_history (*self*, *cmd_name*, *depth*) → sequence<DeviceDataHistory>

Retrieve command history from the command polling buffer. See chapter on Advanced Feature for all details regarding polling

Parameters

cmd_name
(`str`) Command name.

depth
(`int`) The wanted history depth.

Return
This method returns a vector of DeviceDataHistory types.

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

command_inout (*self*, *cmd_name*, *cmd_param=None*, *green_mode=None*, *wait=True*, *timeout=None*) → any

Execute a command on a device.

Parameters

cmd_name
(`str`) Command name.

cmd_param
(`any`) It should be a value of the type expected by the command or a DeviceData object with this value inserted. It can be ommited if the command should not get any argument.

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
The result of the command. The type depends on the command. It may be None.

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.

command_inout_asynch(*self*, *cmd_name*) → id

command_inout_asynch(*self*, *cmd_name*, *cmd_param*) → id

command_inout_asynch(*self*, *cmd_name*, *cmd_param*, *forget*) → id

Execute asynchronously (polling model) a command on a device

Parameters

cmd_name
(`str`) Command name.

cmd_param
(`any`) It should be a value of the type expected by the command or a DeviceData object with this value inserted. It can be ommited if the command should not get any argument. If the command should get no argument and you want to set the 'forget' param, use None for cmd_param.

forget
(`bool`) If this flag is set to true, this means that the client does not care at all about the server answer and will even not try to get it. Default value is False. Please, note that device re-connection will not take place (in case it is needed) if the fire and forget mode is used. Therefore, an application using only fire and forget requests is not able to automatically re-connect to device.

Return
(`int`) This call returns an asynchronous call identifier which is needed to get the command result (see command_inout_reply)

Throws
ConnectionFailed, TypeError, anything thrown by command_query

command_inout_asynch(self, cmd_name, `callback`) -> None

command_inout_asynch(self, cmd_name, cmd_param, `callback`) -> None

Execute asynchronously (`callback` model) a command on a device.

Parameters

cmd_name
(`str`) Command name.

cmd_param
(any)It should be a value of the type expected by the command or a DeviceData object with this value inserted. It can be ommited if the command should not get any argument.

callback
Any callable object (function, lambda...) or any oject with a method named "cmd_ended".

Return
None

Throws
`ConnectionFailed`, TypeError, anything thrown by command_query

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.` `:class:`ApiUtil()`.set_asynch_cb_sub_model`

---

command_inout_raw(*self, cmd_name, cmd_param=None*) → *DeviceData*

Execute a command on a device.

Parameters

cmd_name
(`str`) Command name.

cmd_param
(`any`) It should be a value of the type expected by the command or a DeviceData object with this value inserted. It can be ommited if the command should not get any argument.

Return
A DeviceData object.

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

command_inout_reply(*self, id*) → *DeviceData*

Check if the answer of an asynchronous command_inout is arrived (polling model). If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceData object. 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`) Asynchronous call identifier.

Return
(`DeviceData`)

Throws
`AsynCall`, `AsynReplyNotArrived`, `CommunicationFailed`, `DevFailed` from device

command_inout_reply(self, id, timeout) -> `DeviceData`

Check if the answer of an asynchronous command_inout is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a `DeviceData` object. 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`) Asynchronous call identifier.

timeout
(`int`)

Return
(*DeviceData*)

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

command_inout_reply_raw (*self, id, timeout*) → *DeviceData*

Check if the answer of an asynchronous command_inout is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceData object. 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`) Asynchronous call identifier.

timeout
(`int`)

Return
(*DeviceData*)

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

command_list_query (*self*) → sequence<CommandInfo>

Query the device for information on all commands.

Parameters
None

Return
(`CommandInfoList`) Sequence of CommandInfo objects

command_query (*self, command*) → *CommandInfo*

Query the device for information about a single command.

Parameters

command
(`str`) command name

Return
(`CommandInfo`) object

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

Example

```
com_info = dev.command_query(""DevString"")
print(com_info.cmd_name)
print(com_info.cmd_tag)
print(com_info.in_type)
print(com_info.out_type)
print(com_info.in_type_desc)
print(com_info.out_type_desc)
print(com_info.disp_level)
```

See CommandInfo documentation string form more detail

connect (*self*, *corba_name*) → None

Creates a connection to a TANGO device using it's stringified CORBA reference i.e. IOR or corbaloc.

Parameters

corba_name
(`str`) Name of the CORBA object

Return
None

New in PyTango 7.0.0

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), TypeError

description (*self*) → str

Get device description.

Parameters
None

Return
(str) describing the device

dev_name (*self*) → str

Return the device name as it is stored locally

Parameters
None

Return
(str)

event_queue_size (*self, event_id*) → int

Returns the number of stored events in the event reception buffer. After every call to DeviceProxy.get_events(), the event queue size is 0. During event subscription the client must have chosen the 'pull model' for this event. event_id is the event identifier returned by the DeviceProxy.subscribe_event() method.

Parameters

event_id
(int) event identifier

Return
an integer with the queue size

Throws
EventSystemFailed

New in PyTango 7.0.0

get_access_control (*self*) → *AccessControlType*

Returns the current access control type

Parameters
None

Return
(*AccessControlType*) The current access control type

New in PyTango 7.0.0

get_access_right (*self*) → *AccessControlType*

Returns the current access control type

Parameters
None

Return
(`AccessControlType`) The current access control type

New in PyTango 8.0.0

get_asynch_replies (*self, call_timeout*) → None

Try to obtain data returned by a command asynchronously requested. This method blocks for the specified timeout if the reply is not yet arrived. This method fires callback when the reply arrived. If the timeout is set to 0, the call waits undefinitely for the reply

Parameters

call_timeout
(`int`) timeout in miliseconds

Return
None

New in PyTango 7.0.0

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, TypeError

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, TypeError

get_attribute_list(*self*) → sequence<str>

Return the names of all attributes implemented for this device.

Parameters
None

Return
sequence<str>

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

get_attribute_poll_period(*self, attr_name*) → int

Return the attribute polling period.

Parameters

attr_name
(`str`) attribute name

Return
polling period in milliseconds

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, TypeError

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, TypeError

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, TypeError

get_command_list(*self*) → sequence<str>

Return the names of all commands implemented for this device.

Parameters
None

Return
sequence<str>

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

get_command_poll_period(*self*, *cmd_name*) → int

Return the command polling period.

Parameters

cmd_name
(`str`) command name

Return
polling period in milliseconds

get_db_host (*self*) → str

Returns a string with the database host.

Parameters
None

Return
(`str`)

New in PyTango 7.0.0

get_db_port (*self*) → str

Returns a string with the database port.

Parameters
None

Return
(`str`)

New in PyTango 7.0.0

get_db_port_num (*self*) → int

Returns an integer with the database port.

Parameters
None

Return
(`int`)

New in PyTango 7.0.0

get_dev_host (*self*) → str

Returns the current host

Parameters
None

Return
(`str`) the current host

New in PyTango 7.2.0

get_dev_port (*self*) → str

Returns the current port

Parameters
None

Return
(`str`) the current port

New in PyTango 7.2.0

get_device_db (*self*) → *Database*

Returns the internal database reference

Parameters
None

Return
(`Database`) object

New in PyTango 7.0.0

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`, TypeError, ValueError

See Also
subscribe_event

New in PyTango 7.0.0

get_fqdn (*self*) → str

Returns the fully qualified domain name

Parameters
None

Return
(`str`) the fully qualified domain name

New in PyTango 7.2.0

get_from_env_var (*self*) → bool

Returns True if determined by environment variable or False otherwise

Parameters
None

Return
(bool)

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_idl_version(*self*) → int

Get the version of the Tango Device interface implemented by the device

Parameters
None

Return
(int)

get_last_event_date(*self*, *event_id*) → *TimeVal*

Returns the arrival time of the last event stored in the event reception buffer. After every call to DeviceProxy:get_events(), the event reception buffer is empty. In this case an exception will be returned. During event subscription the client must have chosen the 'pull model' for this event. event_id is the event identifier returned by the DeviceProxy.subscribe_event() method.

Parameters

event_id
(int) event identifier

Return
(*tango.TimeVal*) representing the arrival time

Throws
EventSystemFailed

New in PyTango 7.0.0

get_locker(*self*, *lockinfo*) → bool

If the device is locked, this method returns True an set some locker process informations in the structure passed as argument. If the device is not locked, the method returns False.

Parameters

lockinfo [out]
(tango.LockInfo) object that will be filled with lock informantion

Return
(bool) True if the device is locked by us. Otherwise, False

New in PyTango 7.0.0

get_logging_level (*self*) → int

Returns the current device's logging level, where:

- 0=OFF
- 1=FATAL
- 2=ERROR
- 3=WARNING
- 4=INFO
- 5=DEBUG

:Parameters:None :Return: (`int`) representing the current logging level

New in PyTango 7.0.0

get_logging_target (*self*) → sequence<str>

Returns a sequence of string containing the current device's logging targets. Each vector element has the following format: target_type::target_name. An empty sequence is returned is the device has no logging targets.

Parameters
None

Return
a squence<str> with the logging targets

New in PyTango 7.0.0

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, TypeError

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, TypeError

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, TypeError

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, TypeError

New in PyTango 7.0.0

get_source (*self*) → *DevSource*

Get the data source(device, polling buffer, polling buffer then device) used by command_inout or read_attribute methods

Parameters
None

Return
(`DevSource`)

Example

```
source = dev.get_source()
if source == DevSource.CACHE_DEV : ...
```

get_tango_lib_version (*self*) → int

Returns the Tango lib version number used by the remote device Otherwise, throws exception.

Return
(`int`) The device Tango lib version as a 3 or 4 digits number. Possible return value are: 100,200,500,520,700,800,810,. . .

New in PyTango 8.1.0

get_timeout_millis (*self*) → int

Get the client side timeout in milliseconds

Parameters
None

Return
(`int`)

get_transparency_reconnection (*self*) → bool

Returns the device transparency reconnection flag.

Parameters
None

Return
(`bool`) True if transparency reconnection is set or False otherwise

import_info (*self*) → *DbDevImportInfo*

Query the device for import info from the database.

Parameters
None

Return
(*DbDevImportInfo*)

Example

```
dev_import = dev.import_info()
print(dev_import.name)
print(dev_import.exported)
print(dev_ior.ior)
print(dev_version.version)
```

All DbDevImportInfo fields are strings except for exported which is an integer"

info (*self*) → *DeviceInfo*

A method which returns information on the device

Parameters
None

Return
(*DeviceInfo*) object

Example

```
    dev_info = dev.info()
    print(dev_info.dev_class)
    print(dev_info.server_id)
    print(dev_info.server_host)
    print(dev_info.server_version)
    print(dev_info.doc_url)
    print(dev_info.dev_type)

All DeviceInfo fields are strings except for the server_
↪version
which is an integer"
```

is_attribute_polled (*self, attr_name*) → bool

True if the attribute is polled.

Parameters
attr_name (*str*) – attribute name

Returns
boolean value

Return type
bool

is_command_polled (*self, cmd_name*) → bool

True if the command is polled.

Parameters
cmd_name (*str*) – command name

Returns
boolean value

Return type
bool

is_dbase_used (*self*) → bool

Returns if the database is being used

Parameters
None

Return
(bool) True if the database is being used

New in PyTango 7.2.0

is_event_queue_empty (*self, event_id*) → bool

Returns true when the event reception buffer is empty. During event subscription the client must have chosen the 'pull model' for this event. event_id is the event identifier returned by the DeviceProxy.subscribe_event() method.

Parameters

event_id
(int) event identifier

Return
(bool) True if queue is empty or False otherwise

Throws
EventSystemFailed

New in PyTango 7.0.0

is_locked (*self*) → bool

Returns True if the device is locked. Otherwise, returns False.

Parameters
None

Return
(bool) True if the device is locked. Otherwise, False

New in PyTango 7.0.0

is_locked_by_me (*self*) → bool

Returns True if the device is locked by the caller. Otherwise, returns False (device not locked or locked by someone else)

Parameters
None

Return
(bool) True if the device is locked by us. Otherwise, False

New in PyTango 7.0.0

lock (*self, (int)lock_validity*) → None

Lock a device. The lock_validity is the time (in seconds) the lock is kept valid after the previous lock call. A default value of 10 seconds is provided and should be fine in most cases. In case it is necessary to change the lock validity, it's not possible to ask for a validity less than a minimum value set to 2 seconds. The library provided an automatic system to periodically re lock the device until an unlock call. No code is needed to start/stop this automatic re-locking system. The locking system is re-entrant. It is then allowed to call this method on a device already locked by the same process. The locking system has the following features:

- It is impossible to lock the database device or any device server process admin device
- Destroying a locked DeviceProxy unlocks the device
- Restarting a locked device keeps the lock
- It is impossible to restart a device locked by someone else
- Restarting a server breaks the lock

A locked device is protected against the following calls when executed by another client:

- command_inout call except for device state and status requested via command and for the set of commands defined as allowed following the definition of allowed command in the Tango control access schema.
- write_attribute call
- write_read_attribute call
- set_attribute_config call

Parameters

lock_validity
(int) lock validity time in seconds (optional, default value is tango.constants.DEFAULT_LOCK_VALIDITY)

Return
None

New in PyTango 7.0.0

locking_status (*self*) → str

This method returns a plain string describing the device locking status. This string can be:

- 'Device <device name> is not locked' in case the device is not locked
- 'Device <device name> is locked by CPP or Python client with PID <pid> from host <host name>' in case the device is locked by a CPP client
- 'Device <device name> is locked by JAVA client class <main class> from host <host name>' in case the device is locked by a JAVA client

Parameters
None

Return
a string representing the current locking status

New in PyTango 7.0.0"

name (*self*) → str

Return the device name from the device itself.

pending_asynch_call (*self*) → int

Return number of device asynchronous pending requests"

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

poll_attribute (*self, attr_name, period*) → None

Add an attribute to the list of polled attributes.

Parameters

attr_name
(str) attribute name

period
(int) polling period in milliseconds

Return
None

poll_command (*self, cmd_name, period*) → None

Add a command to the list of polled commands.

Parameters

cmd_name
(str) command name

period
(int) polling period in milliseconds

Return
None

polling_status (*self*) → sequence<str>

Return the device polling status.

Parameters
None

Return
(sequence<str>) One string for each polled command/attribute. Each string is multi-line string with:

- attribute/command name
- attribute/command polling period in milliseconds
- attribute/command polling ring buffer
- time needed for last attribute/command execution in milliseconds
- time since data in the ring buffer has not been updated
- delta time between the last records in the ring buffer
- exception parameters in case of the last execution failed

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

Check if the answer of an asynchronous read_attribute is arrived (polling model).

Parameters

id
(`int`) is the asynchronous call identifier.

extract_as
(`ExtractAs`)

Return
If the reply is arrived and if it is a valid reply, it is returned to the caller in a list of DeviceAttribute. 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.

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

New in PyTango 7.0.0

read_attributes_reply *(self, id, timeout, extract_as=ExtractAs.Numpy)* **->** *`DeviceAttribute`*

Check if the answer of an asynchronous read_attributes is arrived (polling model).

Parameters

id
(`int`) is the asynchronous call identifier.

timeout
(`int`)

extract_as
(`ExtractAs`)

Return
If the reply is arrived and if it is a valid reply, it is returned to the caller in a list of DeviceAttribute. 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.

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

New in PyTango 7.0.0

`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

reconnect (*self*, *db_used*) → None

Reconnecto to a CORBA object.

Parameters

db_used
(bool) Use thatabase

Return
None

New in PyTango 7.0.0

remove_logging_target (*self*, *target_type_target_name*) → None

Removes a logging target from the device's target list.

The target_type_target_name input parameter must follow the format: target_type::target_name. Supported target types are: console, file and device. For a device target, the target_name part of the target_type_target_name parameter must contain the name of a log consumer device (as defined in). For a file target, target_name is the full path to the file to remove. If omitted, the default log file is removed. Finally, the target_name part of the target_type_target_name input parameter is ignored in case of a console target and can be omitted. If target_name is set to '*', all targets of the specified target_type are removed.

Parameters

target_type_target_name
(str) logging target

Return
None

New in PyTango 7.0.0

set_access_control(*self, acc*) → None

Sets the current access control type

Parameters

acc
(`AccessControlType`) the type of access control to set

Return
None

New in PyTango 7.0.0

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, TypeError

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, TypeError

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, TypeError

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, TypeError

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_logging_level(*self, (int)level*) → None

Changes the device's logging level, where:

- 0=OFF
- 1=FATAL
- 2=ERROR
- 3=WARNING
- 4=INFO
- 5=DEBUG

Parameters

level
(int) logging level

Return
None

New in PyTango 7.0.0

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, TypeError

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, TypeError

set_source (*self*, *source*) → None

Set the data source(device, polling buffer, polling buffer then device) for command_inout and read_attribute methods.

Parameters

source
(*DevSource*) constant.

Return
None

Example

```
dev.set_source(DevSource.CACHE_DEV)
```

set_timeout_millis (*self*, *timeout*) → None

Set client side timeout for device in milliseconds. Any method which takes longer than this time to execute will throw an exception

Parameters

timeout
integer value of timeout in milliseconds

Return
None

Example

```
dev.set_timeout_millis(1000)
```

set_transparency_reconnection (*self*, *yesno*) → None

Set the device transparency reconnection flag

Parameters
" - val : (bool) True to set transparency reconnection " or False otherwise

Return
None

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

stop_poll_attribute (*self, attr_name*) → None

Remove an attribute from the list of polled attributes.

Parameters

attr_name
(`str`) attribute name

Return
None

stop_poll_command (*self, cmd_name*) → None

Remove a command from the list of polled commands.

Parameters

cmd_name
(`str`) command name

Return
None

subscribe_event (*event_type, cb, stateless=False, green_mode=None*) → 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. This method is currently used device interface change events only.

Parameters

event_type
(*EventType*) Is the event reason and must be on the enumerated values: * EventType.INTERFACE_CHANGE_EVENT

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

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

green_mode
the corresponding green mode (default is GreenMode.Synchronous)

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

Throws
EventSystemFailed, TypeError

subscribe_event(self, attr_name, event, `callback`, filters=[], stateless=False, extract_as=Numpy, green_mode=None) -> 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`)

green_mode
the corresponding green mode (default is GreenMode.Synchronous)

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

Throws
`EventSystemFailed`, TypeError

subscribe_event(self, attr_name, event, queuesize, filters=[], stateless=False, green_mode=None) -> 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.

unlock (*self*, *(bool)force*) → None

Unlock a device. If used, the method argument provides a back door on the locking system. If this argument is set to true, the device will be unlocked even if the caller is not the locker. This feature is provided for administration purpopse and should be used very carefully. If this feature is used, the locker will receive a DeviceUnlocked during the next call which is normally protected by the locking Tango system.

Parameters

force
(`bool`) force unlocking even if we are not the locker (optional, default value is False)

Return
None

New in PyTango 7.0.0

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, KeyError

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

Check if the answer of an asynchronous write_attributes 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_attributes_reply *(self, id, timeout)* **->** `None`

Check if the answer of an asynchronous write_attributes 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_pipe (*self*, *blob*, *green_mode=None*, *wait=True*, *timeout=None*)

Write a *blob* to a single pipe. The *blob* comprises: 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 also a *blob*.

Parameters

blob
a tuple with two elements: blob name (string) and blob data (sequence).

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`, `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 9.2.1

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), TypeError

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(. . .):

event_queue_size(self, event_id) -> int

Returns the number of stored events in the event reception buffer. After every call to DeviceProxy.get_events(), the event queue size is 0. During event subscription the client must have chosen the 'pull model' for this event. event_id is the event identifier returned by the DeviceProxy.subscribe_event() method.

Parameters

event_id
(int) event identifier

Return
an integer with the queue size

Throws
EventSystemFailed

New in PyTango 7.0.0

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, TypeError

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, TypeError, ValueError

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(. . .):

get_last_event_date(self, event_id) -> TimeVal

Returns the arrival time of the last event stored in the event reception buffer. After every call to DeviceProxy:get_events(), the event reception buffer is empty. In this case an exception will be returned. During event subscription the client must have chosen the 'pull model' for this event. event_id is the event identifier returned by the DeviceProxy.subscribe_event() method.

Parameters

event_id
(`int`) event identifier

Return
(*tango.TimeVal*) representing the arrival time

Throws
EventSystemFailed

New in PyTango 7.0.0

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(. . .):

get_attribute_poll_period(self, attr_name) -> int

Return the attribute polling period.

Parameters

attr_name
(`str`) attribute name

Return
polling period in milliseconds

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(…):

get_transparency_reconnection(self) -> bool

Returns the device transparency reconnection flag.

Parameters
None

Return
(bool) True if transparency reconnection is set or False otherwise

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(…):

attribute_history(self, attr_name, depth, extract_as=ExtractAs.Numpy) -> sequence<DeviceAttributeHistory>

Retrieve attribute history from the attribute polling buffer. See chapter on Advanced Feature for all details regarding polling

Parameters

attr_name
(str) Attribute name.

depth
(int) The wanted history depth.

extract_as
(ExtractAs)

Return
This method returns a vector of DeviceAttributeHistory types.

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

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(. . .):

is_event_queue_empty(self, event_id) -> bool

Returns true when the event reception buffer is empty. During event subscription the client must have chosen the 'pull model' for this event. event_id is the event identifier returned by the DeviceProxy.subscribe_event() method.

Parameters

event_id
(int) event identifier

Return
(bool) True if queue is empty or False otherwise

Throws
EventSystemFailed

New in PyTango 7.0.0

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(. . .):

is_attribute_polled(self, attr_name) -> bool

True if the attribute is polled.

param str attr_name
attribute name

returns
boolean value

rtype
bool

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(...):

poll_attribute(self, attr_name, period) -> None

Add an attribute to the list of polled attributes.

Parameters

attr_name
(`str`) attribute name

period
(`int`) polling period in milliseconds

Return
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), TypeError

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, TypeError

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, TypeError

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, TypeError

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, TypeError

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(...):

set_transparency_reconnection(self, yesno) -> None

Set the device transparency reconnection flag

Parameters
" - val : (bool) True to set transparency reconnection " or False otherwise

Return
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(…):

stop_poll_attribute(self, attr_name) -> None

Remove an attribute from the list of polled attributes.

Parameters

attr_name
(str) attribute name

Return
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(event_type, cb, stateless=False, green_mode=None) -> 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. This method is currently used device interface change events only.

Parameters

event_type
(*EventType*) Is the event reason and must be on the enumerated values: * EventType.INTERFACE_CHANGE_EVENT

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

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

green_mode
the corresponding green mode (default is GreenMode.Synchronous)

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

Throws
EventSystemFailed, TypeError

subscribe_event(self, attr_name, event, `callback`, filters=[], stateless=False, extract_as=Numpy, green_mode=None) -> 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`)

green_mode
the corresponding green mode (default is GreenMode.Synchronous)

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

Throws
EventSystemFailed, TypeError

subscribe_event(self, attr_name, event, queuesize, filters=[], stateless=False, green_mode=None) -> 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, KeyError

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*>)

command_inout_asynch (*self, cmd_name, forget=False, forward=True, reserved=- 1*) → int

command_inout_asynch (*self, cmd_name, param, forget=False, forward=True, reserved=- 1*) → int

command_inout_asynch (*self, cmd_name, param_list, forget=False, forward=True, reserved=- 1*) → int

Executes a Tango command on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to Group.command_inout_reply() to obtain the results.

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.

forget
(bool) Fire and forget flag. If set to true, it means that no reply

is expected (i.e. the caller does not care about it and will not even try to get it)

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.

reserved
(int) is reserved for internal purpose and should not be used. This parameter may disappear in a near future.

Return
(int) request id. Pass the returned request id to Group.command_inout_reply() to obtain the results.

Throws

command_inout_reply (*self*, *req_id*, *timeout_ms=0*) → sequence<GroupCmdReply>

Returns the results of an asynchronous command.

Parameters

req_id
(int) Is a request identifier previously returned by one of the command_inout_asynch methods

timeout_ms
(int) For each device in the hierarchy, if the command result is not yet available, command_inout_reply wait timeout_ms milliseconds before throwing an exception. This exception will be part of the global reply. If timeout_ms is set to 0, command_inout_reply waits "indefinitely".

Return
(sequence<*GroupCmdReply*>)

Throws

contains (*self*, *pattern*, *forward=True*) → bool

Parameters

pattern
(str) The pattern can be a fully qualified or simple group name, a device name or a device name pattern.

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

Return
(bool) Returns true if the hierarchy contains groups and/or devices which name matches the specified pattern. Returns false otherwise.

Throws

disable (**args*, ***kwds*)

Disables a group or a device element in a group.

enable (**args*, ***kwds*)

Enables a group or a device element in a group.

get_device_list (*self*, *forward=True*) → sequence<str>

Considering the following hierarchy:

```
g2.add("my/device/04")
g2.add("my/device/05")

g4.add("my/device/08")
g4.add("my/device/09")

g3.add("my/device/06")
g3.add(g4)
g3.add("my/device/07")

g1.add("my/device/01")
g1.add(g2)
g1.add("my/device/03")
g1.add(g3)
g1.add("my/device/02")
```

The returned vector content depends on the value of the forward option. If set to true, the results will be organized as follows:

```
    dl = g1.get_device_list(True)

dl[0] contains "my/device/01" which belongs to g1
dl[1] contains "my/device/04" which belongs to g1.g2
dl[2] contains "my/device/05" which belongs to g1.g2
dl[3] contains "my/device/03" which belongs to g1
dl[4] contains "my/device/06" which belongs to g1.g3
dl[5] contains "my/device/08" which belongs to g1.g3.g4
dl[6] contains "my/device/09" which belongs to g1.g3.g4
dl[7] contains "my/device/07" which belongs to g1.g3
dl[8] contains "my/device/02" which belongs to g1
```

If the forward option is set to false, the results are:

```
    dl = g1.get_device_list(False);

dl[0] contains "my/device/01" which belongs to g1
dl[1] contains "my/device/03" which belongs to g1
dl[2] contains "my/device/02" which belongs to g1
```

Parameters

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

Return
(sequence<`str`>) The list of devices currently in the hierarchy.

Throws

get_fully_qualified_name (**args*, ***kwds*)

Get the complete (dpt-separated) name of the group. This takes into consideration the name of the group and its parents.

get_name (**args*, ***kwds*)
Get the name of the group. Eg: Group('name').get_name() == 'name'

get_size (*self*, *forward=True*) → int

Parameters

forward
(`bool`) If it is set to true (the default), the request is forwarded to sub-groups.

Return
(`int`) The number of the devices in the hierarchy

Throws

is_enabled (**args*, ***kwds*)
Check if a group is enabled. *New in PyTango 7.0.0*

name_equals (**args*, ***kwds*)
New in PyTango 7.0.0

name_matches (**args*, ***kwds*)
New in PyTango 7.0.0

ping (*self*, *forward=True*) → bool

Ping all devices in a group.

Parameters

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

Return
(`bool`) This method returns true if all devices in the group are alive, false otherwise.

Throws

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

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

read_attribute_asynch (*self*, *attr_name*, *forward=True*, *reserved=- 1*) → int

Reads an attribute on each device in the group asynchronously. The method sends the request to all devices and returns immediately.

Parameters

attr_name
(`str`) Name of the attribute to read.

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.

reserved
(`int`) is reserved for internal purpose and should not be used. This parameter may disappear in a near future.

Return
(int) request id. Pass the returned request id to Group.read_attribute_reply() to obtain the results.

Throws

read_attribute_reply (*self*, *req_id*, *timeout_ms=0*) → sequence<GroupAttrReply>

Returns the results of an asynchronous attribute reading.

Parameters

req_id
(int) a request identifier previously returned by read_attribute_asynch.

timeout_ms
(int) For each device in the hierarchy, if the attribute value is not yet available, read_attribute_reply wait timeout_ms milliseconds before throwing an exception. This exception will be part of the global reply. If timeout_ms is set to 0, read_attribute_reply waits "indefinitely".

Return
(sequence<*GroupAttrReply*>)

Throws

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

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

read_attributes_asynch (*self*, *attr_names*, *forward=True*, *reserved=- 1*) → int

Reads the attributes on each device in the group asynchronously. The method sends the request to all devices and returns immediately.

Parameters

attr_names
(sequence<str>) Name of the attributes to read.

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.

reserved
(int) is reserved for internal purpose and should not be used. This parameter may disappear in a near future.

Return
(int) request id. Pass the returned request id to Group.read_attributes_reply() to obtain the results.

Throws

read_attributes_reply (*self*, *req_id*, *timeout_ms=0*) → sequence<GroupAttrReply>

Returns the results of an asynchronous attribute reading.

Parameters

req_id
(`int`) a request identifier previously returned by read_attribute_asynch.

timeout_ms
(`int`) For each device in the hierarchy, if the attribute value is not yet available, read_attribute_reply ait timeout_ms milliseconds before throwing an exception. This exception will be part of the global reply. If timeout_ms is set to 0, read_attributes_reply waits "indefinitely".

Return
(sequence<*GroupAttrReply*>)

Throws

remove_all (*self*) → None
Removes all elements in the _RealGroup. After such a call, the _RealGroup is empty.

set_timeout_millis (*self*, *timeout_ms*) → bool

Set client side timeout for all devices composing the group in milliseconds. Any method which takes longer than this time to execute will throw an exception.

Parameters

timeout_ms
(`int`)

Return
None

Throws
(errors are ignored)

New in PyTango 7.0.0

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(...))

write_attribute_asynch (*self*, *attr_name*, *value*, *forward=True*, *multi=False*) → int

Writes an attribute on each device in the group asynchronously. The method sends the request to all devices and returns immediately.

Parameters

attr_name
(`str`) Name of the attribute to write.

value
(`any`) Value to write. See DeviceProxy.write_attribute

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.

multi
(`bool`) If it is set to false (the default), the same value is applied to all devices in the group. Otherwise the value is interpreted

as a sequence of values, and each value is applied to the corresponding device in the group. In this case len(value) must be equal to group.get_size()!

Return
(int) request id. Pass the returned request id to Group.write_attribute_reply() to obtain the acknowledgements.

Throws

write_attribute_reply (*self*, *req_id*, *timeout_ms=0*) → sequence<GroupReply>

Returns the acknowledgements of an asynchronous attribute writing.

Parameters

req_id
(int) a request identifier previously returned by write_attribute_asynch.

timeout_ms
(int) For each device in the hierarchy, if the acknowledgment is not yet available, write_attribute_reply wait timeout_ms milliseconds before throwing an exception. This exception will be part of the global reply. If timeout_ms is set to 0, write_attribute_reply waits "indefinitely".

Return
(sequence<*GroupReply*>)

Throws

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** (**args*, ***kwargs*)

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** (**args*, ***kwargs*)

Bases:

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** (**args*, ***kwargs*)

Bases:

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.

get_data_raw (*self*) → any

Get the DeviceData containing the output parameter of the command.

Parameters
None

Return
(`DeviceData`) 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*

`tango.futures.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*

`tango.gevent.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**(**args*, ***kwargs*)

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

cleanup() → None

Destroy the ApiUtil singleton instance. After *cleanup()* all references to *DeviceProxy*, *AttributeProxy* or *Database* objects in the current process become invalid and these objects need to be reconstructed.

Parameters
None

Return
None

New in PyTango 9.3.0

get_asynch_cb_sub_model(*self*) → *cb_sub_model*

Get the asynchronous callback sub-model.

Parameters
None

Return
(*cb_sub_model*) the active asynchronous callback sub-model.

New in PyTango 7.1.3

get_asynch_replies(*self*) → None

Fire callback methods for all (any device) asynchronous requests (command and attribute) with already arrived replied. Returns immediately if there is no replies already arrived or if there is no asynchronous requests.

Parameters
None

Return
None

Throws
None, all errors are reported using the err and errors fields of the parameter passed to the `callback` method.

New in PyTango 7.1.3

get_asynch_replies *(self)* **->** `None`

Fire callback methods for all (any device) asynchronous requests (command and attributes) with already arrived replied. Wait and block the caller for timeout milliseconds if they are some device asynchronous requests which are not yet arrived. Returns immediately if there is no asynchronous request. If timeout is set to 0, the call waits until all the asynchronous requests sent has received a reply.

Parameters

timeout
(`int`) timeout (milliseconds)

Return
None

Throws
AsynReplyNotArrived. All other errors are reported using the err and errors fields of the object passed to the `callback` methods.

New in PyTango 7.1.3

instance() → *ApiUtil*

Returns the ApiUtil singleton instance.

Parameters
None

Return
(*ApiUtil*) a reference to the ApiUtil singleton object.

New in PyTango 7.1.3

pending_asynch_call(*self*, *req*) → int

Return number of asynchronous pending requests (any device). The input parameter is an enumeration with three values which are:

- POLLING: Return only polling model asynchronous request number
- CALL_BACK: Return only callback model asynchronous request number
- ALL_ASYNCH: Return all asynchronous request number

Parameters

req
(*asyn_req_type*) asynchronous request type

Return
(`int`) the number of pending requests for the given type

New in PyTango 7.1.3

set_asynch_cb_sub_model(*self*, *model*) → None

Set the asynchronous callback sub-model between the pull and push sub-model. The cb_sub_model data type is an enumeration with two values which are:

- PUSH_CALLBACK: The push sub-model
- PULL_CALLBACK: The pull sub-model

Parameters

model (*cb_sub_model*) the callback sub-model

Return None

New in PyTango 7.1.3

5.2.6 Information classes

See also *Event configuration information*

Attribute

class tango.**AttributeAlarmInfo**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args, **kwargs*)

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**(**args, **kwargs*)

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**(**args, **kwargs*)

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**(**args, **kwargs*)

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**(**args, **kwargs*)

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**(**args, **kwargs*)

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

class ExtractAs

Defines what will go into value field of DeviceAttribute, or what will Attribute.get_write_value() return… Not all the possible values are valid in all the cases.

Valid possible values are:

- Numpy : Value will be stored in [value, w_value]. If the attribute is an scalar, they will contain a value. If it's an SPECTRUM or IMAGE it will be exported as a numpy array.
- Tuple : Value will be stored in [value, w_value]. If the attribute is an scalar, they will contain a value. If it's an SPECTRUM or IMAGE it will be exported as a tuple or tuple of tuples.
- List : Value will be stored in [value, w_value]. If the attribute is an scalar, they will contain a value. If it's an SPECTRUM or IMAGE it will be exported as a list or list of lists

- String : The data will be stored 'as is', the binary data as it comes from TangoC++ in 'value'.
- Nothing : The value will not be extracted from DeviceAttribute

get_date (*self*) → *TimeVal*

Get the time at which the attribute was read by the server.

Note: It's the same as reading the "time" attribute.

Parameters
None

Return
(`TimeVal`) The attribute read timestamp.

get_err_stack (*self*) → sequence<DevError>

Returns the error stack reported by the server when the attribute was read.

Parameters
None

Return
(sequence<`DevError`>)

set_w_dim_x (*self, val*) → None

Sets the write value dim x.

Parameters

val
(`int`) new write dim x

Return
None

New in PyTango 8.0.0

set_w_dim_y (*self, val*) → None

Sets the write value dim y.

Parameters

val
(`int`) new write dim y

Return
None

New in PyTango 8.0.0

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**(**args*, ***kwargs*)

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.

extract(*self*) → any

Get the actual value stored in the DeviceData.

Parameters
None

Return
Whatever is stored there, or None.

get_type(*self*) → *CmdArgType*

This method returns the Tango data type of the data inside the DeviceData object.

Parameters
None

Return
The content arg type.

insert(*self*, *data_type*, *value*) → None

Inserts a value in the DeviceData.

Parameters

data_type
value
(`any`) The value to insert

Return
Whatever is stored there, or None.

is_empty(*self*) → bool

It can be used to test whether the DeviceData object has been initialized or not.

Parameters
None

Return
True or False depending on whether the DeviceData object contains data or not.

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(**args, **kwargs*)

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(**args, **kwargs*)

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(**args, **kwargs*)

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(**args, **kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args, **kwargs*)

Bases:

See `DeviceAttribute`.

class tango.**DeviceDataHistory**(**args, **kwargs*)

Bases:

See `DeviceData`.

5.2.11 Enumerations & other classes

Enumerations

class tango.**LockerLanguage**(**args, **kwargs*)

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**(**args, **kwargs*)

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**(**args*, ***kwargs*)

An enumeration representing the MessBoxType

- STOP
- INFO

New in PyTango 7.0.0

class tango.**PollObjType**(**args*, ***kwargs*)

An enumeration representing the PollObjType

- POLL_CMD
- POLL_ATTR
- EVENT_HEARTBEAT
- STORE_SUBDEV

New in PyTango 7.0.0

class tango.**PollCmdCode**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

An enumeration representing the type of serialization performed by the device server

- BY_DEVICE
- BY_CLASS
- BY_PROCESS
- NO_SYNC

class tango.**AttReqType**(**args*, ***kwargs*)

An enumeration representing the type of attribute request

- READ_REQ
- WRITE_REQ

class tango.**LockCmdCode**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

An enumeration representing the LogTarget

- LOG_CONSOLE
- LOG_FILE
- LOG_DEVICE

New in PyTango 7.0.0

class tango.**EventType**(**args*, ***kwargs*)

An enumeration representing event type

- CHANGE_EVENT
- QUALITY_EVENT
- PERIODIC_EVENT
- ARCHIVE_EVENT
- USER_EVENT
- ATTR_CONF_EVENT
- DATA_READY_EVENT
- INTERFACE_CHANGE_EVENT
- PIPE_EVENT

DATA_READY_EVENT - New in PyTango 7.0.0 INTERFACE_CHANGE_EVENT - New in PyTango 9.2.2 PIPE_EVENT - New in PyTango 9.2.2

class tango.**KeepAliveCmdCode**(**args*, ***kwargs*)

An enumeration representing the KeepAliveCmdCode

- EXIT_TH

New in PyTango 7.0.0

class tango.**AccessControlType**(**args*, ***kwargs*)

An enumeration representing the AccessControlType

- ACCESS_READ
- ACCESS_WRITE

New in PyTango 7.0.0

class tango.**asyn_req_type**(**args*, ***kwargs*)

An enumeration representing the asynchronous request type

- POLLING
- CALLBACK
- ALL_ASYNCH

class tango.**cb_sub_model**(**args*, ***kwargs*)

An enumeration representing callback sub model

- PUSH_CALLBACK
- PULL_CALLBACK

class tango.**AttrQuality**(**args*, ***kwargs*)

An enumeration representing the attribute quality

- ATTR_VALID
- ATTR_INVALID
- ATTR_ALARM
- ATTR_CHANGING
- ATTR_WARNING

class tango.**AttrWriteType**(**args*, ***kwargs*)

An enumeration representing the attribute type

- READ
- READ_WITH_WRITE

- WRITE
- READ_WRITE

class tango.**AttrDataFormat**(**args*, ***kwargs*)

An enumeration representing the attribute format

- SCALAR
- SPECTRUM
- IMAGE
- FMT_UNKNOWN

class tango.**PipeWriteType**(**args*, ***kwargs*)

An enumeration representing the pipe type

- PIPE_READ
- PIPE_READ_WRITE

class tango.**DevSource**(**args*, ***kwargs*)

An enumeration representing the device source for data

- DEV
- CACHE
- CACHE_DEV

class tango.**ErrSeverity**(**args*, ***kwargs*)

An enumeration representing the error severity

- WARN
- ERR
- PANIC

class tango.**DevState**(**args*, ***kwargs*)

An enumeration representing the device state

- ON
- OFF
- CLOSE
- OPEN
- INSERT
- EXTRACT
- MOVING
- STANDBY
- FAULT
- INIT
- RUNNING
- ALARM
- DISABLE
- UNKNOWN

class tango.**DispLevel**(**args*, ***kwargs*)

An enumeration representing the display level

- OPERATOR
- EXPERT

class tango.**GreenMode**(**args*, ***kwargs*)

An enumeration representing the GreenMode

- Synchronous
- Futures
- Gevent

New in PyTango 8.1.0

Other classes

class tango.**Release**

Summarize release information as class attributes.

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>
- release: (str) pre-release, post-release or development release; it is empty for final releases.
- 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**(**args, **kwargs*)

Time value structure with the following members:

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

static fromdatetime(*dt*) → *TimeVal*

A static method returning a *tango.TimeVal* object representing the given datetime.datetime

Parameters

dt
(datetime.datetime) a datetime object

Return
(*TimeVal*) representing the given timestamp

New in version 7.1.0.

New in version 7.1.2: Documented

static fromtimestamp(*ts*) → *TimeVal*

A static method returning a *tango.TimeVal* object representing the given timestamp

Parameters

ts
(float) a timestamp

Return
(`TimeVal`) representing the given timestamp

New in version 7.1.0.

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()`)

static now () → *TimeVal*

A static method returning a `tango.TimeVal` object representing the current time

Parameters
None

Return
(`TimeVal`) representing the current time

New in version 7.1.0.

New in version 7.1.2: Documented

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
from tango.server import attribute, command


class Clock(Device):

    time = attribute()

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

    @command(dtype_in=str, dtype_out=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

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

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

class PowerSupply(Device):

    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__":
    PowerSupply.run_server()
```

Pretty cool, uh?

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. 'int64'
4. `tango.CmdArgType.DevLong64`
5. 'DevLong64'
6. `numpy.int64`

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
'DevShort'	DevShort
numpy.int16	DevShort
'uint16'	DevUShort
DevUShort	DevUShort

continues on next page

Table 2 – continued from previous page

dtype argument	converts to tango type
'DevUShort'	DevUShort
numpy.uint16	DevUShort
'int32'	DevLong
DevLong	DevLong
'DevLong'	DevLong
numpy.int32	DevLong
'uint32'	DevULong
DevULong	DevULong
'DevULong'	DevULong
numpy.uint32	DevULong
int	DevLong64
'int'	DevLong64
'int64'	DevLong64
DevLong64	DevLong64
'DevLong64'	DevLong64
numpy.int64	DevLong64
'uint'	DevULong64
'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
DevVarShortArray	DevVarShortArray
'DevVarShortArray'	DevVarShortArray
DevVarLongArray	DevVarLongArray
'DevVarLongArray'	DevVarLongArray

continues on next page

Table 2 – continued from previous page

dtype argument	converts to tango type
`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`

class `tango.server.`**`Device`**(*cl*, *name*)

Bases: `BaseDevice`

Device class for the high-level API.

All device specific classes should inherit from this class.

`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`** (`server.attribute` *or* `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 (if attr is of type server.attribute, then use the fget field in the attr object instead)
- **`w_meth`** (*`callable`*) – the write method to be called on a write request (if attr is writable) (if attr is of type server.attribute, then use the fset field in the attr object instead)
- **`is_allo_meth`** (*`callable`*) – the method that is called to check if it is possible to access the attribute or not (if attr is of type server.attribute, then use the fisallowed field in the attr object instead)

Returns

the newly created attribute.

Return type

Attr

Raises

`DevFailed` –

add_command (*self, cmd, device_level=True*) → cmd

Add a new command to the device command list.

Parameters

- **cmd** – the new command to be added to the list
- **device_level** – Set this flag to true if the command must be added for only this device

Returns
The command to add

Return type
Command

Raises
DevFailed –

always_executed_hook ()

Tango always_executed_hook. Default implementation does nothing

append_status (*self, status, new_line=False*)

Appends a string to the device status.

Parameters

- **status** (*str*) – the string to be appened to the device status
- **new_line** (*bool*) – If true, appends a new line character before the string. Default is False

check_command_exists (*self*)

Check that a command is supported by the device and does not need input value.

The method throws an exception if the command is not defined or needs an input value.

Parameters
cmd_name (*str*) – the command name

Raises

- **DevFailed** –
- **API_IncompatibleCmdArgumentType** –
- **API_CommandNotFound** –

New in PyTango 7.1.2

debug_stream (*self, msg, *args*)

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

delete_device (*self*)

Delete the device.

dev_state (*self*) → *DevState*

Get device state.

Default method to get device state. The behaviour of this method depends on the device state. If the device state is ON or ALARM, it reads the attribute(s) with an alarm level defined, check if the read value is above/below the alarm and eventually change the state to ALARM, return the device state. For all th other device state, this method simply returns the state This method can be redefined in sub-classes in case of the default behaviour does not fullfill the needs.

Returns
the device state

Return type
DevState

Raises
DevFailed – If it is necessary to read attribute(s) and a problem occurs during the reading

dev_status (*self*) → str

Get device status.

Default method to get device status. It returns the contents of the device dev_status field. If the device state is ALARM, alarm messages are added to the device status. This method can be redefined in sub-classes in case of the default behaviour does not fullfill the needs.

Returns
the device status

Return type
str

Raises
DevFailed – If it is necessary to read attribute(s) and a problem occurs during the reading

error_stream (*self, msg, *args*)

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

fatal_stream (*self, msg, *args*)

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

get_attr_min_poll_period (*self*) → Sequence[str]

Returns the min attribute poll period

Returns
the min attribute poll period

Return type
Sequence[str]

New in PyTango 7.2.0

get_attr_poll_ring_depth(*self, attr_name*) → int
Returns the attribute poll ring depth.

Parameters
attr_name (*str*) – the attribute name

Returns
the attribute poll ring depth

Return type
int

New in PyTango 7.1.2

get_attribute_config(*self, attr_names*) → list[*DeviceAttributeConfig*]
Returns the list of AttributeConfig for the requested names

Parameters
attr_names (*list[str]*) – sequence of str with attribute names

Returns
`tango.DeviceAttributeConfig` for each requested attribute name

Return type
list[`tango.DeviceAttributeConfig`]

get_attribute_config_2(*self, attr_names*) → list[AttributeConfig_2]
Returns the list of AttributeConfig_2 for the requested names

Parameters
attr_names (*list[str]*) – sequence of str with attribute names

Returns
list of `tango.AttributeConfig_2` for each requested attribute name

Return type
list[`tango.AttributeConfig_2`]

get_attribute_config_3(*self, attr_name*) → list[AttributeConfig_3]
Returns the list of AttributeConfig_3 for the requested names

Parameters
attr_names (*list[str]*) – sequence of str with attribute names

Returns
list of `tango.AttributeConfig_3` for each requested attribute name

Return type
list[`tango.AttributeConfig_3`]

get_attribute_poll_period(*self, attr_name*) → int
Returns the attribute polling period (ms) or 0 if the attribute is not polled.

Parameters
attr_name (*str*) – attribute name

Returns
attribute polling period (ms) or 0 if it is not polled

Return type
int

New in PyTango 8.0.0

get_cmd_min_poll_period(*self*) → Sequence[str]

Returns the min command poll period.

Returns
the min command poll period

Return type
Sequence[str]

New in PyTango 7.2.0

get_cmd_poll_ring_depth(*self, cmd_name*) → int

Returns the command poll ring depth.

Parameters
cmd_name (*str*) – the command name

Returns
the command poll ring depth

Return type
int

New in PyTango 7.1.2

get_command_poll_period(*self, cmd_name*) → int

Returns the command polling period (ms) or 0 if the command is not polled.

Parameters
cmd_name (*str*) – command name

Returns
command polling period (ms) or 0 if it is not polled

Return type
int

New in PyTango 8.0.0

get_dev_idl_version(*self*) → int

Returns the IDL version.

Returns
the IDL version

Return type
int

New in PyTango 7.1.2

get_device_attr(*self*) → *MultiAttribute*

Get device multi attribute object.

Returns
the device's MultiAttribute object

Return type
MultiAttribute

get_device_class(*self*)

Get device class singleton.

Returns
the device class singleton (device_class field)

Return type
DeviceClass

get_device_properties (*self, ds_class=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

Raises
DevFailed –

get_exported_flag (*self*) → bool

Returns the state of the exported flag

Returns
the state of the exported flag

Return type
bool

New in PyTango 7.1.2

get_logger (*self*) → Logger

Returns the Logger object for this device

Returns
the Logger object for this device

Return type
Logger

get_min_poll_period (*self*) → int

Returns the min poll period.

Returns
the min poll period

Return type
int

New in PyTango 7.2.0

get_name (*self*)

Get a COPY of the device name.

Returns
the device name

Return type
str

get_non_auto_polled_attr (*self*) → Sequence[str]

Returns a COPY of the list of non automatic polled attributes

Returns
a COPY of the list of non automatic polled attributes

Return type
Sequence[str]

New in PyTango 7.1.2

get_non_auto_polled_cmd (*self*) → Sequence[str]

Returns a COPY of the list of non automatic polled commands

Returns
a COPY of the list of non automatic polled commands

Return type
Sequence[str]

New in PyTango 7.1.2

get_poll_old_factor (*self*) → int
Returns the poll old factor

Returns
the poll old factor

Return type
int

New in PyTango 7.1.2

get_poll_ring_depth (*self*) → int
Returns the poll ring depth

Returns
the poll ring depth

Return type
int

New in PyTango 7.1.2

get_polled_attr (*self*) → Sequence[str]
Returns a COPY of the list of polled attributes

Returns
a COPY of the list of polled attributes

Return type
Sequence[str]

New in PyTango 7.1.2

get_polled_cmd (*self*) → Sequence[str]
Returns a COPY of the list of polled commands

Returns
a COPY of the list of polled commands

Return type
Sequence[str]

New in PyTango 7.1.2

get_prev_state (*self*) → *DevState*
Get a COPY of the device's previous state.

Returns
the device's previous state

Return type
DevState

get_state (*self*) → *DevState*
Get a COPY of the device state.

Returns
Current device state

Return type
DevState

get_status(*self*) → str

Get a COPY of the device status.

Returns
the device status

Return type
str

info_stream(*self, msg, *args*)

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

init_device()

Tango init_device method. Default implementation calls *get_device_properties()*

init_logger(*self*) → None

Setups logger for the device. Called automatically when device starts.

initialize_dynamic_attributes()

Method executed at initializion phase to create dynamic attributes. Default implementation does nothing. Overwrite when necessary.

is_attribute_polled(*self, attr_name*) → bool

True if the attribute is polled.

Parameters
attr_name (*str*) – attribute name

Returns
True if the attribute is polled

Return type
bool

is_command_polled(*self, cmd_name*) → bool

True if the command is polled.

Parameters
cmd_name (*str*) – attribute name

Returns
True if the command is polled

Return type
bool

is_device_locked(*self*) → bool

Returns if this device is locked by a client.

Returns
True if it is locked or False otherwise

Return type
bool

New in PyTango 7.1.2

is_polled(*self*) → bool

Returns if it is polled

Returns
True if it is polled or False otherwise

Return type
bool

New in PyTango 7.1.2

is_there_subscriber(*self, att_name, event_type*) → bool

Check if there is subscriber(s) listening for the event.

This method returns a boolean set to true if there are some subscriber(s) listening on the event specified by the two method arguments. Be aware that there is some delay (up to 600 sec) between this method returning false and the last subscriber unsubscription or crash...

The device interface change event is not supported by this method.

Parameters

- **att_name** (`str`) – the attribute name
- **event_type** (`EventType`) – the event type

Returns
True if there is at least one listener or False otherwise

Return type
bool

poll_attribute(*self, attr_name, period*) → None

Add an attribute to the list of polled attributes.

Parameters

- **attr_name** (`str`) – attribute name
- **period** (`int`) – polling period in milliseconds

Returns
None

Return type
None

poll_command(*self, cmd_name, period*) → None

Add a command to the list of polled commands.

Parameters

- **cmd_name** (`str`) – attribute name
- **period** (`int`) – polling period in milliseconds

Returns
None

Return type
None

push_archive_event(*self, attr_name*)

push_archive_event(*self, attr_name, except*)

push_archive_event(*self, attr_name, data, dim_x=1, dim_y=0*)

push_archive_event(*self, attr_name, str_data, data*)

push_archive_event(*self, attr_name, data, time_stamp, quality, dim_x=1, dim_y=0*)

push_archive_event (*self, attr_name, str_data, data, time_stamp, quality*)

Push an archive event for the given attribute name.

The event is pushed to the notification daemon.

Parameters

- **attr_name** (*str*) – attribute name
- **data** – the data to be sent as attribute event data. Data must be compatible with the attribute type and format. for SPECTRUM and IMAGE attributes, data can be any type of sequence of elements compatible with the attribute type
- **str_data** (*str*) – special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.
- **except** (DevFailed) – Instead of data, you may want to send an exception.
- **dim_x** (*int*) – the attribute x length. Default value is 1
- **dim_y** (*int*) – the attribute y length. Default value is 0
- **time_stamp** (*double*) – the time stamp
- **quality** (AttrQuality) – the attribute quality factor

Raises
DevFailed – If the attribute data type is not coherent.

push_att_conf_event (*self, attr*)

Push an attribute configuration event.

Parameters
attr (Attribute) – the attribute for which the configuration event will be sent.

New in PyTango 7.2.1

push_change_event (*self, attr_name*)

push_change_event (*self, attr_name, except*)

push_change_event (*self, attr_name, data, dim_x=1, dim_y=0*)

push_change_event (*self, attr_name, str_data, data*)

push_change_event (*self, attr_name, data, time_stamp, quality, dim_x=1, dim_y=0*)

push_change_event (*self, attr_name, str_data, data, time_stamp, quality*)

Push a change event for the given attribute name.

The event is pushed to the notification daemon.

Parameters

- **attr_name** (*str*) – attribute name
- **data** – the data to be sent as attribute event data. Data must be compatible with the attribute type and format. for SPECTRUM and IMAGE attributes, data can be any type of sequence of elements compatible with the attribute type
- **str_data** (*str*) – special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.
- **except** (DevFailed) – Instead of data, you may want to send an exception.
- **dim_x** (*int*) – the attribute x length. Default value is 1

- **dim_y** (*int*) – the attribute y length. Default value is 0
- **time_stamp** (*double*) – the time stamp
- **quality** (*AttrQuality*) – the attribute quality factor

Raises
DevFailed – If the attribute data type is not coherent.

push_data_ready_event (*self, attr_name, counter=0*)
Push a data ready event for the given attribute name.

The event is pushed to the notification daemon.

The method needs only the attribue name and an optional "counter" which will be passed unchanged within the event

Parameters

- **attr_name** (*str*) – attribute name
- **counter** (*int*) – the user counter

Raises
DevFailed – If the attribute name is unknown.

push_event (*self, attr_name, filt_names, filt_vals*)

push_event (*self, attr_name, filt_names, filt_vals, data, dim_x=1, dim_y=0*)
push_event (*self, attr_name, filt_names, filt_vals, str_data, data*)
push_event (*self, attr_name, filt_names, filt_vals, data, time_stamp, quality, dim_x=1, dim_y=0*)
push_event (*self, attr_name, filt_names, filt_vals, str_data, data, time_stamp, quality*)

Push a user event for the given attribute name.

The event is pushed to the notification daemon.

Parameters

- **attr_name** (*str*) – attribute name
- **filt_names** (*Sequence[str]*) – the filterable fields name
- **filt_vals** (*Sequence[double]*) – the filterable fields value
- **data** – the data to be sent as attribute event data. Data must be compatible with the attribute type and format. for SPECTRUM and IMAGE attributes, data can be any type of sequence of elements compatible with the attribute type
- **str_data** (*str*) – special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.
- **dim_x** (*int*) – the attribute x length. Default value is 1
- **dim_y** (*int*) – the attribute y length. Default value is 0
- **time_stamp** (*double*) – the time stamp
- **quality** (*AttrQuality*) – the attribute quality factor

Raises
DevFailed – If the attribute data type is not coherent.

push_pipe_event (*self, blob*)
Push an pipe event.

Parameters
blob – the blob which pipe event will be send.

New in PyTango 9.2.2

read_attr_hardware (*self, attr_list*)

Read the hardware to return attribute value(s).

Default method to implement an action necessary on a device to read the hardware involved in a read attribute CORBA call. This method must be redefined in sub-classes in order to support attribute reading

Parameters

attr_list (*Sequence[int]*) – list of indices in the device object attribute vector of an attribute to be read.

Raises

DevFailed – This method does not throw exception but a redefined method can.

register_signal (*self, signo*)

Register a signal.

Register this device as device to be informed when signal signo is sent to to the device server process

Parameters

signo (*int*) – signal identifier

remove_attribute (*self, attr_name*)

Remove one attribute from the device attribute list.

Parameters

attr_name (*str*) – attribute name

Raises

DevFailed –

remove_command (*self, attr_name*)

Remove one command from the device command list.

Parameters

- **cmd_name** (*str*) – command name to be removed from the list
- **free_it** (*bool*) – set to true if the command object must be freed.
- **clean_db** – Clean command related information (included polling info if the command is polled) from database.

Raises

DevFailed –

classmethod run_server (*args=None, **kwargs*)

Run the class as a device server. It is based on the tango.server.run method.

The difference is that the device class and server name are automatically given.

Args:

args (iterable): args as given in the tango.server.run method
without the server name. If None, the sys.argv list is used

kwargs: the other keywords argument are as given
in the tango.server.run method.

set_archive_event (*self, attr_name, implemented, detect=True*)

Set an implemented flag for the attribute to indicate that the server fires archive events manually, without the polling to be started.

If the detect parameter is set to true, the criteria specified for the archive event are verified and the event is only pushed if they are fullfilled. If detect is set to false the event is fired without any value checking!

Parameters

- **attr_name** (*str*) – attribute name
- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – Triggers the verification of the change event properties when set to true. Default value is true.

set_attribute_config_3 (*self*, *new_conf*) → None

Sets attribute configuration locally and in the Tango database

Parameters

new_conf (list[`tango.AttributeConfig_3`]) – The new attribute(s) configuration. One AttributeConfig structure is needed for each attribute to update

Returns

None

Return type

None

set_change_event (*self*, *attr_name*, *implemented*, *detect=True*)

Set an implemented flag for the attribute to indicate that the server fires change events manually, without the polling to be started.

If the detect parameter is set to true, the criteria specified for the change event are verified and the event is only pushed if they are fullfilled. If detect is set to false the event is fired without any value checking!

Parameters

- **attr_name** (*str*) – attribute name
- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – Triggers the verification of the change event properties when set to true. Default value is true.

set_data_ready_event (*self*, *attr_name*, *implemented*)

Set an implemented flag for the attribute to indicate that the server fires data ready events manually.

Parameters

- **attr_name** (*str*) – attribute name
- **implemented** (*bool*) – True when the server fires change events manually.

set_state (*self*, *new_state*)

Set device state.

Parameters

new_state (`DevState`) – the new device state

set_status (*self*, *new_status*)

Set device status.

Parameters

new_status (*str*) – the new device status

signal_handler (*self, signo*)

Signal handler.

The method executed when the signal arrived in the device server process. This method is defined as virtual and then, can be redefined following device needs.

Parameters
signo (*int*) – the signal number

Raises
DevFailed – This method does not throw exception but a redefined method can.

start_logging (*self*) → None

Starts logging

stop_logging (*self*) → None

Stops logging

stop_poll_attribute (*self, attr_name*) → None

Remove an attribute from the list of polled attributes.

Parameters
attr_name (*str*) – attribute name

Returns
None

Return type
None

stop_poll_command (*self, cmd_name*) → None

Remove a command from the list of polled commands.

Parameters
cmd_name (*str*) – cmd_name name

Returns
None

Return type
None

stop_polling (*self*)

stop_polling (*self, with_db_upd*)

Stop all polling for a device. if the device is polled, call this method before deleting it.

Parameters
with_db_upd (*bool*) – Is it necessary to update db?

New in PyTango 7.1.2

unregister_signal (*self, signo*)

Unregister a signal.

Unregister this device as device to be informed when signal signo is sent to to the device server process

Parameters
signo (*int*) – signal identifier

warn_stream (*self, msg, *args*)

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

write_attr_hardware (*self*)
Write the hardware for attributes.

Default method to implement an action necessary on a device to write the hardware involved in a write attribute. This method must be redefined in sub-classes in order to support writable attribute

Parameters
attr_list (*Sequence[int]*) – list of indices in the device object attribute vector of an attribute to be written.

Raises
DevFailed – This method does not throw exception but a redefined method can.

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):

    voltage = attribute()

    def read_voltage(self):
        return 999.999
```

The same can be achieved with:

```
class PowerSupply(Device):

    @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*)
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
fisallowed	str or callable	'is_<attr_name>_allowed'	is allowed method name or method object
label	str	'<attr_name>'	attribute label

Table 3 – continued from previous page

parameter	type	default value	description
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
forwarded	bool	False	the attribute should be forwarded if True

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):

    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):

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

(continues on next page)

(continued from previous page)

```
    @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):

    @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):

    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):

    @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
fisallowed	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):

    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):

    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='', mandatory=False, 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):

    host = device_property(dtype=str)
    port = device_property(dtype=int, mandatory=True)
```

Parameters

- **dtype** – Data type (see *Data types*)
- **doc** – property documentation (optional)
- **(optional** (*mandatory*) – default is False)
- **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):

    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, raises=False*)

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, run

class PowerSupply(Device):
    pass

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, run

class PowerSupply(Device):
    pass

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
- **raises** (*bool*) – Disable error handling and propagate exceptions from the server

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]

Changed in version 9.2.2: *raises* argument has been added

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.**LatestDeviceImpl**

Latest implementation of the TANGO device base class (alias for Device_5Impl).

It 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** (server.attribute *or* 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 (if attr is of type server.attribute, then use the fget field in the attr object instead)
- **w_meth** (*callable*) – the write method to be called on a write request (if attr is writable) (if attr is of type server.attribute, then use the fset field in the attr object instead)
- **is_allo_meth** (*callable*) – the method that is called to check if it is possible to access the attribute or not (if attr is of type server.attribute, then use the fisallowed field in the attr object instead)

Returns
the newly created attribute.

Return type
Attr

Raises
DevFailed –

add_command (*self, cmd, device_level=True*) → cmd

Add a new command to the device command list.

Parameters

- **cmd** – the new command to be added to the list
- **device_level** – Set this flag to true if the command must be added for only this device

Returns
The command to add

Return type
Command

Raises
DevFailed –

always_executed_hook (*self*)

Hook method.

Default method to implement an action necessary on a device before any command is executed. This method can be redefined in sub-classes in case of the default behaviour does not fullfill the needs

Raises
DevFailed – This method does not throw exception but a redefined method can.

append_status (*self, status, new_line=False*)

Appends a string to the device status.

Parameters

- **status** (*str*) – the string to be appened to the device status
- **new_line** (*bool*) – If true, appends a new line character before the string. Default is False

check_command_exists (*self*)

Check that a command is supported by the device and does not need input value.

The method throws an exception if the command is not defined or needs an input value.

Parameters
cmd_name (*str*) – the command name

Raises

- ***DevFailed*** –
- **API_IncompatibleCmdArgumentType** –
- **API_CommandNotFound** –

New in PyTango 7.1.2

debug_stream (*self, msg, *args*)

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

delete_device (*self*)

Delete the device.

dev_state (*self*) → *DevState*

Get device state.

Default method to get device state. The behaviour of this method depends on the device state. If the device state is ON or ALARM, it reads the attribute(s) with an alarm level defined, check if the read value is above/below the alarm and eventually change the state to ALARM, return the device state. For all th other device state, this method simply returns the state This method can be redefined in sub-classes in case of the default behaviour does not fullfill the needs.

Returns
the device state

Return type
DevState

Raises
DevFailed – If it is necessary to read attribute(s) and a problem occurs during the reading

dev_status (*self*) → str

Get device status.

Default method to get device status. It returns the contents of the device dev_status field. If the device state is ALARM, alarm messages are added to the device status. This method can be redefined in sub-classes in case of the default behaviour does not fullfill the needs.

Returns
the device status

Return type
str

Raises
: ***DevFailed*** – If it is necessary to read attribute(s) and a problem occurs during the reading

error_stream (*self, msg, *args*)
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

fatal_stream (*self, msg, *args*)
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

get_attr_min_poll_period (*self*) → Sequence[str]
Returns the min attribute poll period

Returns
: the min attribute poll period

Return type
: Sequence[str]

New in PyTango 7.2.0

get_attr_poll_ring_depth (*self, attr_name*) → int
Returns the attribute poll ring depth.

Parameters
: **attr_name** (*str*) – the attribute name

Returns
: the attribute poll ring depth

Return type
: int

New in PyTango 7.1.2

get_attribute_config (*self, attr_names*) → list[*DeviceAttributeConfig*]
Returns the list of AttributeConfig for the requested names

Parameters
: **attr_names** (*list[str]*) – sequence of str with attribute names

Returns
: *tango.DeviceAttributeConfig* for each requested attribute name

Return type
: list[*tango.DeviceAttributeConfig*]

get_attribute_config_2 (*self, attr_names*) → list[AttributeConfig_2]

Returns the list of AttributeConfig_2 for the requested names

Parameters
attr_names (*list[str]*) – sequence of str with attribute names

Returns
list of `tango.AttributeConfig_2` for each requested attribute name

Return type
list[`tango.AttributeConfig_2`]

get_attribute_config_3 (*self, attr_name*) → list[AttributeConfig_3]

Returns the list of AttributeConfig_3 for the requested names

Parameters
attr_names (*list[str]*) – sequence of str with attribute names

Returns
list of `tango.AttributeConfig_3` for each requested attribute name

Return type
list[`tango.AttributeConfig_3`]

get_attribute_poll_period (*self, attr_name*) → int

Returns the attribute polling period (ms) or 0 if the attribute is not polled.

Parameters
attr_name (*str*) – attribute name

Returns
attribute polling period (ms) or 0 if it is not polled

Return type
int

New in PyTango 8.0.0

get_cmd_min_poll_period (*self*) → Sequence[str]

Returns the min command poll period.

Returns
the min command poll period

Return type
Sequence[str]

New in PyTango 7.2.0

get_cmd_poll_ring_depth (*self, cmd_name*) → int

Returns the command poll ring depth.

Parameters
cmd_name (*str*) – the command name

Returns
the command poll ring depth

Return type
int

New in PyTango 7.1.2

get_command_poll_period (*self, cmd_name*) → int

Returns the command polling period (ms) or 0 if the command is not polled.

Parameters
cmd_name (*str*) – command name

Returns
command polling period (ms) or 0 if it is not polled

Return type
int

New in PyTango 8.0.0

get_dev_idl_version(*self*) → int

Returns the IDL version.

Returns
the IDL version

Return type
int

New in PyTango 7.1.2

get_device_attr(*self*) → *MultiAttribute*

Get device multi attribute object.

Returns
the device's MultiAttribute object

Return type
MultiAttribute

get_device_class(*self*)

Get device class singleton.

Returns
the device class singleton (device_class field)

Return type
DeviceClass

get_device_properties(*self, ds_class=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

Raises
DevFailed –

get_exported_flag(*self*) → bool

Returns the state of the exported flag

Returns
the state of the exported flag

Return type
bool

New in PyTango 7.1.2

get_logger(*self*) → Logger

Returns the Logger object for this device

Returns
the Logger object for this device

Return type
Logger

get_min_poll_period(*self*) → int
Returns the min poll period.

Returns
the min poll period

Return type
int

New in PyTango 7.2.0

get_name(*self*)
Get a COPY of the device name.

Returns
the device name

Return type
str

get_non_auto_polled_attr(*self*) → Sequence[str]
Returns a COPY of the list of non automatic polled attributes

Returns
a COPY of the list of non automatic polled attributes

Return type
Sequence[str]

New in PyTango 7.1.2

get_non_auto_polled_cmd(*self*) → Sequence[str]
Returns a COPY of the list of non automatic polled commands

Returns
a COPY of the list of non automatic polled commands

Return type
Sequence[str]

New in PyTango 7.1.2

get_poll_old_factor(*self*) → int
Returns the poll old factor

Returns
the poll old factor

Return type
int

New in PyTango 7.1.2

get_poll_ring_depth(*self*) → int
Returns the poll ring depth

Returns
the poll ring depth

Return type
int

New in PyTango 7.1.2

get_polled_attr(*self*) → Sequence[str]
Returns a COPY of the list of polled attributes

Returns
a COPY of the list of polled attributes

Return type
Sequence[str]

New in PyTango 7.1.2

get_polled_cmd (*self*) → Sequence[str]
Returns a COPY of the list of polled commands

Returns
a COPY of the list of polled commands

Return type
Sequence[str]

New in PyTango 7.1.2

get_prev_state (*self*) → *DevState*
Get a COPY of the device's previous state.

Returns
the device's previous state

Return type
DevState

get_state (*self*) → *DevState*
Get a COPY of the device state.

Returns
Current device state

Return type
DevState

get_status (*self*) → str
Get a COPY of the device status.

Returns
the device status

Return type
str

info_stream (*self, msg, *args*)
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

init_device (*self*)
Intialize the device.

init_logger (*self*) → None
Setups logger for the device. Called automatically when device starts.

is_attribute_polled (*self, attr_name*) → bool
True if the attribute is polled.

Parameters
attr_name (`str`) – attribute name

Returns
True if the attribute is polled

Return type
bool

is_command_polled(*self*, *cmd_name*) → bool

True if the command is polled.

Parameters
cmd_name (*str*) – attribute name

Returns
True if the command is polled

Return type
bool

is_device_locked(*self*) → bool

Returns if this device is locked by a client.

Returns
True if it is locked or False otherwise

Return type
bool

New in PyTango 7.1.2

is_polled(*self*) → bool

Returns if it is polled

Returns
True if it is polled or False otherwise

Return type
bool

New in PyTango 7.1.2

is_there_subscriber(*self*, *att_name*, *event_type*) → bool

Check if there is subscriber(s) listening for the event.

This method returns a boolean set to true if there are some subscriber(s) listening on the event specified by the two method arguments. Be aware that there is some delay (up to 600 sec) between this method returning false and the last subscriber unsubscription or crash…

The device interface change event is not supported by this method.

Parameters

- **att_name** (*str*) – the attribute name
- **event_type** (*EventType*) – the event type

Returns
True if there is at least one listener or False otherwise

Return type
bool

poll_attribute(*self*, *attr_name*, *period*) → None

Add an attribute to the list of polled attributes.

Parameters

- **attr_name** (*str*) – attribute name
- **period** (*int*) – polling period in milliseconds

Returns
None

Return type
None

poll_command (*self, cmd_name, period*) → None

Add a command to the list of polled commands.

Parameters

- **cmd_name** (*str*) – attribute name
- **period** (*int*) – polling period in milliseconds

Returns
None

Return type
None

push_archive_event (*self, attr_name*)

push_archive_event (*self, attr_name, except*)

push_archive_event (*self, attr_name, data, dim_x=1, dim_y=0*)

push_archive_event (*self, attr_name, str_data, data*)

push_archive_event (*self, attr_name, data, time_stamp, quality, dim_x=1, dim_y=0*)

push_archive_event (*self, attr_name, str_data, data, time_stamp, quality*)

Push an archive event for the given attribute name.

The event is pushed to the notification daemon.

Parameters

- **attr_name** (*str*) – attribute name
- **data** – the data to be sent as attribute event data. Data must be compatible with the attribute type and format. for SPECTRUM and IMAGE attributes, data can be any type of sequence of elements compatible with the attribute type
- **str_data** (*str*) – special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.
- **except** (*DevFailed*) – Instead of data, you may want to send an exception.
- **dim_x** (*int*) – the attribute x length. Default value is 1
- **dim_y** (*int*) – the attribute y length. Default value is 0
- **time_stamp** (*double*) – the time stamp
- **quality** (*AttrQuality*) – the attribute quality factor

Raises
DevFailed – If the attribute data type is not coherent.

push_att_conf_event (*self, attr*)

Push an attribute configuration event.

Parameters
attr (*Attribute*) – the attribute for which the configuration event will be sent.

New in PyTango 7.2.1

push_change_event (*self, attr_name*)

push_change_event (*self, attr_name, except*)

push_change_event (*self, attr_name, data, dim_x=1, dim_y=0*)

push_change_event (*self, attr_name, str_data, data*)

push_change_event (*self, attr_name, data, time_stamp, quality, dim_x=1, dim_y=0*)

push_change_event (*self, attr_name, str_data, data, time_stamp, quality*)

Push a change event for the given attribute name.

The event is pushed to the notification daemon.

Parameters

- **attr_name** (*str*) – attribute name
- **data** – the data to be sent as attribute event data. Data must be compatible with the attribute type and format. for SPECTRUM and IMAGE attributes, data can be any type of sequence of elements compatible with the attribute type
- **str_data** (*str*) – special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.
- **except** (DevFailed) – Instead of data, you may want to send an exception.
- **dim_x** (*int*) – the attribute x length. Default value is 1
- **dim_y** (*int*) – the attribute y length. Default value is 0
- **time_stamp** (*double*) – the time stamp
- **quality** (AttrQuality) – the attribute quality factor

Raises

DevFailed – If the attribute data type is not coherent.

push_data_ready_event (*self, attr_name, counter=0*)

Push a data ready event for the given attribute name.

The event is pushed to the notification daemon.

The method needs only the attribue name and an optional "counter" which will be passed unchanged within the event

Parameters

- **attr_name** (*str*) – attribute name
- **counter** (*int*) – the user counter

Raises

DevFailed – If the attribute name is unknown.

push_event (*self, attr_name, filt_names, filt_vals*)

push_event (*self, attr_name, filt_names, filt_vals, data, dim_x=1, dim_y=0*)

push_event (*self, attr_name, filt_names, filt_vals, str_data, data*)

push_event (*self, attr_name, filt_names, filt_vals, data, time_stamp, quality, dim_x=1, dim_y=0*)

push_event (*self, attr_name, filt_names, filt_vals, str_data, data, time_stamp, quality*)

Push a user event for the given attribute name.

The event is pushed to the notification daemon.

Parameters

- **attr_name** (*str*) – attribute name
- **filt_names** (*Sequence[str]*) – the filterable fields name
- **filt_vals** (*Sequence[double]*) – the filterable fields value
- **data** – the data to be sent as attribute event data. Data must be compatible with the attribute type and format. for SPECTRUM and IMAGE attributes, data can be any type of sequence of elements compatible with the attribute type
- **str_data** (*str*) – special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.
- **dim_x** (*int*) – the attribute x length. Default value is 1
- **dim_y** (*int*) – the attribute y length. Default value is 0
- **time_stamp** (*double*) – the time stamp
- **quality** (*AttrQuality*) – the attribute quality factor

Raises
DevFailed – If the attribute data type is not coherent.

push_pipe_event (*self, blob*)

Push an pipe event.

Parameters
blob – the blob which pipe event will be send.

New in PyTango 9.2.2

read_attr_hardware (*self, attr_list*)

Read the hardware to return attribute value(s).

Default method to implement an action necessary on a device to read the hardware involved in a read attribute CORBA call. This method must be redefined in sub-classes in order to support attribute reading

Parameters
attr_list (*Sequence[int]*) – list of indices in the device object attribute vector of an attribute to be read.

Raises
DevFailed – This method does not throw exception but a redefined method can.

register_signal (*self, signo*)

Register a signal.

Register this device as device to be informed when signal signo is sent to to the device server process

Parameters
signo (*int*) – signal identifier

remove_attribute (*self, attr_name*)

Remove one attribute from the device attribute list.

Parameters
attr_name (*str*) – attribute name

Raises
DevFailed –

remove_command (*self*, *attr_name*)

Remove one command from the device command list.

Parameters

- **cmd_name** (*str*) – command name to be removed from the list
- **free_it** (*bool*) – set to true if the command object must be freed.
- **clean_db** – Clean command related information (included polling info if the command is polled) from database.

Raises

DevFailed –

set_archive_event (*self*, *attr_name*, *implemented*, *detect=True*)

Set an implemented flag for the attribute to indicate that the server fires archive events manually, without the polling to be started.

If the detect parameter is set to true, the criteria specified for the archive event are verified and the event is only pushed if they are fullfilled. If detect is set to false the event is fired without any value checking!

Parameters

- **attr_name** (*str*) – attribute name
- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – Triggers the verification of the change event properties when set to true. Default value is true.

set_attribute_config_3 (*self*, *new_conf*) → None

Sets attribute configuration locally and in the Tango database

Parameters

new_conf (list[tango.AttributeConfig_3]) – The new attribute(s) configuration. One AttributeConfig structure is needed for each attribute to update

Returns

None

Return type

None

set_change_event (*self*, *attr_name*, *implemented*, *detect=True*)

Set an implemented flag for the attribute to indicate that the server fires change events manually, without the polling to be started.

If the detect parameter is set to true, the criteria specified for the change event are verified and the event is only pushed if they are fullfilled. If detect is set to false the event is fired without any value checking!

Parameters

- **attr_name** (*str*) – attribute name
- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – Triggers the verification of the change event properties when set to true. Default value is true.

set_data_ready_event (*self*, *attr_name*, *implemented*)

Set an implemented flag for the attribute to indicate that the server fires data ready events manually.

Parameters

- **attr_name** (*str*) – attribute name
- **implemented** (*bool*) – True when the server fires change events manually.

set_state (*self, new_state*)

Set device state.

Parameters
new_state (DevState) – the new device state

set_status (*self, new_status*)

Set device status.

Parameters
new_status (*str*) – the new device status

signal_handler (*self, signo*)

Signal handler.

The method executed when the signal arrived in the device server process. This method is defined as virtual and then, can be redefined following device needs.

Parameters
signo (*int*) – the signal number

Raises
DevFailed – This method does not throw exception but a redefined method can.

start_logging (*self*) → None

Starts logging

stop_logging (*self*) → None

Stops logging

stop_poll_attribute (*self, attr_name*) → None

Remove an attribute from the list of polled attributes.

Parameters
attr_name (*str*) – attribute name

Returns
None

Return type
None

stop_poll_command (*self, cmd_name*) → None

Remove a command from the list of polled commands.

Parameters
cmd_name (*str*) – cmd_name name

Returns
None

Return type
None

stop_polling (*self*)

stop_polling (*self, with_db_upd*)

Stop all polling for a device. if the device is polled, call this method before deleting it.

Parameters
with_db_upd (*bool*) – Is it necessary to update db?

New in PyTango 7.1.2

unregister_signal (*self, signo*)

Unregister a signal.

Unregister this device as device to be informed when signal signo is sent to to the device server process

Parameters
signo (*int*) – signal identifier

warn_stream (*self, msg, *args*)

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

write_attr_hardware (*self*)

Write the hardware for attributes.

Default method to implement an action necessary on a device to write the hardware involved in a write attribute. This method must be redefined in sub-classes in order to support writable attribute

Parameters
attr_list (*Sequence[int]*) – list of indices in the device object attribute vector of an attribute to be written.

Raises
DevFailed – This method does not throw exception but a redefined method can.

5.3.3 DeviceClass

class tango.**DeviceClass** (**args, **kwargs*)

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

DeviceClass.add_wiz_class_prop(self, str, str) -> None

DeviceClass.add_wiz_class_prop(self, str, str, str) -> None

For internal usage only

Parameters
None

Return
None

DeviceClass.add_wiz_dev_prop(self, str, str) -> None

DeviceClass.add_wiz_dev_prop(self, str, str, str) -> None

For internal usage only

Parameters
None

Return
None

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

export_device(*self, dev, corba_dev_name='Unused'*) → None

For internal usage only

Parameters

dev
(`DeviceImpl`) device object

corba_dev_name
(`str`) CORBA device name. Default value is 'Unused'

Return
None

get_class_attr(*self*) → None
Returns the instance of the `tango.MultiClassAttribute` for the class

Param
None

Returns
the instance of the `tango.MultiClassAttribute` for the class

Return type
`tango.MultiClassAttribute`

get_cmd_by_name (*self, (str)cmd_name*) → tango.Command

Get a reference to a command object.

Parameters

cmd_name
(str) command name

Return
(tango.Command) tango.Command object

New in PyTango 8.0.0

get_command_list (*self*) → sequence<tango.Command>

Gets the list of tango.Command objects for this class

Parameters
None

Return
(sequence<tango.Command>) list of tango.Command objects for this class

New in PyTango 8.0.0

get_cvs_location (*self*) → None

Gets the cvs localtion

Parameters
None

Return
(str) cvs location

get_cvs_tag (*self*) → str

Gets the cvs tag

Parameters
None

Return
(str) cvs tag

get_device_list (*self*) → sequence<tango.DeviceImpl>

Gets the list of tango.DeviceImpl objects for this class

Parameters
None

Return
(sequence<tango.DeviceImpl>) list of tango.DeviceImpl objects for this class

get_doc_url (*self*) → str

Get the TANGO device class documentation URL.

Parameters
None

Return
(`str`) the TANGO device type name

get_name (*self*) → str

Get the TANGO device class name.

Parameters
None

Return
(`str`) the TANGO device class name.

get_pipe_by_name (*self, pipe_name, dev_name*) → None
Returns the `Pipe` instance with name <pipe_name> for the specified device

Parameters

- **pipe_name** (`str`) – name of the pipe
- **dev_name** (`str`) – name of the device

Returns
tango.server.pipe object

Return type
tango.server.pipe

get_pipe_list (*self, dev_name*) → None
Returns the list of pipes for the specified device

Parameters
dev_name (*atr*) – name of the device

Returns
list of *tango.server.pipe* objects for device

Return type
tango.server.pipe

get_type (*self*) → str

Gets the TANGO device type name.

Parameters
None

Return
(`str`) the TANGO device type name

register_signal (*self, signo*) → None
register_signal (*self, signo, own_handler=false*) → None

Register a signal. Register this class as class to be informed when signal signo is sent to to the device server process. The second version of the method is available only under Linux.

Throws tango.DevFailed:

- if the signal number is out of range
- if the operating system failed to register a signal for the process.

Parameters

signo
(`int`) signal identifier

own_handler
(`bool`) true if you want the device signal handler to be executed in its own handler instead of being executed by the signal thread. If this parameter is set to true, care should be taken on how the handler is written. A default false value is provided

Return
None

set_type (*self*, *dev_type*) → None

Set the TANGO device type name.

Parameters

dev_type
(`str`) the new TANGO device type name

Return
None

signal_handler (*self*, *signo*) → None

Signal handler.

The method executed when the signal arrived in the device server process. This method is defined as virtual and then, can be redefined following device class needs.

Parameters

signo
(`int`) signal identifier

Return
None

unregister_signal (*self*, *signo*) → None

Unregister a signal. Unregister this class as class to be informed when signal signo is sent to to the device server process

Parameters

signo
(`int`) signal identifier

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**(**args*, ***kwargs*)

This class represents a Tango writable attribute.

check_type(*self*)

This method checks data type and throws an exception in case of unsupported data type

Raises
`DevFailed`: If the data type is unsupported.

get_assoc(*self*) → str

Get the associated name.

Returns
the associated name

Return type
bool

get_cl_name(*self*) → str

Returns the class name.

Returns
the class name

Return type
str

New in PyTango 7.2.0

get_class_properties (*self*) → Sequence[AttrProperty]

Get the class level attribute properties.

Returns
the class attribute properties

Return type
Sequence[AttrProperty]

get_disp_level (*self*) → *DispLevel*

Get the attribute display level.

Returns
the attribute display level

Return type
DispLevel

get_format (*self*) → *AttrDataFormat*

Get the attribute format.

Returns
the attribute format

Return type
AttrDataFormat

get_memorized (*self*) → bool

Determine if the attribute is memorized or not.

Returns
True if the attribute is memorized

Return type
bool

get_memorized_init (*self*) → bool

Determine if the attribute is written at startup from the memorized value if it is memorized.

Returns
True if initialized with memorized value or not

Return type
bool

get_name (*self*) → str

Get the attribute name.

Returns
the attribute name

Return type
str

get_polling_period (*self*) → int

Get the polling period (mS).

Returns
the polling period (mS)

Return type
int

get_type (*self*) → int

Get the attribute data type.

Returns
the attribute data type

Return type
int

get_user_default_properties (*self*) → Sequence[AttrProperty]

Get the user default attribute properties.

Returns
the user default attribute properties

Return type
Sequence[AttrProperty]

get_writable (*self*) → *AttrWriteType*

Get the attribute write type.

Returns
the attribute write type

Return type
AttrWriteType

is_allowed (*self, device, request_type*) → bool

Returns whether the request_type is allowed for the specified device

Parameters

- **device** (`tango.server.Device`) – instance of Device
- **request_type** (`AttReqType`) – AttReqType.READ_REQ for read request or AttReqType.WRITE_REQ for write request

Returns
True if request_type is allowed for the specified device

Return type
bool

is_archive_event (*self*) → bool

Check if the archive event is fired manually for this attribute.

Returns
true if a manual fire archive event is implemented.

Return type
bool

is_assoc (*self*) → bool

Determine if it is assoc.

Returns
if it is assoc

Return type
bool

is_change_event (*self*) → bool

Check if the change event is fired manually for this attribute.

Returns
true if a manual fire change event is implemented.

Return type
bool

is_check_archive_criteria(*self*) → bool

Check if the archive event criteria should be checked when firing the event manually.

Returns
true if a archive event criteria will be checked.

Return type
bool

is_check_change_criteria(*self*) → bool

Check if the change event criteria should be checked when firing the event manually.

Returns
true if a change event criteria will be checked.

Return type
bool

is_data_ready_event(*self*) → bool

Check if the data ready event is fired for this attribute.

Returns
true if firing data ready event is implemented.

Return type
bool

New in PyTango 7.2.0

set_archive_event(*self*)

Set a flag to indicate that the server fires archive events manually without the polling to be started for the attribute.

If the detect parameter is set to true, the criteria specified for the archive event are verified and the event is only pushed if they are fullfilled.

If detect is set to false the event is fired without checking!

Parameters

- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – Triggers the verification of the archive event properties when set to true.

set_change_event(*self, implemented, detect*)

Set a flag to indicate that the server fires change events manually without the polling to be started for the attribute.

If the detect parameter is set to true, the criteria specified for the change event are verified and the event is only pushed if they are fullfilled.

If detect is set to false the event is fired without checking!

Parameters

- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – Triggers the verification of the change event properties when set to true.

set_cl_name(*self, cl*)

Sets the class name.

Parameters
cl (*str*) – new class name

New in PyTango 7.2.0

set_class_properties (*self, props*)

Set the class level attribute properties.

Parameters
props (*StdAttrPropertyVector*) – new class level attribute properties

set_data_ready_event (*self, implemented*)

Set a flag to indicate that the server fires data ready events.

Parameters
implemented (*bool*) – True when the server fires data ready events

New in PyTango 7.2.0

set_default_properties (*self*)

Set default attribute properties.

Parameters
attr_prop (UserDefaultAttrProp) – the user default property class

set_disp_level (*self, disp_lelel*)

Set the attribute display level.

Parameters
disp_level (DispLevel) – the new display level

set_memorized (*self*)

Set the attribute as memorized in database (only for scalar and writable attribute).

With no argument the setpoint will be written to the attribute during initialisation!

set_memorized_init (*self, write_on_init*)

Set the initialisation flag for memorized attributes.

- true = the setpoint value will be written to the attribute on initialisation
- false = only the attribute setpoint is initialised.

No action is taken on the attribute

Parameters
write_on_init (*bool*) – if true the setpoint value will be written to the attribute on initialisation

set_polling_period (*self, period*)

Set the attribute polling update period.

Parameters
period (*int*) – the attribute polling period (in mS)

Attribute

class tango.**Attribute** (**args, **kwargs*)

This class represents a Tango attribute.

check_alarm (*self*) → bool

Check if the attribute read value is below/above the alarm level.

Returns
true if the attribute is in alarm condition.

Return type
bool

Raises
DevFailed – If no alarm level is defined.

get_assoc_ind (*self*) → int
Get index of the associated writable attribute.

Returns
the index in the main attribute vector of the associated writable attribute

Return type
int

get_assoc_name (*self*) → str
Get name of the associated writable attribute.

Returns
the associated writable attribute name

Return type
str

get_attr_serial_model (*self*) → AttrSerialModel
Get attribute serialization model.

Returns
The attribute serialization model

Return type
AttrSerialModel

New in PyTango 7.1.0

get_data_format (*self*) → *AttrDataFormat*
Get attribute data format.

Returns
the attribute data format

Return type
AttrDataFormat

get_data_size (*self*)
Get attribute data size.

Returns
the attribute data size

Return type
int

get_data_type (*self*) → int
Get attribute data type.

Returns
the attribute data type

Return type
int

get_date (*self*) → *TimeVal*
Get a COPY of the attribute date.

Returns
the attribute date

Return type
TimeVal

get_label (*self*) → str

Get attribute label property.

Returns
the attribute label

Return type
str

get_max_dim_x (*self*) → int

Get attribute maximum data size in x dimension.

Returns
the attribute maximum data size in x dimension. Set to 1 for scalar attribute

Return type
int

get_max_dim_y (*self*) → int

Get attribute maximum data size in y dimension.

Returns
the attribute maximum data size in y dimension. Set to 0 for scalar attribute

Return type
int

get_name (*self*) → str

Get attribute name.

Returns
The attribute name

Return type
str

get_polling_period (*self*) → int

Get attribute polling period.

Returns
The attribute polling period in mS. Set to 0 when the attribute is not polled

Return type
int

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

Returns
the config object filled with attribute configuration information

Return type
AttributeConfig

New in PyTango 7.1.4

get_quality (*self*) → *AttrQuality*

Get a COPY of the attribute data quality.

Returns
the attribute data quality

Return type
AttrQuality

`get_writable` (*self*) → *AttrWriteType*

Get the attribute writable type (RO/WO/RW).

Returns
The attribute write type.

Return type
AttrWriteType

`get_x` (*self*) → int

Get attribute data size in x dimension.

Returns
the attribute data size in x dimension. Set to 1 for scalar attribute

Return type
int

`get_y` (*self*) → int

Get attribute data size in y dimension.

Returns
the attribute data size in y dimension. Set to 0 for scalar attribute

Return type
int

`is_archive_event` (*self*) → bool

Check if the archive event is fired manually (without polling) for this attribute.

Returns
True if a manual fire archive event is implemented.

Return type
bool

New in PyTango 7.1.0

`is_change_event` (*self*) → bool

Check if the change event is fired manually (without polling) for this attribute.

Returns
True if a manual fire change event is implemented.

Return type
bool

New in PyTango 7.1.0

`is_check_archive_criteria` (*self*) → bool

Check if the archive event criteria should be checked when firing the event manually.

Returns
True if a archive event criteria will be checked.

Return type
bool

New in PyTango 7.1.0

`is_check_change_criteria` (*self*) → bool

Check if the change event criteria should be checked when firing the event manually.

Returns
True if a change event criteria will be checked.

Return type
bool

New in PyTango 7.1.0

is_data_ready_event (*self*) → bool
Check if the data ready event is fired manually (without polling) for this attribute.

Returns
True if a manual fire data ready event is implemented.

Return type
bool

New in PyTango 7.2.0

is_max_alarm (*self*) → bool
Check if the attribute is in maximum alarm condition.

Returns
true if the attribute is in alarm condition (read value above the max. alarm).

Return type
bool

is_max_warning (*self*) → bool
Check if the attribute is in maximum warning condition.

Returns
true if the attribute is in warning condition (read value above the max. warning).

Return type
bool

is_min_alarm (*self*) → bool
Check if the attribute is in minimum alarm condition.

Returns
true if the attribute is in alarm condition (read value below the min. alarm).

Return type
bool

is_min_warning (*self*) → bool
Check if the attribute is in minimum warning condition.

Returns
true if the attribute is in warning condition (read value below the min. warning).

Return type
bool

is_polled (*self*) → bool
Check if the attribute is polled.

Returns
true if the attribute is polled.

Return type
bool

is_rds_alarm (*self*) → bool
Check if the attribute is in RDS alarm condition.

Returns
true if the attribute is in RDS condition (Read Different than Set).

Return type
bool

is_write_associated(*self*) → bool

Check if the attribute has an associated writable attribute.

Returns
True if there is an associated writable attribute

Return type
bool

remove_configuration(*self*)

Remove the attribute configuration from the database.

This method can be used to clean-up all the configuration of an attribute to come back to its default values or the remove all configuration of a dynamic attribute before deleting it.

The method removes all configured attribute properties and removes the attribute from the list of polled attributes.

New in PyTango 7.1.0

set_archive_event(*self*, *implemented*, *detect=True*)

Set a flag to indicate that the server fires archive events manually, without the polling to be started for the attribute.

If the detect parameter is set to true, the criteria specified for the archive event are verified and the event is only pushed if they are fullfilled.

Parameters

- **implemented** (*bool*) – True when the server fires archive events manually.
- **detect** (*bool*) – (optional, default is True) Triggers the verification of the archive event properties when set to true.

New in PyTango 7.1.0

set_assoc_ind(*self*, *index*)

Set index of the associated writable attribute.

Parameters
index (*int*) – The new index in the main attribute vector of the associated writable attribute

set_attr_serial_model(*self*, *ser_model*) → void

Set attribute serialization model.

This method allows the user to choose the attribute serialization model.

Parameters
ser_model (*AttrSerialModel*) – The new serialisation model. The serialization model must be one of ATTR_BY_KERNEL, ATTR_BY_USER or ATTR_NO_SYNC

New in PyTango 7.1.0

set_change_event(*self*, *implemented*, *detect=True*)

Set a flag to indicate that the server fires change events manually, without the polling to be started for the attribute.

If the detect parameter is set to true, the criteria specified for the change event are verified and the event is only pushed if they are fullfilled. If detect is set to false the event is fired without any value checking!

Parameters

- **implemented** (*bool*) – True when the server fires change events manually.
- **detect** (*bool*) – (optional, default is True) Triggers the verification of the change event properties when set to true.

New in PyTango 7.1.0

set_data_ready_event (*self, implemented*)

Set a flag to indicate that the server fires data ready events.

Parameters
implemented (*bool*) – True when the server fires data ready events manually.

New in PyTango 7.2.0

set_date (*self, new_date*)

Set attribute date.

Parameters
new_date (TimeVal) – the attribute date

set_properties (*self, attr_cfg, dev*)

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

set_quality (*self, quality, send_event=False*)

Set attribute data quality.

Parameters

- **quality** (AttrQuality) – the new attribute data quality
- **send_event** (*bool*) – true if a change event should be sent. Default is false.

set_value ()

set_value(self, data, dim_x = 1, dim_y = 0) <= DEPRECATED

set_value (*self, data*)

set_value (*self, str_data, data*)

Set internal attribute value.

This method stores the attribute read value inside the object. This method also stores the date when it is called and initializes the attribute quality factor.

param data
the data to be set. Data must be compatible with the attribute type and format. In the DEPRECATED form for SPECTRUM and IMAGE attributes, data can be any type of FLAT sequence of elements compatible with the attribute type. In the new form

(without dim_x or dim_y) data should be any sequence for SPECTRUM and a SEQUENCE of equal-length SEQUENCES for IMAGE attributes. The recommended sequence is a C continuous and aligned numpy array, as it can be optimized.

param str_data
special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.

type str_data
str

param dim_x
[DEPRECATED] the attribute x length. Default value is 1

type dim_x
int

param dim_y
[DEPRECATED] the attribute y length. Default value is 0

type dim_y
int

set_value_date_quality()

set_value_date_quality(self, data, time_stamp, quality, dim_x = 1, dim_y = 0) <= DEPRECATED

set_value_date_quality(*self*, *data*, *time_stamp*, *quality*)

set_value_date_quality(*self*, *str_data*, *data*, *time_stamp*, *quality*)

Set internal attribute value, date and quality factor.

This method stores the attribute read value, the date and the attribute quality factor inside the object.

param data
the data to be set. Data must be compatible with the attribute type and format. In the DEPRECATED form for SPECTRUM and IMAGE attributes, data can be any type of FLAT sequence of elements compatible with the attribute type. In the new form (without dim_x or dim_y) data should be any sequence for SPECTRUM and a SEQUENCE of equal-length SEQUENCES for IMAGE attributes. The recommended sequence is a C continuous and aligned numpy array, as it can be optimized.

param str_data
special variation for DevEncoded data type. In this case 'data' must be a str or an object with the buffer interface.

type str_data
str

param dim_x
[DEPRECATED] the attribute x length. Default value is 1

type dim_x
int

param dim_y
[DEPRECATED] the attribute y length. Default value is 0

type dim_y
int

param time_stamp
the time stamp

type time_stamp
double

param quality
the attribute quality factor

type quality
AttrQuality

WAttribute

class `tango.`**`WAttribute`**(**args*, ***kwargs*)

This class represents a Tango writable attribute.

`get_max_value`(*self*) → obj

Get attribute maximum value or throws an exception if the attribute does not have a maximum value.

Returns
an object with the python maximum value

Return type
obj

`get_min_value`(*self*) → obj

Get attribute minimum value or throws an exception if the attribute does not have a minimum value.

Returns
an object with the python minimum value

Return type
obj

`get_write_value`()

get_write_value(self, lst) <= DEPRECATED

`get_write_value`(*self*, *extract_as=ExtractAs.Numpy*) → obj

Retrieve the new value for writable attribute.

param extract_as
type extract_as
ExtractAs

param lst
[out] a list object that will be filled with the attribute write value (DEPRECATED)

type lst
list

returns
the attribute write value.

rtype
obj

`get_write_value_length`(*self*) → int

Retrieve the new value length (data number) for writable attribute.

Returns
the new value data length

Return type
int

is_max_value (*self*) → bool
Check if the attribute has a maximum value.

Returns
true if the attribute has a maximum value defined

Return type
bool

is_min_value (*self*) → bool
Check if the attribute has a minimum value.

Returns
true if the attribute has a minimum value defined

Return type
bool

set_max_value (*self, data*)
Set attribute maximum value.

Parameters
data – the attribute maximum value. python data type must be compatible with the attribute data format and type.

set_min_value (*self, data*)
Set attribute minimum value.

Parameters
data – the attribute minimum value. python data type must be compatible with the attribute data format and type.

MultiAttribute

class tango.**MultiAttribute** (**args, **kwargs*)
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

check_alarm (*self*) → bool

check_alarm (*self, attr_name*) → bool

check_alarm (*self, ind*) → bool

Checks an alarm.

- The 1st version of the method checks alarm on all attribute(s) with an alarm defined.
- The 2nd version of the method checks alarm for one attribute with a given name.
- The 3rd version of the method checks alarm for one attribute from its index in the main attributes vector.

Parameters

- **attr_name** (`str`) – attribute name
- **ind** (`int`) – the attribute index

Returns
True if at least one attribute is in alarm condition

Return type
bool

Raises

`DevFailed` – If at least one attribute does not have any alarm level defined

New in PyTango 7.0.0

get_attr_by_ind(*self*, *ind*) → *Attribute*

Get `Attribute` object from its index.

This method returns an `Attribute` object from the index in the main attribute vector.

Parameters

ind (`int`) – the attribute index

Returns

the attribute object

Return type

Attribute

get_attr_by_name(*self*, *attr_name*) → *Attribute*

Get `Attribute` object from its name.

This method returns an `Attribute` object with a name passed as parameter. The equality on attribute name is case independant.

Parameters

attr_name (`str`) – attribute name

Returns

the attribute object

Return type

Attribute

Raises

`DevFailed` – If the attribute is not defined.

get_attr_ind_by_name(*self*, *attr_name*) → int

Get Attribute index into the main attribute vector from its name.

This method returns the index in the Attribute vector (stored in the `MultiAttribute` object) of an attribute with a given name. The name equality is case independant.

Parameters

attr_name (`str`) – attribute name

Returns

the attribute index

Return type

int

Raises

`DevFailed` – If the attribute is not found in the vector.

New in PyTango 7.0.0

get_attr_nb(*self*) → int

Get attribute number.

Returns

the number of attributes

Return type

int

New in PyTango 7.0.0

get_attribute_list (*self*) → Sequence[*Attribute*]

Get the list of attribute objects.

Returns
list of attribute objects

Return type
Sequence[*Attribute*]

New in PyTango 7.2.1

get_w_attr_by_ind (*self, ind*) → *WAttribute*

Get a writable attribute object from its index.

This method returns an `WAttribute` object from the index in the main attribute vector.

Parameters
ind (`int`) – the attribute index

Returns
the attribute object

Return type
WAttribute

get_w_attr_by_name (*self, attr_name*) → *WAttribute*

Get a writable attribute object from its name.

This method returns an `WAttribute` object with a name passed as parameter. The equality on attribute name is case independant.

Parameters
attr_name (`str`) – attribute name

Returns
the attribute object

Return type
WAttribute

Raises
DevFailed – If the attribute is not defined.

read_alarm (*self, status*)

Add alarm message to device status.

This method add alarm mesage to the string passed as parameter. A message is added for each attribute which is in alarm condition

Parameters
status (`str`) – a string (should be the device status)

New in PyTango 7.0.0

UserDefaultAttrProp

class tango.**UserDefaultAttrProp** (**args, **kwargs*)

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_abs_change()

set_abs_change(self, def_abs_change) <= DEPRECATED

Set default change event abs_change property.

param def_abs_change
the user default change event abs_change property

type def_abs_change
str

Deprecated since PyTango 8.0. Please use set_event_abs_change instead.

set_archive_abs_change()

set_archive_abs_change(self, def_archive_abs_change) <= DEPRECATED

Set default archive event abs_change property.

param def_archive_abs_change
the user default archive event abs_change property

type def_archive_abs_change
str

Deprecated since PyTango 8.0. Please use set_archive_event_abs_change instead.

set_archive_event_abs_change(*self*, *def_archive_abs_change*)

Set default archive event abs_change property.

Parameters
def_archive_abs_change (*str*) – the user default archive event abs_change property

New in PyTango 8.0

set_archive_event_period(*self*, *def_archive_period*)

Set default archive event period property.

Parameters
def_archive_period (*str*) – t

New in PyTango 8.0

set_archive_event_rel_change(*self*, *def_archive_rel_change*)

Set default archive event rel_change property.

Parameters
def_archive_rel_change (*str*) – the user default archive event rel_change property

New in PyTango 8.0

set_archive_period()

set_archive_period(self, def_archive_period) <= DEPRECATED

Set default archive event period property.

param def_archive_period
t

type def_archive_period
str

Deprecated since PyTango 8.0. Please use set_archive_event_period instead.

set_archive_rel_change()

set_archive_rel_change(self, def_archive_rel_change) <= DEPRECATED

Set default archive event rel_change property.

param def_archive_rel_change
the user default archive event rel_change property

type def_archive_rel_change
str

Deprecated since PyTango 8.0. Please use set_archive_event_rel_change instead.

set_delta_t(*self*, *def_delta_t*)

Set default RDS alarm delta_t property.

Parameters
def_delta_t (`str`) – the user default RDS alarm delta_t property

set_delta_val(*self*, *def_delta_val*)

Set default RDS alarm delta_val property.

Parameters
def_delta_val (`str`) – the user default RDS alarm delta_val property

set_description(*self*, *def_description*)

Set default description property.

Parameters
def_description (`str`) – the user default description property

set_display_unit(*self*, *def_display_unit*)

Set default display unit property.

Parameters
def_display_unit (`str`) – the user default display unit property

set_enum_labels(*self*, *enum_labels*)

Set default enumeration labels.

Parameters
enum_labels (`Sequence[str]`) – list of enumeration labels

New in PyTango 9.2.0

set_event_abs_change(*self*, *def_abs_change*)

Set default change event abs_change property.

Parameters
def_abs_change (`str`) – the user default change event abs_change property

New in PyTango 8.0

set_event_period(*self*, *def_period*)

Set default periodic event period property.

Parameters
def_period (`str`) – the user default periodic event period property

New in PyTango 8.0

set_event_rel_change(*self*, *def_rel_change*)

Set default change event rel_change property.

Parameters
def_rel_change (`str`) – the user default change event rel_change property

New in PyTango 8.0

set_format (*self*, *def_format*)

Set default format property.

Parameters
def_format (*str*) – the user default format property

set_label (*self*, *def_label*)

Set default label property.

Parameters
def_label (*str*) – the user default label property

set_max_alarm (*self*, *def_max_alarm*)

Set default max_alarm property.

Parameters
def_max_alarm (*str*) – the user default max_alarm property

set_max_value (*self*, *def_max_value*)

Set default max_value property.

Parameters
def_max_value (*str*) – the user default max_value property

set_max_warning (*self*, *def_max_warning*)

Set default max_warning property.

Parameters
def_max_warning (*str*) – the user default max_warning property

set_min_alarm (*self*, *def_min_alarm*)

Set default min_alarm property.

Parameters
def_min_alarm (*str*) – the user default min_alarm property

set_min_value (*self*, *def_min_value*)

Set default min_value property.

Parameters
def_min_value (*str*) – the user default min_value property

set_min_warning (*self*, *def_min_warning*)

Set default min_warning property.

Parameters
def_min_warning (*str*) – the user default min_warning property

set_period ()

set_period(self, def_period) <= DEPRECATED

Set default periodic event period property.

param def_period
the user default periodic event period property

type def_period
str

Deprecated since PyTango 8.0. Please use set_event_period instead.

set_rel_change ()

set_rel_change(self, def_rel_change) <= DEPRECATED

Set default change event rel_change property.

param def_rel_change
the user default change event rel_change property

type def_rel_change
str

Deprecated since PyTango 8.0. Please use set_event_rel_change instead.

set_standard_unit (*self*, *def_standard_unit*)
Set default standard unit property.

Parameters
def_standard_unit (*str*) – the user default standard unit property

set_unit (*self*, *def_unit*)
Set default unit property.

Parameters
def_unit (*str*) – te user default unit property

5.3.6 Util

class tango.**Util** (**args*, ***kwargs*)
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

connect_db (*self*) → None

Connect the process to the TANGO database. If the connection to the database failed, a message is displayed on the screen and the process is aborted

Parameters
None

Return
None

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

get_database (*self*) → *Database*

Get a reference to the TANGO database object

Parameters
None

Return
(*Database*) the database

New in PyTango 7.0.0

get_device_by_name (*self, dev_name*) → DeviceImpl

Get a device reference from its name

Parameters

dev_name
(`str`) The TANGO device name

Return
(`DeviceImpl`) The device reference

New in PyTango 7.0.0

get_device_ior (*self, device*) → str

Get the CORBA Interoperable Object Reference (IOR) associated with the device

Parameters
device (*tango.LatestDeviceImpl*) – *tango.LatestDeviceImpl* device object

Returns
the associated CORBA object reference

Return type
str

get_device_list (*self*) → sequence<DeviceImpl>

Get device list from name. It is possible to use a wild card ('*') in the name parameter (e.g. "*", "/tango/tangotest/n*", ...)

Parameters
None

Return
(sequence<DeviceImpl>) the list of device objects

New in PyTango 7.0.0

get_device_list_by_class (*self, class_name*) → sequence<DeviceImpl>

Get the list of device references for a given TANGO class. Return the list of references for all devices served by one implementation of the TANGO device pattern implemented in the process.

Parameters

class_name
(str) The TANGO device class name

Return
(sequence<DeviceImpl>) The device reference list

New in PyTango 7.0.0

get_ds_exec_name (*self*) → str

Get a COPY of the device server executable name.

Parameters
None

Return
(str) a COPY of the device server executable name.

New in PyTango 3.0.4

get_ds_inst_name (*self*) → str

Get a COPY of the device server instance name.

Parameters
None

Return
(str) a COPY of the device server instance name.

New in PyTango 3.0.4

get_ds_name (*self*) → str

Get the device server name. The device server name is the <device server executable name>/<the device server instance name>

Parameters
None

Return
(str) device server name

New in PyTango 3.0.4

get_dserver_device (*self*) → DServer

Get a reference to the dserver device attached to the device server process

Parameters
None

Return
(`DServer`) A reference to the dserver device

New in PyTango 7.0.0

get_dserver_ior (*self*, *device_server*) → str

Get the CORBA Interoperable Object Reference (IOR) associated with the device server

Parameters
device_server (`DServer`) – `DServer` device object

Returns
the associated CORBA object reference

Return type
str

get_host_name (*self*) → str

Get the host name where the device server process is running.

Parameters
None

Return
(`str`) the host name where the device server process is running

New in PyTango 3.0.4

get_pid (*self*) → TangoSys_Pid

Get the device server process identifier.

Parameters
None

Return
(`int`) the device server process identifier

get_pid_str (*self*) → str

Get the device server process identifier as a string.

Parameters
None

Return
(`str`) the device server process identifier as a string

New in PyTango 3.0.4

get_polling_threads_pool_size (*self*) → int

Get the polling threads pool size.

Parameters
None

Return
(`int`) the maximun number of threads in the polling threads pool

get_serial_model (*self*) → *SerialModel*

Get the serialization model.

Parameters
None

Return
(*`SerialModel`*) the serialization model

get_server_version (*self*) → str

Get the device server version.

Parameters
None

Return
(`str`) the device server version.

get_sub_dev_diag (*self*) → SubDevDiag

Get the internal sub device manager

Parameters
None

Return
(`SubDevDiag`) the sub device manager

New in PyTango 7.0.0

get_tango_lib_release (*self*) → int

Get the TANGO library version number.

Parameters
None

Return
(`int`) The Tango library release number coded in 3 digits (for instance 550,551,552,600,. . . .)

get_trace_level (*self*) → int

Get the process trace level.

Parameters
None

Return
(`int`) the process trace level.

get_version_str (*self*) → str

Get the IDL TANGO version.

Parameters
None

Return
(str) the IDL TANGO version.

New in PyTango 3.0.4

init (**args*) → *Util*

Static method that creates and gets the singleton object reference. This method returns a reference to the object of the Util class. If the class singleton object has not been created, it will be instantiated

Parameters
***args** (*str*) – the process commandline arguments

Returns
Util the tango Util object

Return type
Util

instance (*exit=True*) → *Util*

Static method that gets the singleton object reference. If the class has not been initialised with it's init method, this method prints a message and aborts the device server process.

Parameters
exit (*bool*) – exit or throw DevFailed

Returns
the tango *Util* object

Return type
Util

Raises
DevFailed instead of aborting if exit is set to False

is_device_restarting (*self, (str)dev_name*) → bool

Check if the device is actually restarted by the device server process admin device with its DevRestart command

Parameters
dev_name : (str) device name

Return
(bool) True if the device is restarting.

New in PyTango 8.0.0

is_svr_shutting_down (*self*) → bool

Check if the device server process is in its shutting down sequence

Parameters
None

Return
(bool) True if the server is in its shutting down phase.

New in PyTango 8.0.0

is_svr_starting (*self*) → bool

Check if the device server process is in its starting phase

Parameters
None

Return
(bool) True if the server is in its starting phase

New in PyTango 8.0.0

orb_run (*self*) → None

Run the CORBA event loop directly (EXPERT FEATURE!)

This method runs the CORBA event loop. It may be useful if the Util.server_run method needs to be bypassed. Normally, that method runs the CORBA event loop.

Parameters
None

Return
None

reset_filedatabase (*self*) → None

Reread the file database

Parameters
None

Return
None

New in PyTango 7.0.0

server_cleanup (*self*) → None

Release device server resources (EXPERT FEATURE!)

This method cleans up the Tango device server and relinquishes all computer resources before the process exits. It is unnecessary to call this, unless Util.server_run has been bypassed.

server_init (*self, with_window=False*) → None

Initialize all the device server pattern(s) embedded in a device server process.

Parameters

with_window
(bool) default value is False

Return
None

Throws
DevFailed If the device pattern initialistaion failed

server_run (*self*) → None

Run the CORBA event loop. This method runs the CORBA event loop. For UNIX or Linux operating system, this method does not return. For Windows in a non-console mode, this method start a thread which enter the CORBA event loop.

Parameters
None

Return
None

server_set_event_loop (*self, event_loop*) → None

This method registers an event loop function in a Tango server. This function will be called by the process main thread in an infinite loop The process will not use the classical ORB blocking event loop. It is the user responsability to code this function in a way that it implements some kind of blocking in order not to load the computer CPU. The following piece of code is an example of how you can use this feature:

```
_LOOP_NB = 1
def looping():
    global _LOOP_NB
    print "looping", _LOOP_NB
    time.sleep(0.1)
    _LOOP_NB += 1
    return _LOOP_NB > 100

def main():
    py = tango.Util(sys.argv)

    # ...

    U = tango.Util.instance()
    U.server_set_event_loop(looping)
    U.server_init()
    U.server_run()
```

Parameters
None

Return
None

New in PyTango 8.1.0

set_polling_threads_pool_size (*self, thread_nb*) → None

Set the polling threads pool size.

Parameters

thread_nb
(int) the maximun number of threads in the polling threads pool

Return
None

New in PyTango 7.0.0

set_serial_model (*self, ser*) → None

Set the serialization model.

Parameters

ser
(`SerialModel`) the new serialization model. The serialization model must be one of BY_DEVICE, BY_CLASS, BY_PROCESS or NO_SYNC

Return
None

set_server_version (*self*, *vers*) → None

Set the device server version.

Parameters

vers
(`str`) the device server version

Return
None

set_trace_level (*self*, *level*) → None

Set the process trace level.

Parameters

level
(`int`) the new process level

Return
None

trigger_attr_polling (*self*, *dev*, *name*) → None

Trigger polling for polled attribute. This method send the order to the polling thread to poll one object registered with an update period defined as "externally triggerred"

Parameters

dev
(`DeviceImpl`) the TANGO device

name
(`str`) the attribute name which must be polled

Return
None

trigger_cmd_polling (*self*, *dev*, *name*) → None

Trigger polling for polled command. This method send the order to the polling thread to poll one object registered with an update period defined as "externally triggerred"

Parameters

dev
(`DeviceImpl`) the TANGO device

name
(`str`) the command name which must be polled

Return
None

Throws
DevFailed If the call failed

unregister_server (*self*) → None

Unregister a device server process from the TANGO database. If the database call fails, a message is displayed on the screen and the process is aborted

Parameters
None

Return
None

New in PyTango 7.0.0

5.4 Database API

class tango.**Database** (**args*, ***kwargs*)

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.

If TANGO_HOST env is not set, or you want to connect to a specific database, you can provide host and port to constructor:

db = Database(host: str, port: int)

or:

db = Database(host: str, port: str)

Alternatively, it is possible to start Database using file instead of a real database:

db = Database(filename: str)

add_device (*self*, *dev_info*) → None

Add a device to the database. The device name, server and class are specified in the DbDevInfo structure

Example

```
dev_info = DbDevInfo()
dev_info.name = 'my/own/device'
dev_info._class = 'MyDevice'
dev_info.server = 'MyServer/test'
db.add_device(dev_info)
```

Parameters

dev_info
(*DbDevInfo*) device information

Return
None

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_info1, 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_info2.name = 'dserver/' + dev_info1.server
dev_info2._class = 'DServer
dev_info2.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)

build_connection (*self*) → None

Tries to build a connection to the Database server.

Parameters
None

Return
None

New in PyTango 7.0.0

check_access_control (*self, dev_name*) → *AccessControlType*

Check the access for the given device for this client.

Parameters

dev_name
(`str`) device name

Return
the access control type as a AccessControlType object

New in PyTango 7.0.0

check_tango_host (*self, tango_host_env*) → None

Check the TANGO_HOST environment variable syntax and extract database server host(s) and port(s) from it.

Parameters

tango_host_env
(`str`) The TANGO_HOST env. variable value

Return
None

New in PyTango 7.0.0

delete_attribute_alias (*self, alias*) → None

Remove the alias associated to an attribute name.

Parameters

alias
(`str`) alias

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_pipe_property (*self*, *class_name*, *value*) → None

Delete a list of pipe 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 pipe names and value being a list of pipe 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 (*self*, *dev_name*) → None

Delete the device of the specified name from the database.

Parameters

dev_name
(str) device name

Return
None

delete_device_alias (*self*, *alias*) → void

Delete a device alias

Parameters

alias
(str) alias name

Return
None

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_pipe_property (*self*, *dev_name*, *value*) → None

Delete a list of pipe 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 pipe names and value being a list of pipe 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)

delete_server (*self, server*) → None

Delete the device server and its associated devices from database.

Parameters

server
(str) name of the server to be deleted with format: <server name>/<instance>

Return
None

delete_server_info (*self, server*) → None

Delete server information of the specifed server from the database.

Parameters

server
(str) name of the server to be deleted with format: <server name>/<instance>

Return
None

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

New in PyTango 3.0.4

export_device (*self, dev_export*) → None

Update the export info for this device in the database.

Example

```
dev_export = DbDevExportInfo()
dev_export.name = 'my/own/device'
dev_export.ior = <the real ior>
dev_export.host = <the host>
dev_export.version = '3.0'
dev_export.pid = '....'
db.export_device(dev_export)
```

Parameters

dev_export
(*DbDevExportInfo*) export information

Return
None

export_event (*self, event_data*) → None

Export an event to the database.

Parameters

eventdata (sequence<str>) event data (same as DbExportEvent Database command)

Return None

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

New in PyTango 7.0.0

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_access_except_errors (*self*) → DevErrorList

Returns a reference to the control access exceptions.

Parameters None

Return DevErrorList

New in PyTango 7.0.0

get_alias (*self, alias*) → str

Get the device alias name from its name.

Parameters

alias (str) device name

Return alias

New in PyTango 3.0.4

Deprecated since version 8.1.0: Use `get_alias_from_device()` instead

get_alias_from_attribute(*self*, *attr_name*) → str

Get the attribute alias from the full attribute name.

Parameters

attr_name
(`str`) full attribute name

Return
attribute alias

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

New in PyTango 8.1.0

get_alias_from_device(*self*, *alias*) → str

Get the device alias name from its name.

Parameters

alias
(`str`) device name

Return
alias

New in PyTango 8.1.0

get_attribute_alias(*self*, *alias*) → str

Get the full attribute name from an alias.

Parameters

alias
(`str`) attribute alias

Return
full attribute name

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

Deprecated since version 8.1.0: Use `:class:`Database()`.get_attribute_from_alias` instead

get_attribute_alias_list(*self*, *filter*) → *DbDatum*

Get attribute alias list. The parameter alias is a string to filter the alias list returned. Wildcard (*) is supported. For instance, if the string alias passed as the method parameter is initialised with only the * character, all the defined attribute alias will be returned. If there is no alias with the given filter, the returned array will have a 0 size.

Parameters

filter
(`str`) attribute alias filter

Return
DbDatum containing the list of matching attribute alias

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

get_attribute_from_alias (*self, alias*) → str

Get the full attribute name from an alias.

Parameters

alias
(str) attribute alias

Return
full attribute name

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

New in PyTango 8.1.0

get_class_attribute_list (*self, class_name, wildcard*) → *DbDatum*

Query the database for a list of attributes defined for the specified class which match the specified wildcard.

Parameters

class_name
(str) class name

wildcard
(str) attribute name

Return
DbDatum containing the list of matching attributes for the given class

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

New in PyTango 7.0.0

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_attribute_property_history (*self, dev_name, attr_name, prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specifed class attribute property. Note that prop_name and attr_name can contain a wildcard character (eg: 'prop*').

Parameters

dev_name
(str) device name

attr_name
(str) attribute name

prop_name
(str) property name

Return
DbHistoryList containing the list of modifications

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

New in PyTango 7.0.0

get_class_for_device (*self, dev_name*) → str

Return the class of the specified device.

Parameters

dev_name
(str) device name

Return
a string containing the device class

get_class_inheritance_for_device (*self, dev_name*) → *DbDatum*

Return the class inheritance scheme of the specified device.

Parameters

devn_ame
(str) device name

Return
DbDatum with the inheritance class list

New in PyTango 7.0.0

get_class_list (*self, wildcard*) → *DbDatum*

Query the database for a list of classes which match the specified wildcard

Parameters

wildcard
(`str`) class wildcard

Return
DbDatum containing the list of matching classes

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

New in PyTango 7.0.0

get_class_pipe_list (*self, class_name, wildcard*) → *DbDatum*

Query the database for a list of pipes defined for the specified class which match the specified wildcard. This corresponds to the pure C++ API call.

Parameters

class_name
(`str`) class name

wildcard
(`str`) pipe name

Return
DbDatum containing the list of matching pipes for the given class

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

get_class_pipe_property (*self, class_name, value*) → dict<str, dict<str, seq<str>>

Query the database for a list of class pipe properties for the specified class. The method returns all the properties for the specified pipes.

Parameters

class_name
(`str`) class name

propnames
can be one of the following:

1. str [in] - single pipe properties to be fetched
2. DbDatum [in] - single pipe properties to be fetched
3. DbData [in,out] - several pipe properties to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several pipe properties to be fetched

5. sequence<DbDatum> [in] - several pipe properties to be fetched
6. dict<str, obj> [in,out] - keys are pipe names In this case the given dict values will be changed to contain the several pipe property values

Return
a dictionary which keys are the pipe 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_pipe_property_history (*self, dev_name, pipe_name, prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specifed class pipe property. Note that prop_name and attr_name can contain a wildcard character (eg: 'prop*').

Parameters

dev_name
(str) device name

pipe_name
(str) pipe name

prop_name
(str) property name

Return
DbHistoryList containing the list of modifications

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_class_property_history (*self*, *class_name*, *prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specified class property. Note that propname can contain a wildcard character (eg: 'prop*').

Parameters

class_name
(`str`) class name

prop_name
(`str`) property name

Return
DbHistoryList containing the list of modifications

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

New in PyTango 7.0.0

get_class_property_list (*self*, *class_name*) → *DbDatum*

Query the database for a list of properties defined for the specified class.

Parameters

class_name
(`str`) class name

Return
DbDatum containing the list of properties for the specified class

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

get_device_alias (*self*, *alias*) → str

Get the device name from an alias.

Parameters

alias
(`str`) alias

Return
device name

Deprecated since version 8.1.0: Use `get_device_from_alias()` instead

get_device_alias_list (*self, filter*) → *DbDatum*

Get device alias list. The parameter alias is a string to filter the alias list returned. Wildcard (*) is supported.

Parameters

filter
(`str`) a string with the alias filter (wildcard (*) is supported)

Return
DbDatum with the list of device names

New in PyTango 7.0.0

get_device_attribute_list (*self, dev_name, att_list*) → None

Get the list of attribute(s) with some data defined in database for a specified device. Note that this is not the list of all device attributes because not all attribute(s) have some data in database This corresponds to the pure C++ API call.

Parameters

dev_name
(`str`) device name

att_list [out]
(`StdStringVector`) array that will contain the attribute name list

Return
None

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_attribute_property_history(*self, dev_name, attr_name, prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specified device attribute property. Note that propname and devname can contain a wildcard character (eg: 'prop*').

Parameters

dev_name
(`str`) device name

attr_name
(`str`) attribute name

prop_name
(`str`) property name

Return
DbHistoryList containing the list of modifications

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

New in PyTango 7.0.0

get_device_class_list(*self, server*) → *DbDatum*

Query the database for a list of devices and classes served by the specified server. Return a list with the following structure: [device name, class name, device name, class name, . . .]

Parameters

server
(`str`) name of the server with format: <server name>/<instance>

Return
DbDatum containing list with the following structure: [device_name, class name]

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

New in PyTango 3.0.4

get_device_domain(*self, wildcard*) → *DbDatum*

Query the database for a list of of device domain names which match the wildcard provided (* is wildcard for any character(s)). Domain names are case insensitive.

Parameters

wildcard
(`str`) domain filter

Return
DbDatum with the list of device domain names

get_device_exported(*self, filter*) → *DbDatum*

Query the database for a list of exported devices whose names satisfy the supplied filter (* is wildcard for any character(s))

Parameters

filter
(`str`) device name filter (wildcard)

Return
DbDatum with the list of exported devices

get_device_exported_for_class(*self, class_name*) → *DbDatum*

Query database for list of exported devices for the specified class.

Parameters

class_name
(`str`) class name

Return
DbDatum with the list of exported devices for the

New in PyTango 7.0.0

get_device_family(*self, wildcard*) → *DbDatum*

Query the database for a list of of device family names which match the wildcard provided (* is wildcard for any character(s)). Family names are case insensitive.

Parameters

wildcard
(`str`) family filter

Return
DbDatum with the list of device family names

get_device_from_alias(*self, alias*) → str

Get the device name from an alias.

Parameters

alias
(`str`) alias

Return
device name

New in PyTango 8.1.0

get_device_info(*self, dev_name*) → DbDevFullInfo

Query the databse for the full info of the specified device.

Example

```
dev_info = db.get_device_info('my/own/device')
print(dev_info.name)
print(dev_info.class_name)
print(dev_info.ds_full_name)
print(dev_info.exported)
print(dev_info.ior)
print(dev_info.version)
print(dev_info.pid)
print(dev_info.started_date)
print(dev_info.stopped_date)
```

Parameters

dev_name
(`str`) device name

Return
DbDevFullInfo

New in PyTango 8.1.0

get_device_member (*self*, *wildcard*) → *DbDatum*

Query the database for a list of of device member names which match the wildcard provided (* is wildcard for any character(s)). Member names are case insensitive.

Parameters

wildcard
(`str`) member filter

Return
DbDatum with the list of device member names

get_device_name (*self*, *serv_name*, *class_name*) → *DbDatum*

Query the database for a list of devices served by a server for a given device class

Parameters

serv_name
(`str`) server name

class_name
(`str`) device class name

Return
DbDatum with the list of device names

get_device_pipe_list (*self*, *dev_name*, *pipe_list*) → None

Get the list of pipe(s) with some data defined in database for a specified device. Note that this is not the list of all device pipes because not all pipe(s) have some data in database This corresponds to the pure C++ API call.

Parameters

dev_name
(`str`) device name

pipe_list [out]
(`StdStringVector`) array that will contain the pipe name list

Return
None

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

get_device_pipe_property(*self*, *dev_name*, *value*) → dict<str, dict<str, seq<str>>>

Query the database for a list of device pipe properties for the specified device. The method returns all the properties for the specified pipes.

Parameters

dev_name
(`string`) device name

value
can be one of the following:

1. str [in] - single pipe properties to be fetched
2. DbDatum [in] - single pipe properties to be fetched
3. DbData [in,out] - several pipe properties to be fetched In this case (direct C++ API) the DbData will be filled with the property values
4. sequence<str> [in] - several pipe properties to be fetched
5. sequence<DbDatum> [in] - several pipe properties to be fetched
6. dict<str, obj> [in,out] - keys are pipe names In this case the given dict values will be changed to contain the several pipe property values

Return
a dictionary which keys are the pipe 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_pipe_property_history(*self*, *dev_name*, *pipe_name*, *prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specified device pipe property. Note that propname and devname can contain a wildcard character (eg: 'prop*').

Parameters

dev_name
(`str`) device name

pipe_name
(`str`) pipe name

prop_name
(`str`) property name

Return
DbHistoryList containing the list of modifications

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_history (*self, dev_name, prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specified device property. Note that propname can contain a wildcard character (eg: 'prop*'). This corresponds to the pure C++ API call.

Parameters

serv_name
(str) server name

prop_name
(str) property name

Return
DbHistoryList containing the list of modifications

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

New in PyTango 7.0.0

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_device_service_list (*self, dev_name*) → *DbDatum*

Query database for the list of services provided by the given device.

Parameters

dev_name
(`str`) device name

Return
DbDatum with the list of services

New in PyTango 8.1.0

get_file_name (*self*) → str

Returns the database file name or throws an exception if not using a file database

Parameters
None

Return
a string containing the database file name

Throws
`DevFailed`

New in PyTango 7.2.0

get_host_list (*self*) → *DbDatum*

get_host_list (*self, wildcard*) → *DbDatum*

Returns the list of all host names registered in the database.

Parameters

wildcard
(str) (optional) wildcard (eg: '1-c0*')

Return
DbDatum with the list of registered host names

get_host_server_list (*self, host_name*) → *DbDatum*

Query the database for a list of servers registered on the specified host.

Parameters

host_name
(str) host name

Return
DbDatum containing list of servers for the specified host

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

New in PyTango 3.0.4

get_info (*self*) → str

Query the database for some general info about the tables.

Parameters
None

Return
a multiline string

get_instance_name_list (*self, serv_name*) → *DbDatum*

Return the list of all instance names existing in the database for the specifed server.

Parameters

serv_name
(str) server name with format <server name>

Return
DbDatum containing list of instance names for the specified server

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

New in PyTango 3.0.4

get_object_list (*self, wildcard*) → *DbDatum*

Query the database for a list of object (free properties) for which properties are defined and which match the specified wildcard.

Parameters

wildcard
(str) object wildcard

Return
DbDatum containing the list of object names matching the given wildcard

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

New in PyTango 7.0.0

get_object_property_list (*self, obj_name, wildcard*) → *DbDatum*

Query the database for a list of properties defined for the specified object and which match the specified wildcard.

Parameters

obj_name
(str) object name

wildcard
(str) property name wildcard

Return
DbDatum with list of properties defined for the specified object and which match the specified wildcard

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

New in PyTango 7.0.0

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)

get_property_history(*self*, *obj_name*, *prop_name*) → DbHistoryList

Get the list of the last 10 modifications of the specifed object property. Note that propname can contain a wildcard character (eg: 'prop*')

Parameters

serv_name
(`str`) server name

prop_name
(`str`) property name

Return
DbHistoryList containing the list of modifications

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

New in PyTango 7.0.0

get_server_class_list(*self*, *server*) → *DbDatum*

Query the database for a list of classes instantiated by the specified server. The DServer class exists in all TANGO servers and for this reason this class is removed from the returned list.

Parameters

server
(`str`) name of the server to be deleted with format: <server name>/<instance>

Return
DbDatum containing list of class names instanciated by the specified server

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

New in PyTango 3.0.4

`get_server_info` (*self*, *server*) → *DbServerInfo*

Query the database for server information.

Parameters

server
(`str`) name of the server with format: <server name>/<instance>

Return
DbServerInfo with server information

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

New in PyTango 3.0.4

`get_server_list` (*self*) → *DbDatum*

`get_server_list` (*self*, *wildcard*) → *DbDatum*

Return the list of all servers registered in the database. If wildcard parameter is given, then the list of matching servers will be returned (ex: Serial/*)

Parameters

wildcard
(`str`) host wildcard (ex: Serial/*)

Return
DbDatum containing list of registered servers

`get_server_name_list` (*self*) → *DbDatum*

Return the list of all server names registered in the database.

Parameters
None

Return
DbDatum containing list of server names

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

New in PyTango 3.0.4

get_services (*self, serv_name, inst_name*) → *DbDatum*

Query database for specified services.

Parameters

serv_name
(str) service name

inst_name
(str) instance name (can be a wildcard character ('*'))

Return
DbDatum with the list of available services

New in PyTango 3.0.4

import_device (*self, dev_name*) → *DbDevImportInfo*

Query the databse for the export info of the specified device.

Example

```
dev_imp_info = db.import_device('my/own/device')
print(dev_imp_info.name)
print(dev_imp_info.exported)
print(dev_imp_info.ior)
print(dev_imp_info.version)
```

Parameters

dev_name
(str) device name

Return
DbDevImportInfo

is_control_access_checked (*self*) → bool

Returns True if control access is checked or False otherwise.

Parameters
None

Return
(bool) True if control access is checked or False

New in PyTango 7.0.0

is_multi_tango_host (*self*) → bool

Returns if in multi tango host.

Parameters
None

Return
True if multi tango host or False otherwise

New in PyTango 7.1.4

put_attribute_alias (*self, attr_name, alias*) → None

Set an alias for an attribute name. The attribute alias is specified by alias and the attribute name is specifed by attr_name. If the given alias already exists, a DevFailed exception is thrown.

Parameters

attr_name
(str) full attribute name

alias
(str) alias

Return
None

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_pipe_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 pipe names and value being another dictionary which keys are the pipe 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_alias (*self, dev_name, alias*) → None

Query database for list of exported devices for the specified class.

Parameters

dev_name
(str) device name

alias
(str) alias name

Return
None

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_pipe_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 pipe names and value being another dictionary which keys are the pipe 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)

put_server_info (*self*, *info*) → None

Add/update server information in the database.

Parameters

info
(*DbServerInfo*) new server information

Return
None

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

New in PyTango 3.0.4

register_service (*self*, *serv_name*, *inst_name*, *dev_name*) → None

Register the specified service wihtin the database.

Parameters

serv_name
(`str`) service name

inst_name
(`str`) instance name

dev_name
(`str`) device name

Return
None

New in PyTango 3.0.4

`rename_server` (*self*, *old_ds_name*, *new_ds_name*) → None

Rename a device server process.

Parameters

old_ds_name
(`str`) old name

new_ds_name
(`str`) new name

Return
None

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

New in PyTango 8.1.0

`reread_filedatabase` (*self*) → None

Force a complete refresh over the database if using a file based database.

Parameters
None

Return
None

New in PyTango 7.0.0

`set_access_checked` (*self*, *val*) → None

Sets or unsets the control access check.

Parameters

val
(`bool`) True to set or False to unset the access control

Return
None

New in PyTango 7.0.0

`unexport_device` (*self*, *dev_name*) → None

Mark the specified device as unexported in the database

Example

```
db.unexport_device('my/own/device')
```

Parameters

dev_name
(`str`) device name

Return
None

unexport_event (*self, event*) → None

Un-export an event from the database.

Parameters

event
(`str`) event

Return
None

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

New in PyTango 7.0.0

unexport_server (*self, server*) → None

Mark all devices exported for this server as unexported.

Parameters

server
(`str`) name of the server to be unexported with format: <server name>/<instance>

Return
None

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

unregister_service (*self, serv_name, inst_name*) → None

Unregister the specified service from the database.

Parameters

serv_name
(`str`) service name

inst_name
(`str`) instance name

Return
None

New in PyTango 3.0.4

write_filedatabase(*self*) → None

Force a write to the file if using a file based database.

Parameters
None

Return
None

New in PyTango 7.0.0

class tango.**DbDatum**(**args*, ***kwargs*)

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

is_empty(*self*) → bool

Returns True or False depending on whether the DbDatum object contains data or not. It can be used to test whether a property is defined in the database or not.

Parameters
None

Return
(`bool`) True if no data or False otherwise.

New in PyTango 7.0.0

size(*self*) → int

Returns the number of separate elements in the value.

Parameters
None

Return
the number of separate elements in the value.

New in PyTango 7.0.0

class tango.**DbDevExportInfo**(**args*, ***kwargs*)

A structure containing export info for a device (should be retrieved 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.**DbDevExportInfos**(**args*, ***kwargs*)

class tango.**DbDevImportInfo**(**args*, ***kwargs*)

A structure containing import info for a device (should be retrieved 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.**DbDevImportInfos**(**args*, ***kwargs*)

class tango.**DbDevInfo**(**args*, ***kwargs*)

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**(**args*, ***kwargs*)

A structure containing the modifications of a property. No public members.

get_attribute_name(*self*) → str

Returns the attribute name (empty for object properties or device properties)

Parameters
None

Return
(`str`) attribute name

get_date(*self*) → str

Returns the update date

Parameters
None

Return
(`str`) update date

get_name(*self*) → str

Returns the property name.

Parameters
None

Return
(`str`) property name

get_value(*self*) → *DbDatum*

Returns a COPY of the property value

Parameters
None

Return
(*DbDatum*) a COPY of the property value

is_deleted(*self*) → bool

Returns True if the property has been deleted or False otherwise

Parameters
None

Return
(bool) True if the property has been deleted or False otherwise

class tango.**DbServerInfo**(**args, **kwargs*)
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**(**args, **kwargs*)

decode_gray16(*da, extract_as=None*)
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=None*)
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=None*)
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)
```

(continues on next page)

(continued from previous page)

```
    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.**get_enum_labels**(*enum_cls*)

Return list of enumeration labels from Enum class.

The list is useful when creating an attribute, for the *enum_labels* parameter. The enumeration values are checked to ensure they are unique, start at zero, and increment by one.

Parameters **enum_cls** (enum.Enum) – the Enum class to be inspected

Returns List of label strings

Return type list

Raises **EnumTypeError** – in case the given class is invalid

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_binary_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`)
- **conflics** (*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)
- **conflics** (*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.

```python
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:

```python
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**(**args, **kwargs*)

Bases:

A containner for the static methods:
- throw_exception
- re_throw_exception
- print_exception
- compare_exception

print_error_stack(*ex*) → None

Print all the details of a TANGO error stack.

Parameters

ex
(`tango.DevErrorList`) The error stack reference

print_exception(*ex*) → None

Print all the details of a TANGO exception.

Parameters

ex
(*tango.DevFailed*) The *DevFailed* exception

re_throw_exception(*ex*, *reason*, *desc*, *origin*, *sever=tango.ErrSeverity.ERR*) → None

Re-throw a TANGO *DevFailed* exception with one more error. The exception is re-thrown with one more *DevError* object. A default value *tango.ErrSeverity.ERR* is defined for the new *DevError* severity field.

Parameters

ex
(*tango.DevFailed*) The *DevFailed* exception

reason
(str) The exception *DevError* object reason field

desc
(str) The exception *DevError* object desc field

origin
(str) The exception *DevError* object origin field

sever
(*tango.ErrSeverity*) The exception DevError object severity field

Throws
DevFailed

throw_exception(*reason*, *desc*, *origin*, *sever=tango.ErrSeverity.ERR*) → None

Generate and throw a TANGO DevFailed exception. The exception is created with a single *DevError* object. A default value *tango.ErrSeverity.ERR* is defined for the *DevError* severity field.

Parameters

reason
(str) The exception *DevError* object reason field

desc
(str) The exception *DevError* object desc field

origin
(str) The exception *DevError* object origin field

sever
(*tango.ErrSeverity*) The exception DevError object severity field

Throws
DevFailed

throw_python_exception(*type*, *value*, *traceback*) → None

Generate and throw a TANGO DevFailed exception. The exception is created with a single *DevError* object. A default value *tango.ErrSeverity.ERR* is defined for the *DevError* severity field.

The parameters are the same as the ones generates by a call to `sys.exc_info()`.

Parameters

type
(`class`) the exception type of the exception being handled

value
(`object`) exception parameter (its associated value or the second argument to raise, which is always a class instance if the exception type is a class object)

traceback
(`traceback`) traceback object

Throws
`DevFailed`

New in PyTango 7.2.1

static to_dev_failed(*exc_type*, *exc_value*, *traceback*) → *tango.DevFailed*

Generate a TANGO DevFailed exception. The exception is created with a single *`DevError`* object. A default value *tango.ErrSeverity.ERR* is defined for the *`DevError`* severity field.

The parameters are the same as the ones generates by a call to `sys.exc_info()`.

Parameters

type
(`class`) the exception type of the exception being handled

value
(`object`) exception parameter (its associated value or the second argument to raise, which is always a class instance if the exception type is a class object)

traceback
(`traceback`) traceback object

Return
(*`tango.DevFailed`*) a tango exception object

New in PyTango 7.2.1

class tango.**DevError**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

exception tango.**ConnectionFailed**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

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**(**args*, ***kwargs*)

Bases:

exception tango.**NamedDevFailedList**(**args*, ***kwargs*)

Bases:

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

SIX

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 How to contribute

Everyone is welcome to contribute to PyTango project. If you don't feel comfortable with writing core PyTango we are looking for contributors to documentation or/and tests.

It refers to the next section, see *How to Contribute*.

6.2 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'
```

6.3 Check TANGO version

There are two library versions you might be interested in checking: The PyTango version:

```
>>> import tango
>>> tango.__version__
'9.3.4'
>>> tango.__version_info__
(9, 3, 4)
```

and the Tango C++ library version that PyTango was compiled with:

```
>>> import tango
>>> tango.constants.TgLibVers
'9.3.4'
```

6.4 Start server from command line

To start server from the command line execute the following command:

```
$ python <server_file>.py <instance_name>
Ready to accept request
```

To run server without database use option -nodb.

```
$ python <server_file>.py <instance_name> -nodb -port 10000
Ready to accept request
```

Note, that to start server in this mode you should provide a port with either --post, or --ORBendPoint option

Additionally, you can use the following options:

-h, -?, --help : show usage help

-v, --verbose: set the trace level. Can be user in count way: -vvvv set level to 4 or –verbose –verbose set to 2

-vN: directly set the trace level to N, e.g. -v3 - set level to 3

--file <file_name>: start a device server using an ASCII file instead of the Tango database

--host <host_name>: force the host from which server accept requests

--port <port>: force the port on which the device server listens

--nodb: run server without DB

--dlist <dev1,dev2,etc>: the device name list. This option is supported only with the -nodb option

--ORBendPoint giop:tcp:<host>:<port>: Specifying the host from which server accept requests and port on which the device server listens.

Note: any ORB option can be provided if it starts with -ORB<option>

Additionally in Windows the following option can be used:

-i: install the service

-s: install the service and choose the automatic startup mode

-u: uninstall the service

--dbg: run in console mode to debug service. The service must have been installed prior to use it.

Note: all long-options can be provided in non-POSIX format: -port or --port etc...

6.5 Report a bug

Bugs can be reported as issues in PyTango GitLab.

It is also helpful if you can put in the issue description the PyTango information. It can be a dump of:

```
$ python -c "from tango.utils import info; print(info())"
```

6.6 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())
```

6.7 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
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] ]
```

(continues on next page)

(continued from previous page)

```
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
```

6.8 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))
```

6.9 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.DevVarLongStringArray(argin)
print("Result of execution of DevVarLongArray command = {0}".
↪format(result))
```

6.10 Work with Groups

Todo: write this how to

6.11 Handle errors

Todo: write this how to

For now check *Exception API*.

6.12 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"

```

(continues on next page)

(continued from previous page)

```
# 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)
```

6.12.1 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"]
```

(continues on next page)

(continued from previous page)

```
axis1.put_property(axis_properties)
```

6.13 Using clients with multiprocessing

Since version 9.3.0 PyTango provides `cleanup()` which resets CORBA connection. This static function is needed when you want to use `tango` with `multiprocessing` in your client code.

In the case when both your parent process and your child process create `DeviceProxy`, `Database` or/and `AttributeProxy` your child process inherits the context from your parent process, i.e. open file descriptors, the TANGO and the CORBA state. Sharing the above objects between the processes may cause unpredictable errors, e.g. *TRANSIENT_CallTimedout*, *unidentifiable C++ exception*. Therefore, when you start a new process you must reset CORBA connection:

```python
import time
import tango

from multiprocessing import Process


class Worker(Process):

    def __init__(self):
        Process.__init__(self)

    def run(self):
        # reset CORBA connection
        tango.ApiUtil.cleanup()

        proxy = tango.DeviceProxy('test/tserver/1')

        stime = time.time()
        etime = stime
        while etime - stime < 1.:
            try:
                proxy.read_attribute("Value")
            except Exception as e:
                print(str(e))
            etime = time.time()


def runworkers():
    workers = [Worker() for _ in range(6)]
    for wk in workers:
        wk.start()
    for wk in workers:
        wk.join()


db = tango.Database()
dp = tango.DeviceProxy('test/tserver/1')

for i in range(4):
    runworkers()
```

After *cleanup()* all references to `DeviceProxy`, `AttributeProxy` or `Database` objects in the current process become invalid and these objects need to be reconstructed.

6.14 Using clients with multithreading

When performing Tango I/O from user-created threads, there can be problems. This is often more noticeable with event subscription and unsubscription, but it could affect any Tango I/O. As PyTango wraps the cppTango library, we needs to consider how cppTango's threads work.

cppTango was originally developed at a time where C++ didn't have standard threads. All the threads currently created in cppTango are omni threads, since this is what the omniORB library is using to create threads and since this implementation is available for free with omniORB.

In C++, users used to create omni threads in the past so there was no issue. Since C++11, C++ comes with an implementation of standard threads. cppTango is currently (version 9.3.3) not directly thread safe when a user is using C++11 standard threads or threads different than omni threads. This lack of thread safety includes threads created from Python's `threading` module.

In an ideal future cppTango should should protect itself, regardless of what type of threads are used. In the meantime, we need a work-around.

The work-around when using threads which are not omni threads is to create an object of the C++ class `omni_thread::ensure_self` in the user thread, just after the thread creation, and to delete this object only when the thread has finished its job. This `omni_thread::ensure_self` object provides a dummy omniORB ID for the thread. This ID is used when accessing thread locks within cppTango, so the ID must remain the same for the lifetime of the thread. Also note that this object MUST be released before the thread has exited, otherwise omniORB will throw an exception.

A Pythonic way to implement this work-around for multithreaded applications is available via the `EnsureOmniThread` class. It was added in PyTango version 9.3.2. This class is best used as a context handler to wrap the target method of the user thread. An example is shown below:

```python
import tango
from threading import Thread
from time import sleep


def thread_task():
    with tango.EnsureOmniThread():
        eid = dp.subscribe_event(
            "double_scalar", tango.EventType.PERIODIC_EVENT, cb)
        while running:
            print("num events stored {}".format(len(cb.get_events())))
            sleep(1)
        dp.unsubscribe_event(eid)


cb = tango.utils.EventCallback()  # print events to stdout
dp = tango.DeviceProxy("sys/tg_test/1")
dp.poll_attribute("double_scalar", 1000)
thread = Thread(target=thread_task)
running = True
thread.start()
sleep(5)
running = False
thread.join()
```

Another way to create threads in Python is the `concurrent.futures.ThreadPoolExecutor`. The problem with this is that the API does not provide an easy way for the context handler to cover the lifetime of the threads, which are created as daemons. One option is to at least use the context handler for the functions that are submitted to the executor. I.e., `executor.submit(thread_task)`. This is not guaranteed to work. A second option to investigate (if using at least Python 3.7) is the `initializer` argument which could be used to ensure a call to the `__enter__()` method for a thread-specific instance of `EnsureOmniThread`. However, calling the `__exit__()` method on the corresponding object at shutdown is a problem. Maybe it could be submitted as work.

6.15 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 detail 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

```
1  import time
2  from tango.server import Device, attribute, command, pipe
3
4
5  class Clock(Device):
6
7      @attribute
8      def time(self):
9          return time.time()
10
11     @command(dtype_in=str, dtype_out=str)
12     def strftime(self, format):
13         return time.strftime(format)
14
15     @pipe
16     def info(self):
17         return ('Information',
18                 dict(manufacturer='Tango',
19                      model='PS2000',
20                      version_number=123))
21
22
23 if __name__ == "__main__":
24     Clock.run_server()
```

line 2
: import the necessary symbols

line 5
: tango device class definition. A Tango device must inherit from *`tango.server.Device`*

line 7-9
: definition of the *time* attribute. By default, attributes are double, scalar, read-only. Check the *`attribute`* for the complete list of attribute options.

line 11-13
: 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 15-20
: definition of the *info* pipe. Check the *`pipe`* for the complete list of pipe options.

line 24
start the Tango run loop. This method automatically determines the Python class name and exports it as a Tango class. For more complicated cases, check `run()` for the complete list of options

There is a more detailed clock device server in the examples/Clock folder.

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

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

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


class PowerSupply(Device):

    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))
```

(continues on next page)

(continued from previous page)

```python

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


if __name__ == "__main__":
    PowerSupply.run_server()
```

6.16 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:

- Usage
- Property reference

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.

6.16.1 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:

```python
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
```

6.16.2 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
```

6.16.3 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()
```

6.17 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

class PLC(Device):

    # bla, bla my PLC code

if __name__ == "__main__":
    PLC.run_server()
```

... and a `IRMirror` in a `IRMirror.py`:

```
# IRMirror.py

from tango.server import Device

class IRMirror(Device):

    # bla, bla my IRMirror code

if __name__ == "__main__":
    IRMirror.run_server()
```

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:

```
1  import tango
2  import sys
3
4  if __name__ == '__main__':
5      py = tango.Util(sys.argv)
6      util.add_class('SerialLine', 'SerialLine', language="c++")
7      util.add_class(PLCClass, PLC, 'PLC')
8      util.add_class(IRMirrorClass, IRMirror, 'IRMirror')
9
10     U = tango.Util.instance()
11     U.server_init()
12     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

6.18 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 ways, 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 a 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 provides the user with a way to create dynamic attributes.

Using the high-level API, you can re-define a method called `initialize_dynamic_attributes()` on each <Device>. This method will be called automatically by the device_factory for each device. Within this method you create all the dynamic attributes.

If you are still using the low-level API with a <Device>Class instead of just a <Device>, then you can use the generic device_factory's call to the `dyn_attr()` method. It is simply necessary to re-define this method within your <Device>Class and to create the dynamic attributes within this method.

Internally, the high-level API re-defines dyn_attr() to call initialize_dynamic_attributes() for each device.

Note: The dyn_attr() (and initialize_dynamic_attributes() for high-level API) methods are only called once when the device server starts, since the Python device_factory method is only called once. Within the device_factory method, init_device() is called for all devices and only after that is dyn_attr() called for all devices. If the Init command is executed on a device it will not call the dyn_attr() method again.

There is another point to be noted regarding dynamic attributes within a Python device server. The Tango Python device server core checks that for each static attribute there exists methods named <attribute_name>_read and/or <attribute_name>_write and/or is_<attribute_name>_allowed. Using dynamic attributes, it is not possible to define these methods because attribute names and number are known only at run-time. To address this issue, you need to provide references to these methods when calling `add_attribute()`.

The recommended approach with the high-level API is to reference these methods when instantiating a *tango.server.attribute* object using the fget, fset and/or fisallowed kwargs (see example below). Where fget is the method which has to be executed when the attribute is read, fset is the method to be executed when the attribute is written and fisallowed is the method to be executed to implement the attribute state machine. This *tango.server.attribute* object is then passed to the *add_attribute()* method.

Note: The methods used for fget, fset and fisallowed must be methods that exist on your Device class. They cannot be plain functions, nor belong to a different class. You can pass a reference to the bound or unbound method, but during execution the bound method will be used.

Which arguments you have to provide depends on the type of the attribute. For example, a WRITE attribute does not need a read method.

Here is an example of a device which creates a dynamic attribute on startup:

```
from tango import AttrWriteType
from tango.server import Device, attribute

class MyDevice(Device):

    def initialize_dynamic_attributes(self):
        self._values = {"dyn_attr": 0}
        attr = attribute(
            name="dyn_attr",
            dtype=int,
            access=AttrWriteType.READ_WRITE,
            fget=self.generic_read,
            fset=self.generic_write,
            fisallowed=self.generic_is_allowed,
        )
        self.add_attribute(attr)

    def generic_read(self, attr):
        value = self._values[attr.get_name()]
        # unlike a normal static attribute read, we have to modify the value
        # inside this attr object, rather than just returning the value
        attr.set_value(value)

    def generic_write(self, attr):
        self._values[attr.get_name()] = attr.get_write_value()

    def generic_is_allowed(self, req_type):
        # note: we don't know which attribute is being read!
        # req_type will be either AttReqType.READ_REQ or AttReqType.WRITE_REQ
        return True
```

Another way to create dynamic attributes is to do it some time after the device has started. For example, using a command. In this case, we just call the *add_attribute()* method when necessary.

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:

```
from tango import AttrWriteType
from tango.server import Device, attribute, command

```

(continues on next page)

(continued from previous page)

```
class MyDevice(Device):

    def init_device(self):
        super(MyDevice, self).init_device()
        self._values = {}

    @command(dtype_in=str)
    def CreateFloatAttribute(self, attr_name):
        if attr_name not in self._values:
            self._values[attr_name] = 0.0
            attr = attribute(
                name=attr_name,
                dtype=float,
                access=AttrWriteType.READ_WRITE,
                fget=self.generic_read,
                fset=self.generic_write,
            )
            self.add_attribute(attr)
            self.info_stream("Added dynamic attribute %r", attr_name)
        else:
            raise ValueError("Already have an attribute called {!r}".format(attr_name))

    def generic_read(self, attr):
        self.info_stream("Reading attribute %s", attr.get_name())
        value = self._values[attr.get_name()]
        attr.set_value(value)

    def generic_write(self, attr):
        self.info_stream("Writing attribute %s - value %s", attr.get_name(), attr.get_write_value())
        self._values[attr.get_name()] = attr.get_write_value()
```

An approach more in line with the low-level API is also possible, but not recommended for new devices. The Device_3Impl::add_attribute() method has the following signature:

```
add_attribute(self, attr, r_meth=None, w_meth=None,
is_allo_meth=None)
```

attr is an instance of the *tango.Attr* class, r_meth is the method which has to be executed when 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.

Old example:

```
from tango import Attr, AttrWriteType
from tango.server import Device, command

class MyOldDevice(Device):

    @command(dtype_in=str)
    def CreateFloatAttribute(self, attr_name):
        attr = Attr(attr_name, tango.DevDouble, AttrWriteType.READ_WRITE)
        self.add_attribute(attr, self.read_General, self.write_General)

    def read_General(self, attr):
        self.info_stream("Reading attribute %s", attr.get_name())
        attr.set_value(99.99)
```

(continues on next page)

(continued from previous page)

```

    def write_General(self, attr):
        self.info_stream("Writing attribute %s - value %s", attr.get_
↪name(), attr.get_write_value())
```

6.19 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()`.

6.19.1 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 Device, command

class MyDevice(Device):

    @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.

6.19.2 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):

    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)

        # do the same for attributes...
        ...

    def Clone(self, dev_name):
        klass = self.get_device_class()
        klass.create_device(dev_name, alias=None, cb=self.fill_new_device_
↪properties)

    def DeleteSibling(self, dev_name):
        klass = self.get_device_class()
        klass.delete_device(dev_name)
```

Note that the cb parameter is optional. In the example it is given for demonstration purposes only.

6.20 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).

6.20.1 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:

```
if __name__ == '__main__':
    util = tango.Util(sys.argv)
    util.add_class(PyDsExpClass, PyDsExp)

    U = tango.Util.instance()
    U.server_init()
    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

6.20.2 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

```python
1 def __init__(self, name):
2     tango.DeviceClass.__init__(self, name)
3     self.set_type("TestDevice")
```

The device type is set at line 3.

6.20.3 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

6.20.4 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 enumeration. 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

6.20.5 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):

    def __init__(self,cl,name):
        tango.Device.__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
```

(continues on next page)

(continued from previous page)

```
25
26      def IOLong(self, in_data):
27          self.info_stream('IOLong', in_data)
28          in_data = in_data * 2
29          self.info_stream('IOLong returns', in_data)
30          return in_data
31
32      #------------------------------------------------------------------
33
34      def is_IOStringArray_allowed(self):
35          return self.get_state() == tango.DevState.ON
36
37      def IOStringArray(self, in_data):
38          l = range(len(in_data)-1, -1, -1)
39          out_index=0
40          out_data=[]
41          for i in l:
42              self.info_stream('IOStringArray <-', in_data[out_index])
43              out_data.append(in_data[i])
44              self.info_stream('IOStringArray ->',out_data[out_index])
45              out_index += 1
46          self.y = out_data
47          return out_data
48
49      #------------------------------------------------------------------
50      # ATTRIBUTES
51      #------------------------------------------------------------------
52
53      def read_attr_hardware(self, data):
54          self.info_stream('In read_attr_hardware')
55
56      def read_Long_attr(self, the_att):
57          self.info_stream("read_Long_attr")
58
59          the_att.set_value(self.attr_long)
60
61      def is_Long_attr_allowed(self, req_type):
62          return self.get_state() in (tango.DevState.ON,)
63
64      def read_Short_attr_rw(self, the_att):
65          self.info_stream("read_Short_attr_rw")
66
67          the_att.set_value(self.attr_short_rw)
68
69      def write_Short_attr_rw(self, the_att):
70          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 (this will used the latest device implementation class available, e.g. Device_5Impl)

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

SEVEN

HOW TO CONTRIBUTE

Everyone is welcome to contribute to PyTango project. If you don't feel comfortable with writing core PyTango we are looking for contributors to documentation or/and tests.

7.1 Workflow

A Git feature branch workflow is used. More details can be seen in this tutorial. Good practices:

- For commit messages the first line should be short (50 chars or less) and contain a summary of all changes. Provide more detail in additional paragraphs unless the change is trivial.
- Merge requests (MRs) should be ALWAYS made to the `develop` branch.

7.2 reStructuredText and Sphinx

Documentation is written in reStructuredText and built with Sphinx - it's easy to contribute. It also uses autodoc importing docstrings from tango package. Theme is not important, a theme prepared for Tango Community can be also used.

To test the docs locally requires Python >= 3.5:

- `$ python -m pip install six numpy sphinx sphinx_rtd_theme`
- `$ python -m sphinx doc build/sphinx`

To test the docs locally in a Sphinx Docker container:

- `(host) $ cd /path/to/pytango`
- `(host) $ docker run --rm -ti -v $PWD:/docs sphinxdoc/sphinx bash`
- `(container) $ python -m pip install six numpy sphinx_rtd_theme`
- `(container) $ python -m sphinx doc build/sphinx`

After building, open the `build/doc/index.html` page in your browser.

7.3 Source code standard

All code should be PEP8 compatible. We have set up checking code quality with Codacy which uses PyLint under the hood. You can see how well your code is rated on your PR's page.

Note: The accepted policy is that your code **cannot** introduce more issues than it solves!

You can also use other tools for checking PEP8 compliance for your personal use. One good example of such a tool is Flake8 which combines PEP8 and PyFlakes. There are plugins for various IDEs so that you can use your favourite tool easily.

7.4 Using Docker for development

Docker containers are useful for developing, testing and debugging PyTango. See the folder `.devcontainer` in the root of the source repo. It includes instructions for building the Docker images and using them for development.

For direct usage, rather than PyTango developement, a Docker image with PyTango already installed is available: https://hub.docker.com/r/tangocs/tango-pytango.

7.5 Releasing a new version

From time to time a new version should be released. Anyone who wishes to see some features of the development branch released is free to make a new release. The basic steps required are as follows:

Pick a version number

- Semantic version numbering is used: <major>.<minor>.<patch>
- The major and minor version fields (e.g., 9.3) track the TANGO C++ core version.
- Small changes are done as patch releases. For these the version number should correspond the current development number since each release process finishes with a version bump.
- **Patch release example:**
  - `9.3.4.devN` or `9.3.4aN` or `9.3.4bN` (current development branch)
  - changes to `9.3.4` (the actual release)
  - changes to `9.3.5.dev0` (bump the patch version at the end of the release process)
- **Minor release example:**
  - `9.3.4.devN` or `9.3.4aN` or `9.3.4bN` (current development branch)
  - changes to `9.4.0` (the actual release)
  - changes to `9.4.1.dev0` (bump the patch version at the end of the release process)

Create an issue in GitLab

- This is to inform the community that a release is planned.
- Use a checklist similar to the one below:

  Task list:
  - [ ] Read steps in the how-to-contribute docs for making a release

- [ ] Merge request to update changelog and bump version
- [ ] Merge MR (this is the last MR for the release)
- [ ] Make sure CI is OK on develop branch
- [ ] Make sure the documentation is updated for develop (readthedocs)
- [ ] Create an annotated tag from develop branch
- [ ] Make sure the documentation is updated for release (readthedocs)
- [ ] Upload the new version to PyPI
- [ ] Bump the version with "-dev" in the develop branch
- [ ] Create and fill in the release description on GitLab
- [ ] Build conda packages
- [ ] Advertise the release on the mailing list
- [ ] Close this issue

- A check list in this form on GitLab can be ticked off as the work progresses.

Make a branch from `develop` to prepare the release

- Example branch name: `prepare-v9.3.4`.
- Edit the changelog (in `docs/revision.rst`). Include *all* merge requests since the version was bumped after the previous release. Reverted merge requests can be omitted.
- Bump the versions (`tango/release.py` and `appveyor.yml`). E.g. `version_info = (9, 3, 4)`, and `version: 9.3.4.{build}`
- Create a merge request to get these changes reviewed and merged before proceeding.

Make sure CI is OK on `develop` branch

- On Gitlab CI and AppVeyor, all tests, on all versions of Python must be passing. If not, bad luck - you'll have to fix it first, and go back a few steps...

Make sure the documentation is updated

- Log in to https://readthedocs.org.
- Get account permissions for https://readthedocs.org/projects/pytango from another contributor, if necessary.
- **Readthedocs *should* automatically build the docs for each:**
  - push to develop (latest docs)
  - new tags (e.g v9.3.4)
- ***But*, the webhooks are somehow broken, so it probably won't work automatically!**
  - Trigger the builds manually here: https://readthedocs.org/projects/pytango/builds/
  - Set the new version to "active" here: https://readthedocs.org/dashboard/pytango/versions/

Create an annotated tag for the release

- GitLab's can be used to create the tag, but a message must be included. We don't want lightweight tags.
- **Alternatively, create tag from the command line (e.g., for version 9.3.4):**
  - `$ git checkout develop`
  - `$ git pull`
  - `$ git tag -a -m "tag v9.3.4" v9.3.4`
  - `$ git push -v origin refs/tags/v9.3.4`

Upload the new version to PyPI

- The source tarball is automatically uploaded to PyPI by Gitlab CI on tag. Any other wheels must be uploaded manually.
- Log in to https://pypi.org.
- Get account permissions for PyTango from another contributor, if necessary.
- If necessary, pip install twine: https://pypi.org/project/twine/)
- On AppVeyor find the build for the tag, download artifacts, and upload wheels. E.g., for version 9.3.4:
  - `$ twine upload dist/pytango-9.3.4-cp27-cp27m-win32.whl`
  - `$ twine upload dist/pytango-9.3.4-cp27-cp27m-win_amd64.whl`
  - `$ twine upload dist/pytango-9.3.4-cp36-cp36m-win32.whl`
  - `$ twine upload dist/pytango-9.3.4-cp36-cp36m-win_amd64.whl`
  - `$ twine upload dist/pytango-9.3.4-cp37-cp37m-win32.whl`
  - `$ twine upload dist/pytango-9.3.4-cp37-cp37m-win_amd64.whl`
  - `$ twine upload dist/pytango-9.3.4-cp38-cp38m-win32.whl`
  - `$ twine upload dist/pytango-9.3.4-cp38-cp38m-win_amd64.whl`

Bump the version with "-dev" in the develop branch

- Make a branch like `bump-dev-version` from head of `develop`.
- In `tango/release.py`, change `version_info`, e.g. from `(9, 3, 4)` to `(9, 3, 5, 'dev', 0)`.
- In `appveyor.yml`, change `version`, e.g. from `9.3.4.{build}` to `9.3.5.dev0.{build}`.
- Create MR, merge to `develop`.

Create and fill in the release description on GitLab

- Go to the Tags page: https://gitlab.com/tango-controls/pytango/-/tags
- Find the tag created above and click "Edit release notes".
- Content must be the same as the details in the changelog. List all the merge requests since the previous version.

Build conda packages

- Conda-forge is used to build these. See https://github.com/conda-forge/pytango-feedstock
- A new pull request should be created automatically by the Conda forge bot after our tag.
- Get it merged by one of the maintainers.

Advertise the release on the mailing list

- Post on the Python development list.
- Example of a previous post: http://www.tango-controls.org/community/forum/c/development/python/pytango-921-release/

Close off release issue

- All the items on the check list should be ticked off by now.
- Close the issue.

CHAPTER

EIGHT

TESTING PYTANGO DEVICES

Provide a context to run a device without a database.

8.1 Approaches to testing Tango devices

8.1.1 Overview

The follow sections detail different approaches that can be used when automating tests. This includes starting the real devices as normal in a Tango facility, using the `DeviceTestContext` for a more lightweight test, a hybrid approach mixing `DeviceTestContext` and real Tango devices in a Tango facility, and starting multiple devices with the `DeviceTestContext` and `MultiDeviceTestContext`.

Testing a single device without DeviceTestContext

Note: This approach is not recommended for unit testing.

Testing without a `DeviceTestContext` requires a complete Tango environment to be running (this environment is orchestrated by Makefiles and Docker containers in our Tango Example repo). That is, the following four components/processes should be present and configured:

- DSConfig tool
- Tango Databaseds Server
- MySQL/MariaDB
- Tango Device Server (with Tango device under test inside it)

In order to successfully constitute a working Tango environment, the following sequence of operations is required:

1. A running MySQL/MariaDB service.
2. The Tango Databaseds Server configured to connect to the database.
3. The DSConfig tool can be run to bootstrap the database configuration of the Tango Device based on configuration from a file.
4. The Tango Device Server that has been initialised and running the Tango Device.
5. In the test, you can instantiate a PyTango DeviceProxy object to interact with the Tango device under test.

This is a lot of infrastructure and complicated to orchestrate - it is not conducive to lightweight, fast running unit tests. Thus it is not recommended.

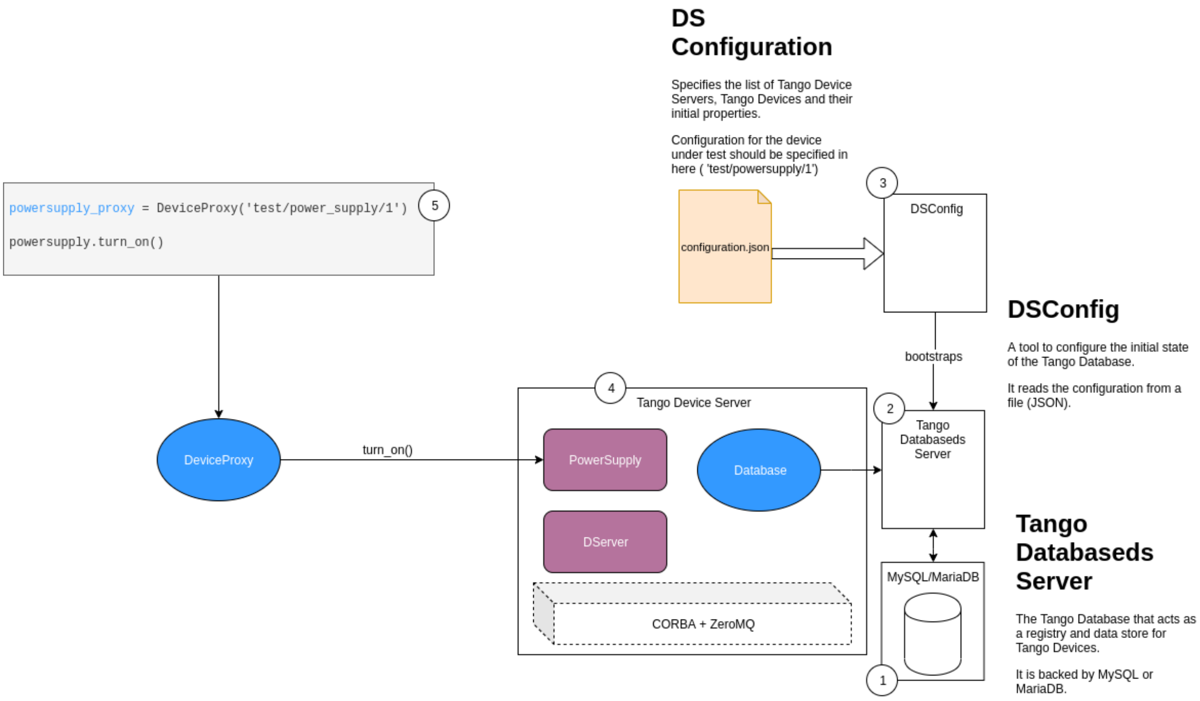


Figure 1. A schematic diagram showing the agents involved when testing a Tango device using the real Tango database and their interactions.

Examples:

- test_2_test_server_using_client.py
- test_3_test_motor_server.py
- test_4_test_event_receiver_server.py

Testing a single device with DeviceTestContext

A utility class is provided by PyTango that aids in testing Tango Devices. It automates a lot of the operations required to start up a Tango runtime environment.:

```
from tango.test_context import DeviceTestContext
```

The *DeviceTestContext* accepts a Tango Device Python class, as an argument, that will be under test (PowerSupply). It also accepts some additional arguments such as properties - see the method signature of *DeviceTestContext* constructor. It will then do the following: Generate stubbed data file that has the minimum configuration details for a Tango Device Server to initialise the Tango Device under test (PowerSupply). It will start the Tango Device Server that contains the Tango Device (in a separate thread by default, but optionally in a subprocess). DServer is a "meta" Tango Device that provides an administrative interface to control all the devices in the Tango Device Server process. The DeviceProxy object can be retrieved from the DeviceContext and can be invoked to interact with Tango Device under test. A DeviceProxy object will expose all the attributes and commands specified for the Tango Device as Python objects, but invoking them will communicate with the real device via CORBA. If events are used, these are transported via ZeroMQ.

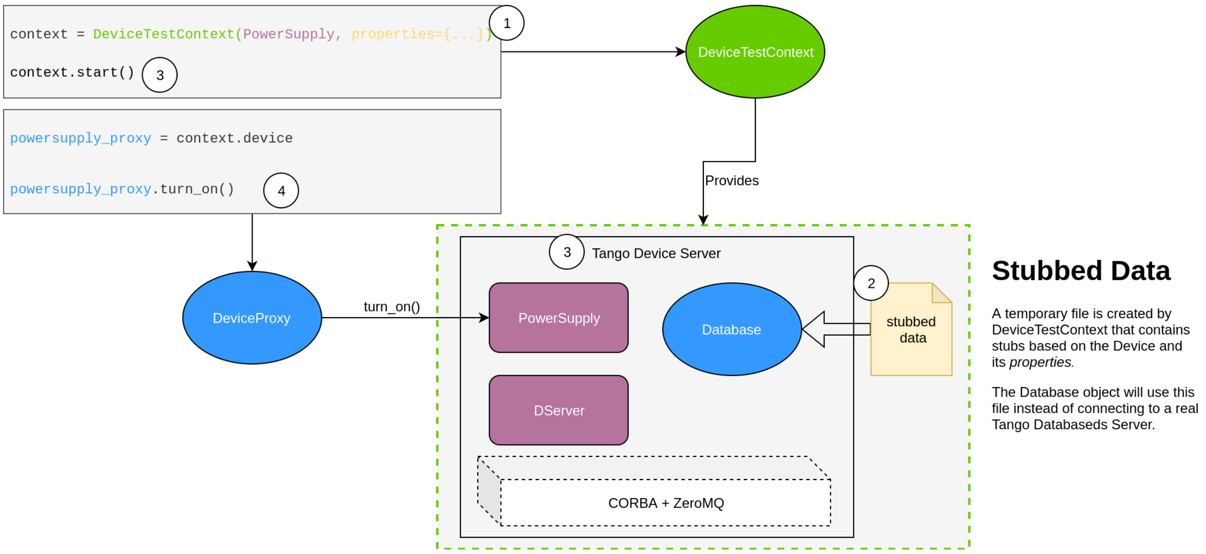


Figure 2. A schematic diagram showing the agents involved when testing a Tango device using the `DeviceTestContext` and their interactions.

You may now proceed to exercise the Tango Device's interface by invoking the appropriate methods/properties on the proxy:

Example Code Snippet	Tango Concept	Description
`powersupply_proxy.turn_on()`	Tango Command	An action that the Tango Device performs.
`powersupply_proxy.voltage`	Tango Attribute	A value that the Tango Device exposes.

Example:

- test_1_server_in_devicetestcontext.py

Testing a single device with DeviceTestContext combined with a real device(s) using the Tango database

This use case first requires the whole test infrastructure described in use case 1 above to be up before the tests can be run against the device (DishLeafNode) in the `DeviceTestContext`. The following sequence of events occur to run rests against the device (DishLeafNode):

- Set up the test infrastructure for the real device - DishMaster (all the steps defined for use case 1 above apply).
- Set up the test infrastructure for the device (DishLeafNode) in the `DeviceTestContext` (all steps in use case 2 above apply).
- Create a proxy (dish_proxy) which exposes the attributes and commands of the real device to be tested.
  - There's a proxy in the provisioned `DeviceTestContext` which knows about the real device but cannot expose its attributes and commands in that context, hence the need for the dish_proxy.

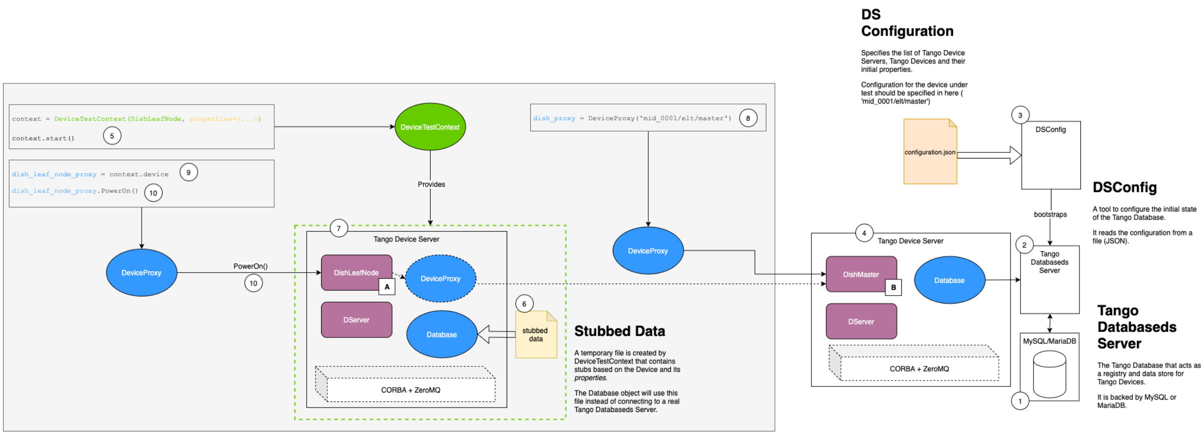


Figure 3. A schematic diagram showing the agents involved when testing multiple Tango devices using the `DeviceTestContext` together with the real Tango database and their interactions.

Examples:

- DishLeafNode/conftest.py

Testing with multiple DeviceTestContexts

Note: This approach is not recommended - rather use `MultiDeviceTestContext`.

The testing scenario depicted in Figure 3 can be implemented without using the real Tango database. In this use case, the underlying device (DishMaster) is provisioned using the `DeviceTestContext`. Just like in the use case above, another proxy (dish_proxy) is created to expose the commands and attributes of the DishMaster Device. The sequence of events which take place to provision each of these DeviceTestContexts are exactly the same as described in use case 1. This is not recommended because it can be done more easily using the `MultiDeviceTestContext`, as shown in the next section.

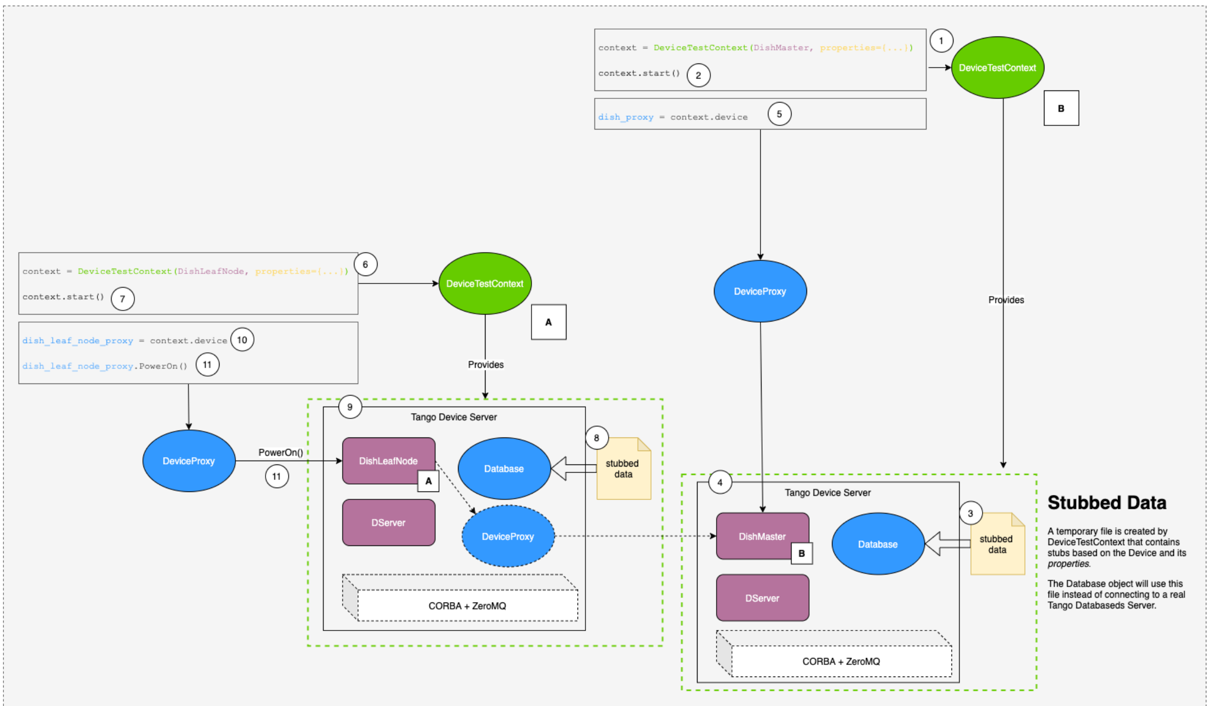


Figure 4. A schematic diagram showing the agents involved when testing multiple Tango devices using the `DeviceTestContext` and their interactions.

Examples:

- Tango forum post

Testing with MultiDeviceTestContext

There is another testing class available in PyTango: *MultiDeviceTestContext*, which helps to simplify testing of multiple devices. In this case the multiple devices are all launched in a single device server.:

```
from tango.test_context import MultiDeviceTestContext
```

The testing scenario depicted in Figure 4 can be implemented with just a single *MultiDeviceTestContext* instead of two *DeviceTestContext* instances (and still without using the real Tango database). In this use case, both devices (DishMaster and DishLeafNode) are provisioned using the *MultiDeviceTestContext*. Just like in the use case above, another proxy (dish_proxy) is created to expose the commands and attributes of the DishMaster Device to the test runner. The sequence of events which take place to provision this MultiDeviceTestContexts is similar that use case 1. The main difference is the devices_info the must be specified beforehand. Here we can define the devices that must be started, their names, and initial properties.

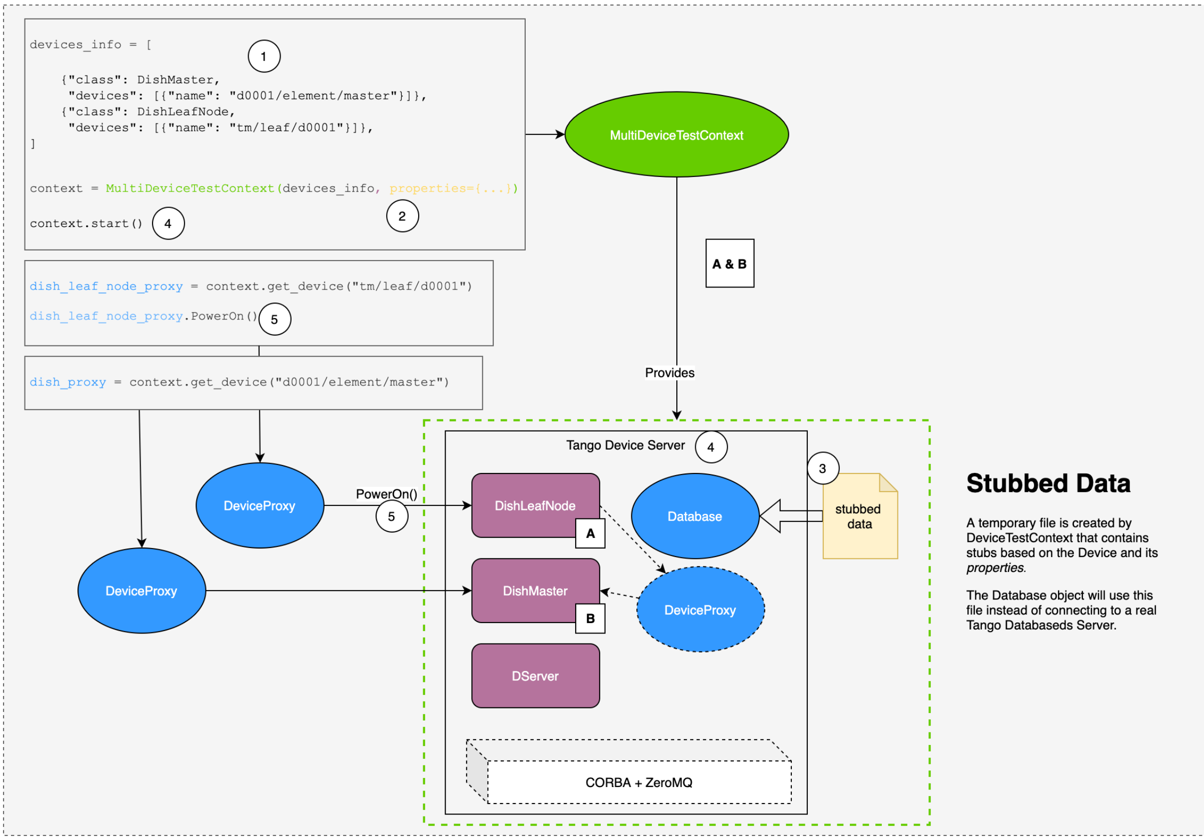


Figure 5. A schematic diagram showing the agents involved when testing multiple Tango devices using the *MultiDeviceTestContext* and their interactions.

Examples:

- MultiDeviceTestContext with fixtures

8.1.2 Issues

1. The `DeviceTestContext` is quite a heavyweight utility class in terms of the dependent components it needs to orchestrate so that testing can be done. It requires the Tango runtime, including ZeroMQ for events, and a Database stub file as a minimum.
2. A single process that uses a `DeviceTestContext` more than once in threaded mode (so not a subprocess, with kwarg `process=True`), will get a segmentation fault. With `process=True`, we cannot access the internals of the class under test from the test runner - we have to use the Tango API via a DeviceProxy.
3. Forwarded attributes do not work.
4. There is no way to unit test (in the strict definition), since the Tango device objects cannot be directly instantiated.

8.1.3 Acknowledgement

Initial content for this page contributed by the Square Kilometre Array.

8.2 Device Test Context Classes API

The API of the testing classes are described here. For an overview of their behaviour, see *Approaches to testing Tango devices*.

8.2.1 DeviceTestContext

class `tango.test_context.`**`DeviceTestContext`**(*device, device_cls=None, server_name=None, instance_name=None, device_name=None, properties=None, db=None, host=None, port=0, debug=3, process=False, daemon=False, timeout=None, memorized=None, green_mode=None*)

Bases: `MultiDeviceTestContext`

Context to run a single device without a database.

The difference with respect to `MultiDeviceTestContext` is that it only allows to export a single device.

Example usage:

```
from time import sleep

from tango.server import Device, attribute, command
from tango.test_context import DeviceTestContext

class PowerSupply(Device):

    @attribute(dtype=float)
    def voltage(self):
        return 1.23

    @command
    def calibrate(self):
        sleep(0.1)
```

(continues on next page)

(continued from previous page)

```python

def test_calibrate():
    '''Test device calibration and voltage reading.'''
    with DeviceTestContext(PowerSupply, process=True) as proxy:
        proxy.calibrate()
        assert proxy.voltage == 1.23
```

Parameters

- **device** (*Device* or DeviceImpl) – Device class to be run.
- **device_cls** – The device class can be provided if using the low-level API. Optional. Not required for high-level API devices, of type *Device*.

append_db_file (*server, instance, tangoclass, device_prop_info*)

Generate a database file corresponding to the given arguments.

delete_db ()

delete temporary database file only if it was created by this class

get_device (*device_name*)

Return the device proxy corresponding to the given device name.

Maintains previously accessed device proxies in a cache to not recreate then on every access.

get_device_access (*device_name=None*)

Return the full device name.

get_server_access ()

Return the full server name.

start ()

Run the server.

stop ()

Kill the server.

8.2.2 MultiDeviceTestContext

class tango.test_context.**MultiDeviceTestContext** (*devices_info, server_name=None, instance_name=None, db=None, host=None, port=0, debug=3, process=False, daemon=False, timeout=None, green_mode=None*)

Bases: object

Context to run device(s) without a database.

The difference with respect to *DeviceTestContext* is that it allows to export multiple devices (even of different Tango classes).

Example usage:

```python
from tango.server import Device, attribute
from tango.test_context import MultiDeviceTestContext


class Device1(Device):

    @attribute
```

(continues on next page)

(continued from previous page)

```python
    def attr1(self):
        return 1.0


class Device2(Device):

    @attribute
    def read_attr2(self):
        return 2.0


devices_info = (
    {
        "class": Device1,
        "devices": [
            {
                "name": "test/device1/1"
            },
        ]
    },
    {
        "class": Device2,
        "devices": [
            {
                "name": "test/device2/1",
            },
        ]
    }
)

def test_devices():
    with MultiDeviceTestContext(devices_info, process=True) as context:
        proxy1 = context.get_device("test/device1/1")
        proxy2 = context.get_device("test/device2/1")
        assert proxy1.attr1 == 1.0
        assert proxy2.attr2 == 2.0
```

Parameters

- **devices_info** (*sequence<dict>*) – a sequence of dicts with information about devices to be exported. Each dict consists of the following keys:
  - ”class” which value is either of:
    - : class:~*tango.server.Device* or the name of some such class
    - a sequence of two elements, the first element being a *DeviceClass* or the name of some such class, the second element being a `DeviceImpl` or the name of such such class
  - ”devices” which value is a sequence of dicts with the following keys:
    - ”name” (str)
    - ”properties” (dict)
    - ”memorized” (dict)
- **server_name** (`str`) – Name to use for the device server. Optional. Default is the first device’s class name.

- **instance_name** (str) – Name to use for the device server instance. Optional. Default is lower-case version of the server name.
- **db** (str) – Path to a pre-populated text file to use for the database. Optional. Default is to create a new temporary file and populate it based on the devices and properties supplied in *devices_info*.
- **host** (str) – Hostname to use for device server's ORB endpoint. Optional. Default is a local IP address.
- **port** (int) – Port number to use for the device server's ORB endpoint. Optional. Default is chosen by omniORB.
- **debug** (int) – Debug level for the device server logging. 0=OFF, 1=FATAL, 2=ERROR, 3=WARN, 4=INFO, 5=DEBUG. Optional. Default is warn.
- **process** (bool) – True if the device server should be launched in a new process, otherwise use a new thread. Note: if the context will be used mutiple times, it may seg fault if the thread mode is chosen. Optional. Default is thread.
- **daemon** (bool) – True if the new thread/process must be created in daemon mode. Optional. Default is not daemon.
- **timeout** (float) – How long to wait (seconds) for the device server to start up, and also how long to wait on joining the thread/process when stopping. Optional. Default differs for thread and process modes.
- **green_mode** (*GreenMode*) – Green mode to use for the device server. Optional. Default uses the Device specification (via green_mode class attribute), or if that isn't specified the global green mode.

append_db_file(*server*, *instance*, *tangoclass*, *device_prop_info*)

Generate a database file corresponding to the given arguments.

delete_db()

delete temporary database file only if it was created by this class

get_device(*device_name*)

Return the device proxy corresponding to the given device name.

Maintains previously accessed device proxies in a cache to not recreate then on every access.

get_device_access(*device_name*)

Return the full device name.

get_server_access()

Return the full server name.

start()

Run the server.

stop()

Kill the server.

8.3 Mocking clients for Testing

8.3.1 Test Doubles: The idea behind mocking Tango entities

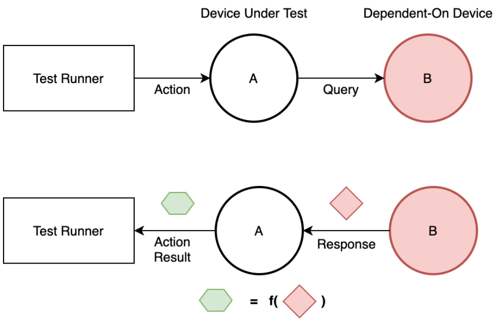
Suppose we want to test a Tango Device, **Device A**. In particular, we want to assert that when a certain *action* is invoked on **Device A**, it should produce an expected result. This will prove to us that **Device A** 's implementation sufficiently manifests the behaviour we would like it to have.

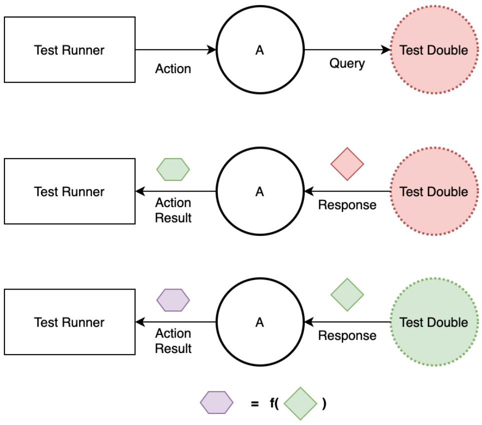
Now suppose **Device A** depends on **Device B** to complete its action. In other words, the *result* will depend, in part, on the state of **Device B**. This means that to test this scenario, both devices need to be in a base state that we control.

This might be difficult to achieve when using real devices since it might require a lot of orchestration and manipulation of details irrelevant to the test scenario, i.e. to get **Device B** into the required state.

A **Test Double** is a component that can act as a real device but is easier to manipulate and configure into the states that we want during testing. This means that we can replace **Device B** with its **Test Double** as long as it conforms to the interface that **Device A** expects.

What's more, we can manipulate the **Test Double** to respond in the way we expect **Device B** to respond under the various conditions we want to test. A **Mock** is simply a type of **Test Double** that might have some conditional logic or behaviour to help in testing.





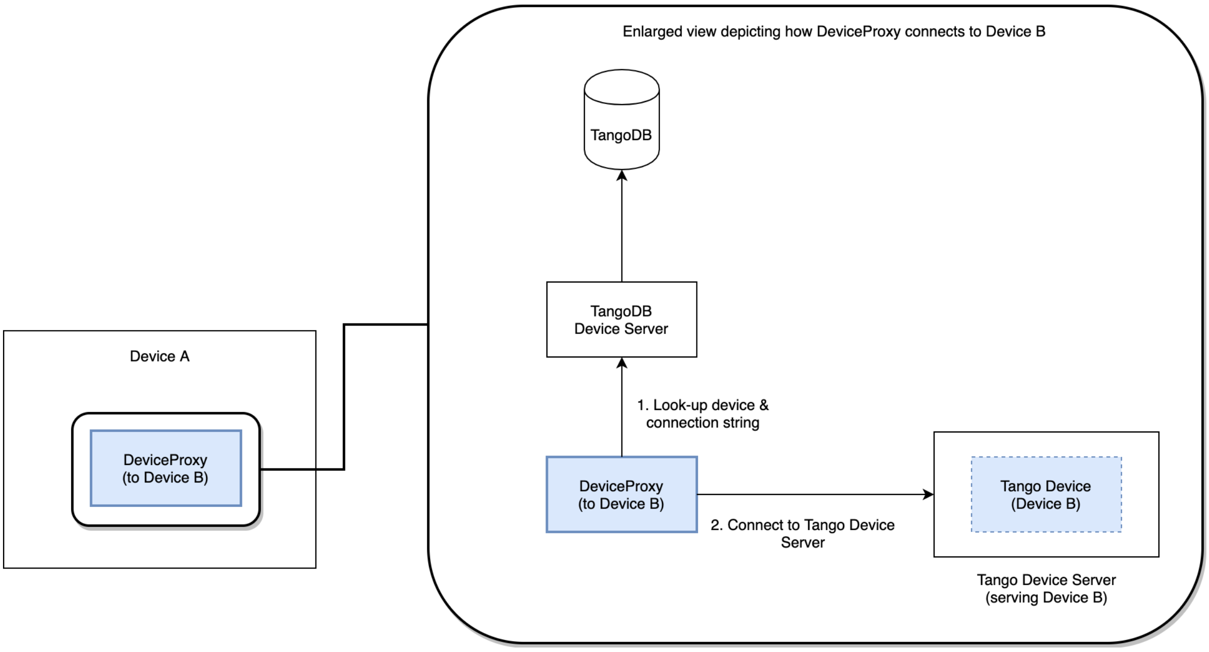
8.3.2 Tango's DeviceProxys

In Tango, the **DeviceProxy** is an object that allows communication between devices. It can be regarded as the *client* part of a *client-server* interaction.

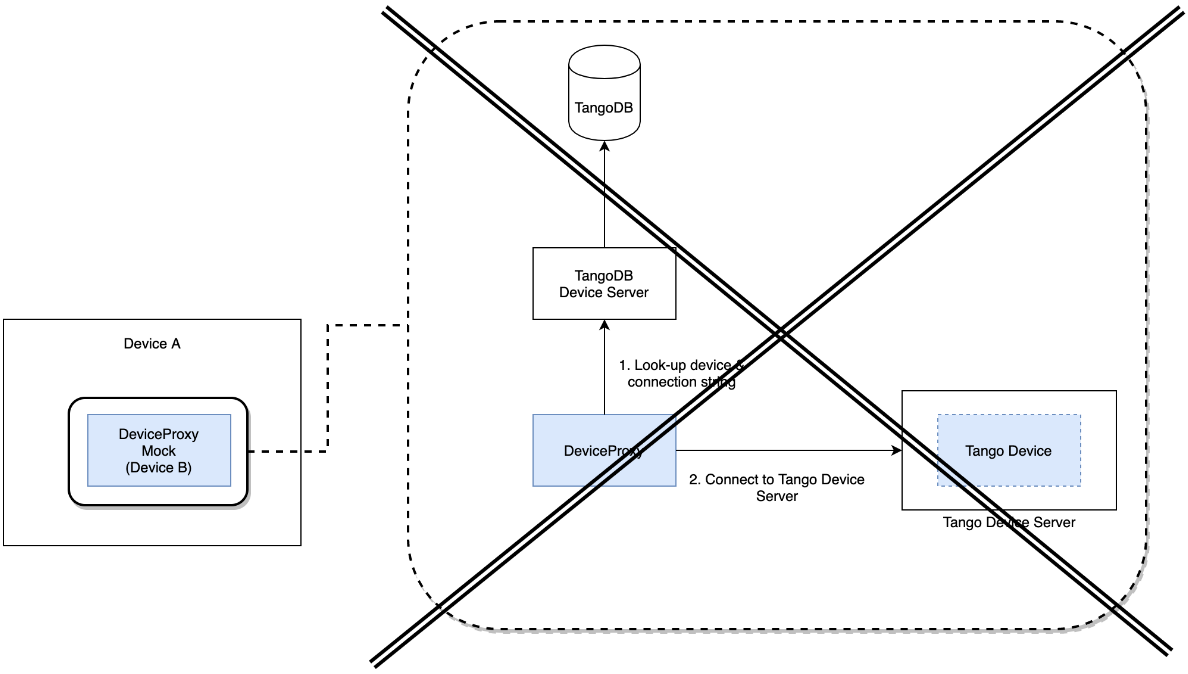
Thus, any Tango device (say, **Device A**) that depends on a secondary Tango device (**Device B**) will need to use a **DeviceProxy** to connect and communicate with the secondary device (**Device B**).

This invariably means that in the implementation of **Device A**, it will be instantiating and using a **DeviceProxy** object.

However, the mechanism by which this happens is a multi-step process which requires communication with a TangoDB DS and an appropriately configured Tango system that contains a deployed **Device B**.



If we can replace the **DeviceProxy** object that **Device A** will use to communicate to **Device B** with our own Mock object (**DeviceProxyMock**), we can test the behaviour of **Device A** while faking the responses it expects to receive from querying **Device B**. All this without the need for deploying a real **Device B**, since for all intents and purposes, the **DeviceProxyMock** would represent the real device.



In other words, mocking the **DeviceProxy** is sufficient to mock the underlying device it connects to, with reference to how **DeviceProxy** is used by **Device A**.

8.3.3 Mocking the DeviceProxy

In some **PyTango** devices, such as those in the SKA TMC Prototype, the **DeviceProxy** object is instantiated during the operation of the **Device Under Test** (**DUT**). Also, there isn't usually an explicit interface to inject a **DeviceProxyMock** as a replacement for the real class.

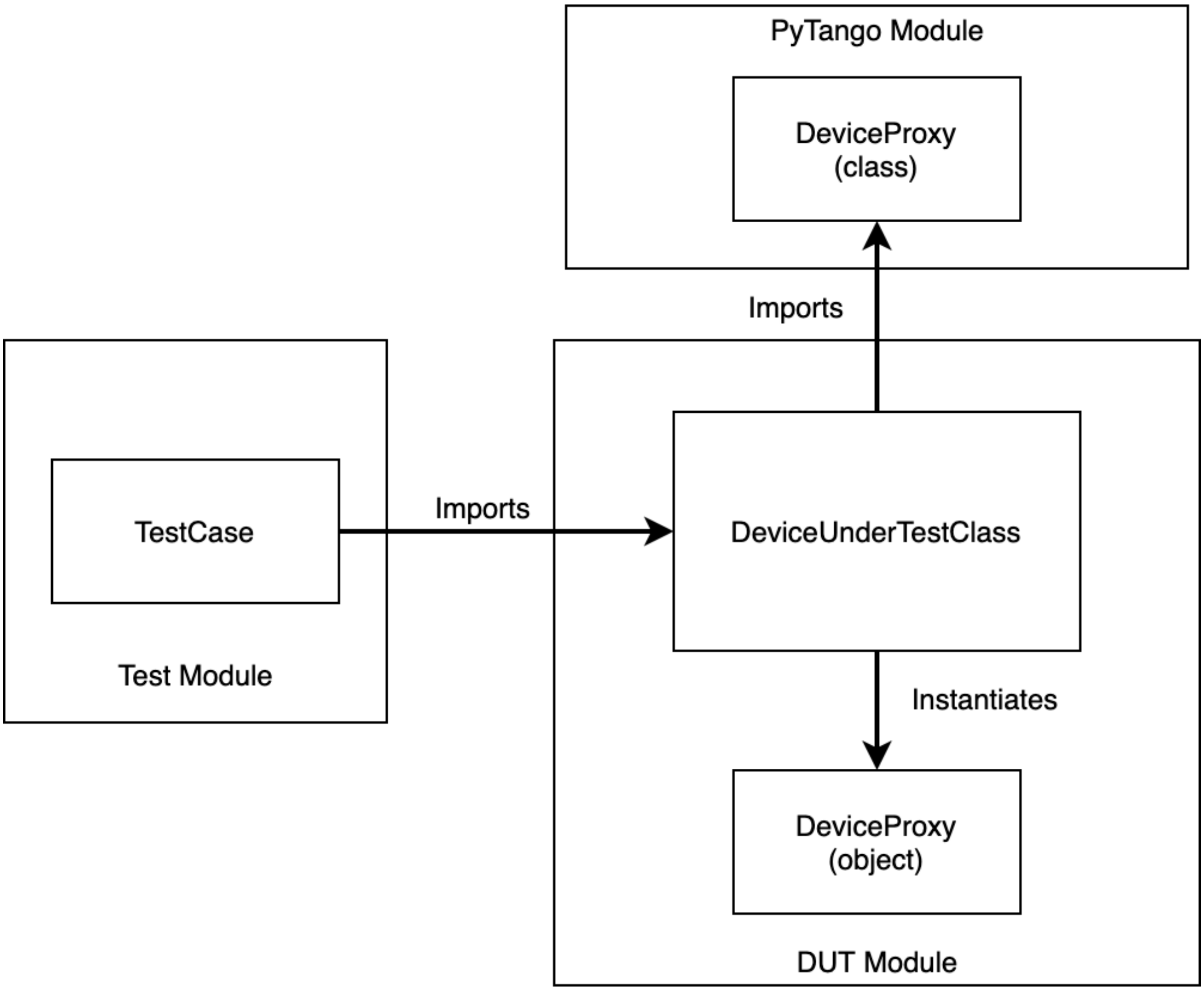
As an example, the CentralNode (at v0.1.8) device from the TMC Prototype instantiates all the **DeviceProxy** objects it needs to connect to its child elements/components in its init_device method:

```
class CentralNode(SKABaseDevice):
...
  def init_device(self):
     ...
     self._leaf_device_proxy.append(DeviceProxy(self._dish_leaf_node_
↪devices[name]))
     ...
     self._csp_master_leaf_proxy = DeviceProxy(self.CspMasterLeafNodeFQDN)
     ...
     self._sdp_master_leaf_proxy = DeviceProxy(self.SdpMasterLeafNodeFQDN)
     ...
     subarray_proxy = DeviceProxy(self.TMMidSubarrayNodes[subarray])
```

Unfortunately, the *init_device* method does not allow us to provide the class we want the device to use when it needs to instantiate its **DeviceProxys**.

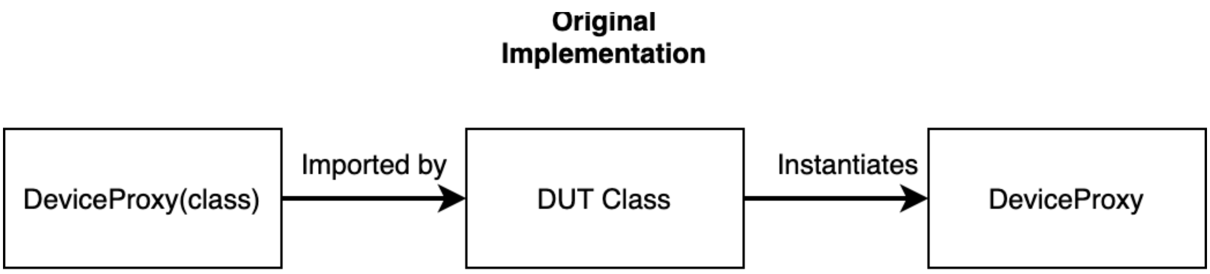
So we will have to mock the **DeviceProxy** class that the **DUT** imports before it instantiates that class.

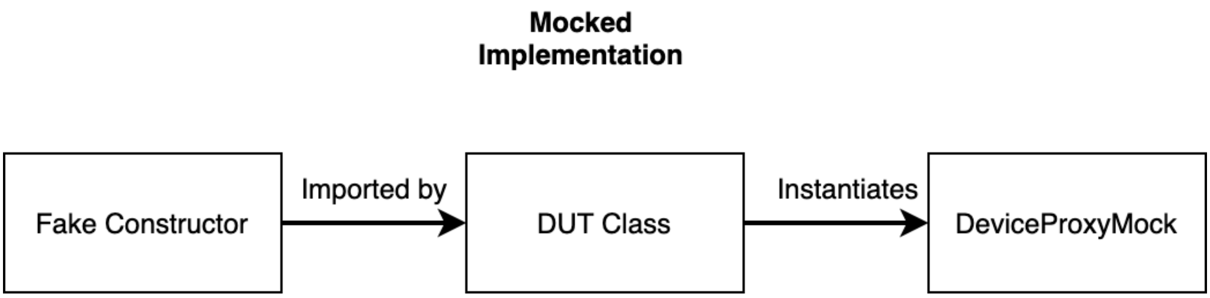
The diagram below illustrates the relationship between the TestCase, **DUT** and its transitive import of the **DeviceProxy** class from the **PyTango** module:



So, we want to replace the imported **DeviceProxy** class with our own Fake Constructor that will provide a Mocked Device Proxy for the **DUT** during tests.

In other words, we want to replace the thing that instantiates the **DeviceProxy** (i.e. the constructor) with our own callable object that constructs a mocked **DeviceProxy** object instead of the real one. We want to move from the original implementation to the mocked implementation shown in the diagram below:





8.3.4 Solution

This can be achieved by using the unittest.mock library that comes with Python 3 or the third-party library mock for Python 2.7.

The *mock.patch()* method allows us to temporarily change the object that a name points to with another one.

We use this mechanism to replace the **DeviceProxy** class (constructor) with our own fake constructor (a mock) that returns a Mock object:

```
with mock.patch(device_proxy_import_path) as patched_constructor:
    patched_constructor.side_effect = lambda device_fqdn: proxies_to_mock.get(device_fqdn, Mock())
    patched_module = importlib.reload(sys.modules[device_under_test.__module__])
```

An alternative to using *mock.patch* is pytest's *monkeypatch*. Its *.setattr* method provides the same functionality, i.e. allowing you to intercept what an object call would normally do and substituting its full execution with your own specification. There are more examples of its use in the OET implementation which is discussed below.

proxies_to_mock is a dictionary that maps **DeviceProxyMock** objects to their associated Tango device addresses that we expect the **DUT** to use when instantiating **DeviceProxy** objects. A brand new generic *Mock()* is returned if a specific mapping isn't provided.

Since the **DeviceProxy** class is defined at import time, we will need to reload the module that holds the **DUT**. This is why we explicitly call *importlib.reload(...)* in the context of *mock.patch()*.

For full details and code that implement this solution, see the following merge requests:

- https://gitlab.com/ska-telescope/tmc-prototype/-/merge_requests/23
- https://gitlab.com/ska-telescope/tmc-prototype/-/merge_requests/35

8.3.5 Moving on

Once we mocked **DeviceProxy**, then we can use the constructor of this object to return a device that is fake. This can be:

- a stub device, programmed to behave in a way that suits the tests that we are writing; in this case we are using the stub to inject other inputs to the **DUT**, under control of the test case;
- a mock device, a stub device where we can inspect also how the **DUT** interacted with it, and we can write assertions.

The benefits that we can achieve with the technique described here are:

1. ability to test the **DUT** in isolation
2. ability to create tests that are very fast (no network, no databases)
3. ability to inject into the **DUT** indirect inputs
4. ability to observe the indirect outputs of the **DUT**
5. ability to observe the interactions that the **DUT** has with the mock.

8.3.6 Using pytest and fixtures

The above mocking techniques can be achieved in a very succint way using pytest fixtures. Examples of this can be found in the pytango/examples. And more examples are available in the last section of the *Unit testing Tango devices in Python* presentation from the Tango 2020 November status update meeting.

8.3.7 Acknowledgement

Initial content for this page contributed by the Square Kilometre Array.

CHAPTER

NINE

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 GitLab issues page.

How can I contribute to PyTango and the documentation?

Contribution are always welcome!

You can open pull requests on the GitLab PRs 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

TEN

PYTANGO ENHANCEMENT PROPOSALS

10.1 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

10.1.1 Abstract

This TEP aims to define a new high level API for writting device servers.

10.1.2 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,))
```

10.1.3 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:**
    1. calls super constructor
    2. debug message
    3. 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:**
    1. create *Util*
    2. register tango class
    3. 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")
    4. call *server_init()*
    5. call *server_run()*

10.1.4 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()
```

(continues on next page)

(continued from previous page)

```
    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)
```

10.1.5 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
```

(continues on next page)

(continued from previous page)

```
# 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 -----#
```

(continues on next page)

(continued from previous page)

```
        #----- 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 = {
        }
```

(continues on next page)

(continued from previous page)

```
    #     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__()")
```

(continues on next page)

(continued from previous page)

```
        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')
```

(continues on next page)

(continued from previous page)

```
        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()
```

10.1.6 References

tango.server

10.1.7 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

10.1.8 Copyright

This document has been placed in the public domain.

10.2 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

10.2.1 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).

10.2.2 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.

10.2.3 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

10.2.4 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

10.2.5 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.

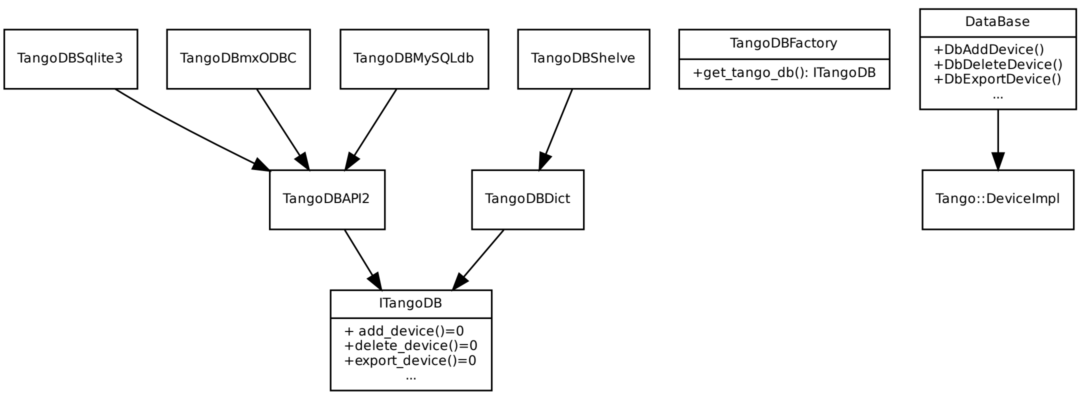
10.2.6 Step 3 - Architecture

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).



10.2.7 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).

10.2.8 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.

10.2.9 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-trunk
```

10.2.10 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).

10.2.11 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

ELEVEN

HISTORY OF CHANGES

Contributers
T. Coutinho

Last Update
Sep 29, 2022

11.1 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. Su
24/07/09	8.1	Added migration info, added missing API doc	T. Coutinho/R. Su
21/09/09	8.2	Added migration info, release of 7.0.0beta2	T. Coutinho/R. Su
12/11/09	8.3	Update to Tango 7.1.	T. Coutinho/R. Su
??/12/09	8.4	Update to PyTango 7.1.0 rc1	T. Coutinho/R. Su
19/02/10	8.5	Update to PyTango 7.1.1	T. Coutinho/R. Su
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

continues o

Table 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	9.2.0a	Update to PyTango 9.2.0a	T. Coutinho
15/08/16	9.2.0	9.2.0 Release	V. Michel
23/01/17	9.2.1	9.2.1 Release	V. Michel
27/09/17	9.2.2	9.2.2 Release	G. Cuni/V. Miche
30/05/18	9.2.3	9.2.3 Release	V. Michel
30/07/18	9.2.4	9.2.4 Release	V. Michel
28/11/18	9.2.5	9.2.5 Release	A. Joubert
13/03/19	9.3.0	9.3.0 Release	T. Coutinho
08/08/19	9.3.1	9.3.1 Release	A. Joubert
30/04/20	9.3.2	9.3.2 Release	A. Joubert
24/12/20	9.3.3	9.3.3 Release	A. Joubert
14/06/22	9.3.4	9.3.4 Release	A. Joubert
07/09/22	9.3.5	9.3.5 Release	Y. Matveev
28/09/22	9.3.6	9.3.6 Release	Y. Matveev

11.2 Version history

CHAPTER

TWELVE

INDEXES

- modindex
- genindex

Last update: Sep 29, 2022

PYTHON MODULE INDEX

t

tango, 21
tango.server, 109
tango.test_context, 269

INDEX

A

AccessControlType (*class in tango*), 105
add() (*tango.Group method*), 83
add_attribute() (*tango.LatestDeviceImpl method*), 134
add_attribute() (*tango.server.Device method*), 113
add_class() (*tango.Util method*), 175
add_command() (*tango.LatestDeviceImpl method*), 135
add_command() (*tango.server.Device method*), 113
add_Cpp_TgClass() (*tango.Util method*), 175
add_device() (*tango.Database method*), 185
add_logging_target() (*tango.DeviceProxy method*), 30
add_server() (*tango.Database method*), 185
add_TgClass() (*tango.Util method*), 175
adm_name() (*tango.DeviceProxy method*), 31
alias() (*tango.DeviceProxy method*), 31
always_executed_hook() (*tango.LatestDeviceImpl method*), 135
always_executed_hook() (*tango.server.Device method*), 114
ApiUtil (*class in tango*), 93
append_db_file() (*tango.test_context.DeviceTestContext method*), 275
append_db_file() (*tango.test_context.MultiDeviceTestContext method*), 277
append_status() (*tango.LatestDeviceImpl method*), 135
append_status() (*tango.server.Device method*), 114
ArchiveEventInfo (*class in tango*), 101
asyn_req_type (*class in tango*), 105
AsynCall, 236
AsynReplyNotArrived, 237
Attr (*class in tango*), 156
AttrConfEventData (*class in tango*), 102
AttrDataFormat (*class in tango*), 106
AttReqType (*class in tango*), 104
Attribute (*class in tango*), 160
attribute (*class in tango.server*), 127
attribute_history() (*tango.DeviceProxy method*), 31
attribute_list_query() (*tango.DeviceProxy method*), 31
attribute_list_query_ex() (*tango.DeviceProxy method*), 31
attribute_query() (*tango.DeviceProxy method*), 32
AttributeAlarmInfo (*class in tango*), 95
AttributeDimension (*class in tango*), 95
AttributeEventInfo (*class in tango*), 101
AttributeInfo (*class in tango*), 95
AttributeInfoEx (*class in tango*), 96
AttributeProxy (*class in tango*), 68
AttrQuality (*class in tango*), 105
AttrReadEvent (*class in tango*), 101
AttrWriteType (*class in tango*), 105
AttrWrittenEvent (*class in tango*), 101

B

black_box() (*tango.DeviceProxy method*), 32
build_connection() (*tango.Database method*), 186

C

cancel_all_polling_asynch_request() (*tango.DeviceProxy method*), 32
cancel_asynch_request() (*tango.DeviceProxy method*), 32
cb_sub_model (*class in tango*), 105
ChangeEventInfo (*class in tango*), 102
check_access_control() (*tango.Database method*), 187
check_alarm() (*tango.Attribute method*), 160
check_alarm() (*tango.MultiAttribute method*), 169
check_command_exists() (*tango.LatestDeviceImpl method*), 135
check_command_exists() (*tango.server.Device method*), 114
check_tango_host() (*tango.Database method*), 187
check_type() (*tango.Attr method*), 156
class_property (*class in tango.server*), 132
cleanup() (*tango.ApiUtil method*), 93
CmdArgType (*class in tango*), 103
CmdDoneEvent (*class in tango*), 101
command() (*in module tango.server*), 129

command_history() (tango.DeviceProxy method), 33
command_inout() (tango.DeviceProxy method), 33
command_inout() (tango.Group method), 84
command_inout_asynch() (tango.DeviceProxy method), 34
command_inout_asynch() (tango.Group method), 84
command_inout_raw() (tango.DeviceProxy method), 35
command_inout_reply() (tango.DeviceProxy method), 35
command_inout_reply() (tango.Group method), 85
command_inout_reply_raw() (tango.DeviceProxy method), 36
command_list_query() (tango.DeviceProxy method), 36
command_query() (tango.DeviceProxy method), 36
CommandInfo (class in tango), 97
CommunicationFailed, 235
connect() (tango.DeviceProxy method), 37
connect_db() (tango.Util method), 176
ConnectionFailed, 234
contains() (tango.Group method), 85
create_device() (tango.DeviceClass method), 149
create_device() (tango.Util method), 176

D

Database (class in tango), 185
DataReadyEventData (class in tango), 102
DbDatum (class in tango), 217
DbDevExportInfo (class in tango), 217
DbDevExportInfos (class in tango), 217
DbDevImportInfo (class in tango), 217
DbDevImportInfos (class in tango), 218
DbDevInfo (class in tango), 218
DbHistory (class in tango), 218
DbServerInfo (class in tango), 219
debug_stream() (tango.LatestDeviceImpl method), 136
debug_stream() (tango.server.Device method), 114
DebugIt (class in tango), 154
decode_gray16() (tango.EncodedAttribute method), 219
decode_gray8() (tango.EncodedAttribute method), 219
decode_rgb32() (tango.EncodedAttribute method), 220
delete_attribute_alias() (tango.Database method), 187
delete_class_attribute_property() (tango.Database method), 187
delete_class_pipe_property() (tango.Database method), 188
delete_class_property() (tango.Database method), 188
delete_db() (tango.test_context.DeviceTestContext method), 275
delete_db() (tango.test_context.MultiDeviceTestContext method), 277
delete_device() (tango.Database method), 189
delete_device() (tango.DeviceClass method), 149
delete_device() (tango.LatestDeviceImpl method), 136
delete_device() (tango.server.Device method), 114
delete_device() (tango.Util method), 176
delete_device_alias() (tango.Database method), 189
delete_device_attribute_property() (tango.Database method), 189
delete_device_pipe_property() (tango.Database method), 189
delete_device_property() (tango.Database method), 190
delete_property() (tango.AttributeProxy method), 68
delete_property() (tango.Database method), 190
delete_property() (tango.DeviceProxy method), 37
delete_server() (tango.Database method), 191
delete_server_info() (tango.Database method), 191
description() (tango.DeviceProxy method), 38
dev_name() (tango.DeviceProxy method), 38
dev_state() (tango.LatestDeviceImpl method), 136
dev_state() (tango.server.Device method), 114
dev_status() (tango.LatestDeviceImpl method), 136
dev_status() (tango.server.Device method), 115
DevCommandInfo (class in tango), 97
DevError (class in tango), 234
DevFailed, 234
Device (class in tango.server), 113
device_destroyer() (tango.DeviceClass method), 150
device_factory() (tango.DeviceClass method), 150
device_name_factory() (tango.DeviceClass method), 150
device_property (class in tango.server), 132
DeviceAttribute (class in tango), 98
DeviceAttribute.ExtractAs (class in tango), 98
DeviceAttributeConfig (class in tango), 96
DeviceAttributeHistory (class in tango), 103
DeviceClass (class in tango), 148
DeviceData (class in tango), 100
DeviceDataHistory (class in tango), 103
DeviceInfo (class in tango), 97

DeviceProxy (*class in tango*), 29
DeviceProxy() (*in module tango.futures*), 91
DeviceProxy() (*in module tango.gevent*), 92
DeviceTestContext (*class in tango.test_context*), 274
DeviceUnlocked, 237
DevSource (*class in tango*), 106
DevState (*class in tango*), 106
disable() (*tango.Group method*), 85
DispLevel (*class in tango*), 106
dyn_attr() (*tango.DeviceClass method*), 150

E

enable() (*tango.Group method*), 85
encode_gray16() (*tango.EncodedAttribute method*), 221
encode_gray8() (*tango.EncodedAttribute method*), 221
encode_jpeg_gray8() (*tango.EncodedAttribute method*), 222
encode_jpeg_rgb24() (*tango.EncodedAttribute method*), 223
encode_jpeg_rgb32() (*tango.EncodedAttribute method*), 224
encode_rgb24() (*tango.EncodedAttribute method*), 225
EncodedAttribute (*class in tango*), 219
environment variable
    PYTANGO_GREEN_MODE, 13
    TANGO_HOST, 3, 4, 239
error_stream() (*tango.LatestDeviceImpl method*), 137
error_stream() (*tango.server.Device method*), 115
ErrorIt (*class in tango*), 155
ErrSeverity (*class in tango*), 106
event_queue_size() (*tango.AttributeProxy method*), 69
event_queue_size() (*tango.DeviceProxy method*), 38
EventCallback (*class in tango.utils*), 226
EventData (*class in tango*), 102
EventSystemFailed, 237
EventType (*class in tango*), 105
Except (*class in tango*), 232
export_device() (*tango.Database method*), 191
export_device() (*tango.DeviceClass method*), 150
export_event() (*tango.Database method*), 191
export_server() (*tango.Database method*), 192
extract() (*tango.DeviceData method*), 100

F

fatal_stream() (*tango.LatestDeviceImpl method*), 137
fatal_stream() (*tango.server.Device method*), 115
FatalIt (*class in tango*), 156
fromdatetime() (*tango.TimeVal static method*), 107
fromtimestamp() (*tango.TimeVal static method*), 107

G

get_access_control() (*tango.DeviceProxy method*), 38
get_access_except_errors() (*tango.Database method*), 192
get_access_right() (*tango.DeviceProxy method*), 38
get_alias() (*tango.Database method*), 192
get_alias_from_attribute() (*tango.Database method*), 192
get_alias_from_device() (*tango.Database method*), 193
get_assoc() (*tango.Attr method*), 156
get_assoc_ind() (*tango.Attribute method*), 161
get_assoc_name() (*tango.Attribute method*), 161
get_asynch_cb_sub_model() (*tango.ApiUtil method*), 93
get_asynch_replies() (*tango.ApiUtil method*), 93
get_asynch_replies() (*tango.DeviceProxy method*), 39
get_attr_by_ind() (*tango.MultiAttribute method*), 170
get_attr_by_name() (*tango.MultiAttribute method*), 170
get_attr_ind_by_name() (*tango.MultiAttribute method*), 170
get_attr_min_poll_period() (*tango.LatestDeviceImpl method*), 137
get_attr_min_poll_period() (*tango.server.Device method*), 115
get_attr_nb() (*tango.MultiAttribute method*), 170
get_attr_poll_ring_depth() (*tango.LatestDeviceImpl method*), 137
get_attr_poll_ring_depth() (*tango.server.Device method*), 116
get_attr_serial_model() (*tango.Attribute method*), 161
get_attribute_alias() (*tango.Database method*), 193
get_attribute_alias_list() (*tango.Database method*), 193
get_attribute_config() (*tango.DeviceProxy method*), 39
get_attribute_config() (*tango.LatestDeviceImpl method*), 137
get_attribute_config() (*tango.server.Device method*), 116
get_attribute_config_2() (*tango.LatestDeviceImpl method*), 137
get_attribute_config_2() (*tango.server.Device method*), 116

get_attribute_config_3() (tango.LatestDeviceImpl method), 138
get_attribute_config_3() (tango.server.Device method), 116
get_attribute_config_ex() (tango.DeviceProxy method), 40
get_attribute_from_alias() (tango.Database method), 194
get_attribute_list() (tango.DeviceProxy method), 40
get_attribute_list() (tango.MultiAttribute method), 170
get_attribute_name() (tango.DbHistory method), 218
get_attribute_poll_period() (tango.DeviceProxy method), 40
get_attribute_poll_period() (tango.LatestDeviceImpl method), 138
get_attribute_poll_period() (tango.server.Device method), 116
get_cl_name() (tango.Attr method), 156
get_class_attr() (tango.DeviceClass method), 150
get_class_attribute_list() (tango.Database method), 194
get_class_attribute_property() (tango.Database method), 194
get_class_attribute_property_history() (tango.Database method), 195
get_class_for_device() (tango.Database method), 195
get_class_inheritance_for_device() (tango.Database method), 195
get_class_list() (tango.Database method), 196
get_class_list() (tango.Util method), 177
get_class_pipe_list() (tango.Database method), 196
get_class_pipe_property() (tango.Database method), 196
get_class_pipe_property_history() (tango.Database method), 197
get_class_properties() (tango.Attr method), 156
get_class_property() (tango.Database method), 197
get_class_property_history() (tango.Database method), 198
get_class_property_list() (tango.Database method), 198
get_cmd_by_name() (tango.DeviceClass method), 150
get_cmd_min_poll_period() (tango.LatestDeviceImpl method), 138
get_cmd_min_poll_period() (tango.server.Device method), 116
get_cmd_poll_ring_depth() (tango.LatestDeviceImpl method), 138
get_cmd_poll_ring_depth() (tango.server.Device method), 117
get_command_config() (tango.DeviceProxy method), 40
get_command_list() (tango.DeviceClass method), 151
get_command_list() (tango.DeviceProxy method), 41
get_command_poll_period() (tango.DeviceProxy method), 41
get_command_poll_period() (tango.LatestDeviceImpl method), 138
get_command_poll_period() (tango.server.Device method), 117
get_config() (tango.AttributeProxy method), 69
get_cvs_location() (tango.DeviceClass method), 151
get_cvs_tag() (tango.DeviceClass method), 151
get_data() (tango.GroupAttrReply method), 90
get_data() (tango.GroupCmdReply method), 91
get_data_format() (tango.Attribute method), 161
get_data_raw() (tango.GroupCmdReply method), 91
get_data_size() (tango.Attribute method), 161
get_data_type() (tango.Attribute method), 161
get_database() (tango.Util method), 177
get_date() (tango.Attribute method), 161
get_date() (tango.DbHistory method), 218
get_date() (tango.DeviceAttribute method), 99
get_db_host() (tango.DeviceProxy method), 42
get_db_port() (tango.DeviceProxy method), 42
get_db_port_num() (tango.DeviceProxy method), 42
get_dev_host() (tango.DeviceProxy method), 42
get_dev_idl_version() (tango.LatestDeviceImpl method), 139
get_dev_idl_version() (tango.server.Device method), 117
get_dev_port() (tango.DeviceProxy method), 42
get_device() (tango.test_context.DeviceTestContext method), 275
get_device() (tango.test_context.MultiDeviceTestContext method), 277
get_device_access() (tango.test_context.DeviceTestContext method), 275
get_device_access() (tango.test_context.MultiDeviceTestContext method), 277
get_device_alias() (tango.Database method), 198
get_device_alias_list() (tango.Database method), 198
get_device_attr() (tango.LatestDeviceImpl method), 139
get_device_attr() (tango.server.Device method), 117
get_device_attribute_list()

(*tango.Database method*), 199
get_device_attribute_property() (*tango.Database method*), 199
get_device_attribute_property_history() (*tango.Database method*), 200
get_device_by_name() (*tango.Util method*), 177
get_device_class() (*tango.LatestDeviceImpl method*), 139
get_device_class() (*tango.server.Device method*), 117
get_device_class_list() (*tango.Database method*), 200
get_device_db() (*tango.DeviceProxy method*), 42
get_device_domain() (*tango.Database method*), 200
get_device_exported() (*tango.Database method*), 201
get_device_exported_for_class() (*tango.Database method*), 201
get_device_family() (*tango.Database method*), 201
get_device_from_alias() (*tango.Database method*), 201
get_device_info() (*tango.Database method*), 201
get_device_ior() (*tango.Util method*), 177
get_device_list() (*tango.DeviceClass method*), 151
get_device_list() (*tango.Group method*), 85
get_device_list() (*tango.Util method*), 177
get_device_list_by_class() (*tango.Util method*), 178
get_device_member() (*tango.Database method*), 202
get_device_name() (*tango.Database method*), 202
get_device_pipe_list() (*tango.Database method*), 202
get_device_pipe_property() (*tango.Database method*), 203
get_device_pipe_property_history() (*tango.Database method*), 203
get_device_properties() (*tango.LatestDeviceImpl method*), 139
get_device_properties() (*tango.server.Device method*), 117
get_device_property() (*tango.Database method*), 204
get_device_property_history() (*tango.Database method*), 204
get_device_property_list() (*tango.Database method*), 204
get_device_proxy() (*tango.AttributeProxy method*), 70
get_device_service_list() (*tango.Database method*), 205
get_disp_level() (*tango.Attr method*), 157
get_doc_url() (*tango.DeviceClass method*), 151
get_ds_exec_name() (*tango.Util method*), 178
get_ds_inst_name() (*tango.Util method*), 178
get_ds_name() (*tango.Util method*), 178
get_dserver_device() (*tango.Util method*), 178
get_dserver_ior() (*tango.Util method*), 179
get_enum_labels() (*in module tango.utils*), 226
get_err_stack() (*tango.DeviceAttribute method*), 99
get_events() (*tango.AttributeProxy method*), 70
get_events() (*tango.DeviceProxy method*), 43
get_events() (*tango.utils.EventCallback method*), 226
get_exported_flag() (*tango.LatestDeviceImpl method*), 139
get_exported_flag() (*tango.server.Device method*), 118
get_file_name() (*tango.Database method*), 205
get_format() (*tango.Attr method*), 157
get_fqdn() (*tango.DeviceProxy method*), 43
get_from_env_var() (*tango.DeviceProxy method*), 43
get_fully_qualified_name() (*tango.Group method*), 86
get_green_mode() (*in module tango*), 91
get_green_mode() (*tango.DeviceProxy method*), 44
get_home() (*in module tango.utils*), 230
get_host_list() (*tango.Database method*), 205
get_host_name() (*tango.Util method*), 179
get_host_server_list() (*tango.Database method*), 206
get_idl_version() (*tango.DeviceProxy method*), 44
get_info() (*tango.Database method*), 206
get_instance_name_list() (*tango.Database method*), 206
get_label() (*tango.Attribute method*), 161
get_last_event_date() (*tango.AttributeProxy method*), 71
get_last_event_date() (*tango.DeviceProxy method*), 44
get_locker() (*tango.DeviceProxy method*), 44
get_logger() (*tango.LatestDeviceImpl method*), 139
get_logger() (*tango.server.Device method*), 118
get_logging_level() (*tango.DeviceProxy method*), 45
get_logging_target() (*tango.DeviceProxy method*), 45
get_max_dim_x() (*tango.Attribute method*), 162
get_max_dim_y() (*tango.Attribute method*), 162
get_max_value() (*tango.WAttribute method*), 168
get_memorized() (*tango.Attr method*), 157
get_memorized_init() (*tango.Attr method*), 157
get_min_poll_period() (*tango.LatestDeviceImpl method*), 140
get_min_poll_period() (*tango.server.Device

method), 118
get_min_value() (*tango.WAttribute method*), 168
get_name() (*tango.Attr method*), 157
get_name() (*tango.Attribute method*), 162
get_name() (*tango.DbHistory method*), 218
get_name() (*tango.DeviceClass method*), 152
get_name() (*tango.Group method*), 86
get_name() (*tango.LatestDeviceImpl method*), 140
get_name() (*tango.server.Device method*), 118
get_non_auto_polled_attr() (*tango.LatestDeviceImpl method*), 140
get_non_auto_polled_attr() (*tango.server.Device method*), 118
get_non_auto_polled_cmd() (*tango.LatestDeviceImpl method*), 140
get_non_auto_polled_cmd() (*tango.server.Device method*), 118
get_object_list() (*tango.Database method*), 206
get_object_property_list() (*tango.Database method*), 207
get_pid() (*tango.Util method*), 179
get_pid_str() (*tango.Util method*), 179
get_pipe_by_name() (*tango.DeviceClass method*), 152
get_pipe_config() (*tango.DeviceProxy method*), 45
get_pipe_list() (*tango.DeviceClass method*), 152
get_poll_old_factor() (*tango.LatestDeviceImpl method*), 140
get_poll_old_factor() (*tango.server.Device method*), 119
get_poll_period() (*tango.AttributeProxy method*), 71
get_poll_ring_depth() (*tango.LatestDeviceImpl method*), 140
get_poll_ring_depth() (*tango.server.Device method*), 119
get_polled_attr() (*tango.LatestDeviceImpl method*), 140
get_polled_attr() (*tango.server.Device method*), 119
get_polled_cmd() (*tango.LatestDeviceImpl method*), 141
get_polled_cmd() (*tango.server.Device method*), 119
get_polling_period() (*tango.Attr method*), 157
get_polling_period() (*tango.Attribute method*), 162
get_polling_threads_pool_size() (*tango.Util method*), 179
get_prev_state() (*tango.LatestDeviceImpl method*), 141
get_prev_state() (*tango.server.Device method*), 119
get_properties() (*tango.Attribute method*), 162
get_property() (*tango.AttributeProxy method*), 71
get_property() (*tango.Database method*), 207
get_property() (*tango.DeviceProxy method*), 46
get_property_forced() (*tango.Database method*), 207
get_property_history() (*tango.Database method*), 208
get_property_list() (*tango.DeviceProxy method*), 46
get_quality() (*tango.Attribute method*), 162
get_serial_model() (*tango.Util method*), 180
get_server_access() (*tango.test_context.DeviceTestContext method*), 275
get_server_access() (*tango.test_context.MultiDeviceTestContext method*), 277
get_server_class_list() (*tango.Database method*), 208
get_server_info() (*tango.Database method*), 209
get_server_list() (*tango.Database method*), 209
get_server_name_list() (*tango.Database method*), 209
get_server_version() (*tango.Util method*), 180
get_services() (*tango.Database method*), 210
get_size() (*tango.Group method*), 87
get_source() (*tango.DeviceProxy method*), 47
get_state() (*tango.LatestDeviceImpl method*), 141
get_state() (*tango.server.Device method*), 119
get_status() (*tango.LatestDeviceImpl method*), 141
get_status() (*tango.server.Device method*), 119
get_sub_dev_diag() (*tango.Util method*), 180
get_tango_lib_release() (*tango.Util method*), 180
get_tango_lib_version() (*tango.DeviceProxy method*), 47
get_timeout_millis() (*tango.DeviceProxy method*), 47
get_trace_level() (*tango.Util method*), 180
get_transparency_reconnection() (*tango.AttributeProxy method*), 72
get_transparency_reconnection() (*tango.DeviceProxy method*), 47
get_type() (*tango.Attr method*), 157
get_type() (*tango.DeviceClass method*), 152
get_type() (*tango.DeviceData method*), 100
get_user_default_properties() (*tango.Attr method*), 158
get_value() (*tango.DbHistory method*), 218
get_version_str() (*tango.Util method*), 180
get_w_attr_by_ind() (*tango.MultiAttribute method*), 171
get_w_attr_by_name() (*tango.MultiAttribute method*), 171

get_writable() (*tango.Attr method*), 158
get_writable() (*tango.Attribute method*), 163
get_write_value() (*tango.WAttribute method*), 168
get_write_value_length() (*tango.WAttribute method*), 168
get_x() (*tango.Attribute method*), 163
get_y() (*tango.Attribute method*), 163
GreenMode (*class in tango*), 106
Group (*class in tango*), 83
GroupAttrReply (*class in tango*), 90
GroupCmdReply (*class in tango*), 91
GroupReply (*class in tango*), 90

H

history() (*tango.AttributeProxy method*), 72

I

import_device() (*tango.Database method*), 210
import_info() (*tango.DeviceProxy method*), 47
info() (*tango.DeviceProxy method*), 48
info_stream() (*tango.LatestDeviceImpl method*), 141
info_stream() (*tango.server.Device method*), 120
InfoIt (*class in tango*), 155
init() (*tango.Util method*), 181
init_device() (*tango.LatestDeviceImpl method*), 141
init_device() (*tango.server.Device method*), 120
init_logger() (*tango.LatestDeviceImpl method*), 141
init_logger() (*tango.server.Device method*), 120
initialize_dynamic_attributes() (*tango.server.Device method*), 120
insert() (*tango.DeviceData method*), 100
instance() (*tango.ApiUtil method*), 94
instance() (*tango.Util method*), 181
is_allowed() (*tango.Attr method*), 158
is_archive_event() (*tango.Attr method*), 158
is_archive_event() (*tango.Attribute method*), 163
is_array_type() (*in module tango.utils*), 228
is_assoc() (*tango.Attr method*), 158
is_attribute_polled() (*tango.DeviceProxy method*), 48
is_attribute_polled() (*tango.LatestDeviceImpl method*), 141
is_attribute_polled() (*tango.server.Device method*), 120
is_binary_type() (*in module tango.utils*), 229
is_bool() (*in module tango.utils*), 227
is_bool_type() (*in module tango.utils*), 229
is_change_event() (*tango.Attr method*), 158
is_change_event() (*tango.Attribute method*), 163
is_check_archive_criteria() (*tango.Attr method*), 158
is_check_archive_criteria() (*tango.Attribute method*), 163
is_check_change_criteria() (*tango.Attr method*), 159
is_check_change_criteria() (*tango.Attribute method*), 163
is_command_polled() (*tango.DeviceProxy method*), 48
is_command_polled() (*tango.LatestDeviceImpl method*), 142
is_command_polled() (*tango.server.Device method*), 120
is_control_access_checked() (*tango.Database method*), 210
is_data_ready_event() (*tango.Attr method*), 159
is_data_ready_event() (*tango.Attribute method*), 164
is_dbase_used() (*tango.DeviceProxy method*), 48
is_deleted() (*tango.DbHistory method*), 218
is_device_locked() (*tango.LatestDeviceImpl method*), 142
is_device_locked() (*tango.server.Device method*), 120
is_device_restarting() (*tango.Util method*), 181
is_empty() (*tango.DbDatum method*), 217
is_empty() (*tango.DeviceData method*), 100
is_enabled() (*tango.Group method*), 87
is_event_queue_empty() (*tango.AttributeProxy method*), 73
is_event_queue_empty() (*tango.DeviceProxy method*), 49
is_float_type() (*in module tango.utils*), 228
is_int_type() (*in module tango.utils*), 228
is_integer() (*in module tango.utils*), 227
is_locked() (*tango.DeviceProxy method*), 49
is_locked_by_me() (*tango.DeviceProxy method*), 49
is_max_alarm() (*tango.Attribute method*), 164
is_max_value() (*tango.WAttribute method*), 169
is_max_warning() (*tango.Attribute method*), 164
is_min_alarm() (*tango.Attribute method*), 164
is_min_value() (*tango.WAttribute method*), 169
is_min_warning() (*tango.Attribute method*), 164
is_multi_tango_host() (*tango.Database method*), 210
is_non_str_seq() (*in module tango.utils*), 227
is_number() (*in module tango.utils*), 227
is_numerical_type() (*in module tango.utils*), 228
is_polled() (*tango.Attribute method*), 164
is_polled() (*tango.AttributeProxy method*), 73
is_polled() (*tango.LatestDeviceImpl method*), 142
is_polled() (*tango.server.Device method*), 120
is_pure_str() (*in module tango.utils*), 226
is_rds_alarm() (*tango.Attribute method*), 164
is_scalar_type() (*in module tango.utils*), 228

is_seq() (in module tango.utils), 227
is_str_type() (in module tango.utils), 229
is_svr_shutting_down() (tango.Util method), 181
is_svr_starting() (tango.Util method), 181
is_there_subscriber() (tango.LatestDeviceImpl method), 142
is_there_subscriber() (tango.server.Device method), 121
is_write_associated() (tango.Attribute method), 165
isoformat() (tango.TimeVal method), 108

K

KeepAliveCmdCode (class in tango), 105

L

LatestDeviceImpl (class in tango), 134
lock() (tango.DeviceProxy method), 49
LockCmdCode (class in tango), 104
LockerInfo (class in tango), 97
LockerLanguage (class in tango), 103
locking_status() (tango.DeviceProxy method), 50
LogIt (class in tango), 154
LogLevel (class in tango), 104
LogTarget (class in tango), 104

M

MessBoxType (class in tango), 104
module
    tango, 21
    tango.server, 109
    tango.test_context, 269
MultiAttribute (class in tango), 169
MultiDeviceTestContext (class in tango.test_context), 275

N

name() (tango.AttributeProxy method), 73
name() (tango.DeviceProxy method), 50
name_equals() (tango.Group method), 87
name_matches() (tango.Group method), 87
NamedDevFailedList, 238
NonDbDevice, 236
NonSupportedFeature, 236
NotAllowed, 237
now() (tango.TimeVal static method), 108

O

obj_2_str() (in module tango.utils), 229
orb_run() (tango.Util method), 182

P

pending_asynch_call() (tango.ApiUtil method), 94
pending_asynch_call() (tango.DeviceProxy method), 50
PeriodicEventInfo (class in tango), 102
ping() (tango.AttributeProxy method), 73
ping() (tango.DeviceProxy method), 50
ping() (tango.Group method), 87
pipe (class in tango.server), 130
PipeWriteType (class in tango), 106
poll() (tango.AttributeProxy method), 74
poll_attribute() (tango.DeviceProxy method), 51
poll_attribute() (tango.LatestDeviceImpl method), 142
poll_attribute() (tango.server.Device method), 121
poll_command() (tango.DeviceProxy method), 51
poll_command() (tango.LatestDeviceImpl method), 143
poll_command() (tango.server.Device method), 121
PollCmdCode (class in tango), 104
PollDevice (class in tango), 98
polling_status() (tango.DeviceProxy method), 51
PollObjType (class in tango), 104
print_error_stack() (tango.Except method), 232
print_exception() (tango.Except method), 232
push_archive_event() (tango.LatestDeviceImpl method), 143
push_archive_event() (tango.server.Device method), 121
push_att_conf_event() (tango.LatestDeviceImpl method), 143
push_att_conf_event() (tango.server.Device method), 122
push_change_event() (tango.LatestDeviceImpl method), 143
push_change_event() (tango.server.Device method), 122
push_data_ready_event() (tango.LatestDeviceImpl method), 144
push_data_ready_event() (tango.server.Device method), 123
push_event() (tango.LatestDeviceImpl method), 144
push_event() (tango.server.Device method), 123
push_event() (tango.utils.EventCallback method), 226
push_pipe_event() (tango.LatestDeviceImpl method), 145
push_pipe_event() (tango.server.Device method), 123
put_attribute_alias() (tango.Database method), 210
put_class_attribute_property() (tango.Database method), 211
put_class_pipe_property() (tango.Database

method), 211
put_class_property() (tango.Database method), 212
put_device_alias() (tango.Database method), 212
put_device_attribute_property() (tango.Database method), 212
put_device_pipe_property() (tango.Database method), 213
put_device_property() (tango.Database method), 213
put_property() (tango.AttributeProxy method), 74
put_property() (tango.Database method), 214
put_property() (tango.DeviceProxy method), 52
put_server_info() (tango.Database method), 214
PYTANGO_GREEN_MODE, 13
Python Enhancement Proposals
PEP 249, 299, 302

R

re_throw_exception() (tango.Except method), 233
read() (tango.AttributeProxy method), 75
read_alarm() (tango.MultiAttribute method), 171
read_asynch() (tango.AttributeProxy method), 76
read_attr_hardware() (tango.LatestDeviceImpl method), 145
read_attr_hardware() (tango.server.Device method), 124
read_attribute() (tango.DeviceProxy method), 52
read_attribute() (tango.Group method), 87
read_attribute_asynch() (tango.DeviceProxy method), 53
read_attribute_asynch() (tango.Group method), 87
read_attribute_reply() (tango.DeviceProxy method), 53
read_attribute_reply() (tango.Group method), 88
read_attributes() (tango.DeviceProxy method), 53
read_attributes() (tango.Group method), 88
read_attributes_asynch() (tango.DeviceProxy method), 54
read_attributes_asynch() (tango.Group method), 88
read_attributes_reply() (tango.DeviceProxy method), 54
read_attributes_reply() (tango.Group method), 88
read_pipe() (tango.DeviceProxy method), 55
read_reply() (tango.AttributeProxy method), 76
reconnect() (tango.DeviceProxy method), 56
register_service() (tango.Database method), 214
register_signal() (tango.DeviceClass method), 152
register_signal() (tango.LatestDeviceImpl method), 145
register_signal() (tango.server.Device method), 124
Release (class in tango), 107
remove_all() (tango.Group method), 89
remove_attribute() (tango.LatestDeviceImpl method), 145
remove_attribute() (tango.server.Device method), 124
remove_command() (tango.LatestDeviceImpl method), 145
remove_command() (tango.server.Device method), 124
remove_configuration() (tango.Attribute method), 165
remove_logging_target() (tango.DeviceProxy method), 56
rename_server() (tango.Database method), 215
requires_pytango() (in module tango.utils), 230
requires_tango() (in module tango.utils), 230
reread_filedatabase() (tango.Database method), 215
reset_filedatabase() (tango.Util method), 182
run() (in module tango.server), 132
run_server() (tango.server.Device class method), 124

S

scalar_to_array_type() (in module tango.utils), 230
seqStr_2_obj() (in module tango.utils), 229
SerialModel (class in tango), 104
server_cleanup() (tango.Util method), 182
server_init() (tango.Util method), 182
server_run() (in module tango.server), 134
server_run() (tango.Util method), 182
server_set_event_loop() (tango.Util method), 183
set_abs_change() (tango.UserDefaultAttrProp method), 171
set_access_checked() (tango.Database method), 215
set_access_control() (tango.DeviceProxy method), 56
set_archive_abs_change() (tango.UserDefaultAttrProp method), 172
set_archive_event() (tango.Attr method), 159
set_archive_event() (tango.Attribute method), 165
set_archive_event() (tango.LatestDeviceImpl method), 146
set_archive_event() (tango.server.Device method), 124
set_archive_event_abs_change()

(*tango.UserDefaultAttrProp method*), 172
set_archive_event_period() (*tango.UserDefaultAttrProp method*), 172
set_archive_event_rel_change() (*tango.UserDefaultAttrProp method*), 172
set_archive_period() (*tango.UserDefaultAttrProp method*), 172
set_archive_rel_change() (*tango.UserDefaultAttrProp method*), 172
set_assoc_ind() (*tango.Attribute method*), 165
set_asynch_cb_sub_model() (*tango.ApiUtil method*), 94
set_attr_serial_model() (*tango.Attribute method*), 165
set_attribute_config() (*tango.DeviceProxy method*), 57
set_attribute_config_3() (*tango.LatestDeviceImpl method*), 146
set_attribute_config_3() (*tango.server.Device method*), 125
set_change_event() (*tango.Attr method*), 159
set_change_event() (*tango.Attribute method*), 165
set_change_event() (*tango.LatestDeviceImpl method*), 146
set_change_event() (*tango.server.Device method*), 125
set_cl_name() (*tango.Attr method*), 159
set_class_properties() (*tango.Attr method*), 160
set_config() (*tango.AttributeProxy method*), 76
set_data_ready_event() (*tango.Attr method*), 160
set_data_ready_event() (*tango.Attribute method*), 166
set_data_ready_event() (*tango.LatestDeviceImpl method*), 146
set_data_ready_event() (*tango.server.Device method*), 125
set_date() (*tango.Attribute method*), 166
set_default_properties() (*tango.Attr method*), 160
set_delta_t() (*tango.UserDefaultAttrProp method*), 173
set_delta_val() (*tango.UserDefaultAttrProp method*), 173
set_description() (*tango.UserDefaultAttrProp method*), 173
set_disp_level() (*tango.Attr method*), 160
set_display_unit() (*tango.UserDefaultAttrProp method*), 173
set_enum_labels() (*tango.UserDefaultAttrProp method*), 173
set_event_abs_change() (*tango.UserDefaultAttrProp method*), 173
set_event_period() (*tango.UserDefaultAttrProp method*), 173
set_event_rel_change() (*tango.UserDefaultAttrProp method*), 173
set_format() (*tango.UserDefaultAttrProp method*), 174
set_green_mode() (*in module tango*), 91
set_green_mode() (*tango.DeviceProxy method*), 58
set_label() (*tango.UserDefaultAttrProp method*), 174
set_logging_level() (*tango.DeviceProxy method*), 58
set_max_alarm() (*tango.UserDefaultAttrProp method*), 174
set_max_value() (*tango.UserDefaultAttrProp method*), 174
set_max_value() (*tango.WAttribute method*), 169
set_max_warning() (*tango.UserDefaultAttrProp method*), 174
set_memorized() (*tango.Attr method*), 160
set_memorized_init() (*tango.Attr method*), 160
set_min_alarm() (*tango.UserDefaultAttrProp method*), 174
set_min_value() (*tango.UserDefaultAttrProp method*), 174
set_min_value() (*tango.WAttribute method*), 169
set_min_warning() (*tango.UserDefaultAttrProp method*), 174
set_period() (*tango.UserDefaultAttrProp method*), 174
set_pipe_config() (*tango.DeviceProxy method*), 58
set_polling_period() (*tango.Attr method*), 160
set_polling_threads_pool_size() (*tango.Util method*), 183
set_properties() (*tango.Attribute method*), 166
set_quality() (*tango.Attribute method*), 166
set_rel_change() (*tango.UserDefaultAttrProp method*), 174
set_serial_model() (*tango.Util method*), 183
set_server_version() (*tango.Util method*), 184
set_source() (*tango.DeviceProxy method*), 59
set_standard_unit() (*tango.UserDefaultAttrProp method*), 175
set_state() (*tango.LatestDeviceImpl method*), 147
set_state() (*tango.server.Device method*), 125
set_status() (*tango.LatestDeviceImpl method*), 147

set_status() (*tango.server.Device method*), 125
set_timeout_millis() (*tango.DeviceProxy method*), 59
set_timeout_millis() (*tango.Group method*), 89
set_trace_level() (*tango.Util method*), 184
set_transparency_reconnection() (*tango.AttributeProxy method*), 77
set_transparency_reconnection() (*tango.DeviceProxy method*), 59
set_type() (*tango.DeviceClass method*), 153
set_unit() (*tango.UserDefaultAttrProp method*), 175
set_value() (*tango.Attribute method*), 166
set_value_date_quality() (*tango.Attribute method*), 167
set_w_dim_x() (*tango.DeviceAttribute method*), 99
set_w_dim_y() (*tango.DeviceAttribute method*), 99
signal_handler() (*tango.DeviceClass method*), 153
signal_handler() (*tango.LatestDeviceImpl method*), 147
signal_handler() (*tango.server.Device method*), 125
size() (*tango.DbDatum method*), 217
start() (*tango.test_context.DeviceTestContext method*), 275
start() (*tango.test_context.MultiDeviceTestContext method*), 277
start_logging() (*tango.LatestDeviceImpl method*), 147
start_logging() (*tango.server.Device method*), 126
state() (*tango.AttributeProxy method*), 77
state() (*tango.DeviceProxy method*), 59
status() (*tango.AttributeProxy method*), 78
status() (*tango.DeviceProxy method*), 60
stop() (*tango.test_context.DeviceTestContext method*), 275
stop() (*tango.test_context.MultiDeviceTestContext method*), 277
stop_logging() (*tango.LatestDeviceImpl method*), 147
stop_logging() (*tango.server.Device method*), 126
stop_poll() (*tango.AttributeProxy method*), 78
stop_poll_attribute() (*tango.DeviceProxy method*), 60
stop_poll_attribute() (*tango.LatestDeviceImpl method*), 147
stop_poll_attribute() (*tango.server.Device method*), 126
stop_poll_command() (*tango.DeviceProxy method*), 60
stop_poll_command() (*tango.LatestDeviceImpl method*), 147
stop_poll_command() (*tango.server.Device method*), 126
stop_polling() (*tango.LatestDeviceImpl method*), 147
stop_polling() (*tango.server.Device method*), 126
strftime() (*tango.TimeVal method*), 108
subscribe_event() (*tango.AttributeProxy method*), 78
subscribe_event() (*tango.DeviceProxy method*), 60

T

tango
  module, 21
tango.server
  module, 109
tango.test_context
  module, 269
TANGO_HOST, 3, 4, 239
throw_exception() (*tango.Except method*), 233
throw_python_exception() (*tango.Except method*), 233
TimeVal (*class in tango*), 107
to_dev_failed() (*tango.Except static method*), 234
todatetime() (*tango.TimeVal method*), 108
totime() (*tango.TimeVal method*), 108
trigger_attr_polling() (*tango.Util method*), 184
trigger_cmd_polling() (*tango.Util method*), 184

U

unexport_device() (*tango.Database method*), 215
unexport_event() (*tango.Database method*), 216
unexport_server() (*tango.Database method*), 216
unlock() (*tango.DeviceProxy method*), 62
unregister_server() (*tango.Util method*), 185
unregister_service() (*tango.Database method*), 216
unregister_signal() (*tango.DeviceClass method*), 153
unregister_signal() (*tango.LatestDeviceImpl method*), 148
unregister_signal() (*tango.server.Device method*), 126
unsubscribe_event() (*tango.AttributeProxy method*), 80
unsubscribe_event() (*tango.DeviceProxy method*), 62
UserDefaultAttrProp (*class in tango*), 171
Util (*class in tango*), 175

W

warn_stream() (*tango.LatestDeviceImpl method*),

148
warn_stream() (*tango.server.Device method*), 126
WarnIt (*class in tango*), 155
WAttribute (*class in tango*), 168
write() (*tango.AttributeProxy method*), 81
write_asynch() (*tango.AttributeProxy method*), 82
write_attr_hardware() (*tango.LatestDeviceImpl method*), 148
write_attr_hardware() (*tango.server.Device method*), 127
write_attribute() (*tango.DeviceProxy method*), 63
write_attribute() (*tango.Group method*), 89
write_attribute_asynch() (*tango.DeviceProxy method*), 63
write_attribute_asynch() (*tango.Group method*), 89
write_attribute_reply() (*tango.DeviceProxy method*), 63
write_attribute_reply() (*tango.Group method*), 90
write_attributes() (*tango.DeviceProxy method*), 64
write_attributes_asynch() (*tango.DeviceProxy method*), 65
write_attributes_reply() (*tango.DeviceProxy method*), 65
write_filedatabase() (*tango.Database method*), 216
write_pipe() (*tango.DeviceProxy method*), 66
write_read() (*tango.AttributeProxy method*), 82
write_read_attribute() (*tango.DeviceProxy method*), 67
write_read_attributes() (*tango.DeviceProxy method*), 67
write_reply() (*tango.AttributeProxy method*), 82
WrongData, 236
WrongNameSyntax, 235