



PyTango Documentation

Release 9.5.1

PyTango team

Mar 29, 2024

CONTENTS

1	Getting started	1
1.1	Installing pre-compiled packages (easy)	1
1.2	Building and installing from source (hard)	2
1.3	Basic installation check	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	23
5.1	Data types	23
5.2	Client API	32
5.3	Server API	113
5.4	Database API	190
5.5	Encoded API	224
5.6	The Utilities API	230
5.7	Exception API	236
6	How to	243
6.1	How to contribute	243
6.2	Check the default TANGO host	243
6.3	Check TANGO version	243
6.4	Start server from command line	244
6.5	Report a bug	244
6.6	Test the connection to the Device and get it's current state	245
6.7	Read and write attributes	245
6.8	Execute commands	246
6.9	Execute commands with more complex types	247
6.10	Work with Groups	247
6.11	Handle errors	247
6.12	Registering devices	247
6.13	Using clients with multiprocessing	249
6.14	Multithreading - clients and servers	250
6.15	Write a server	251
6.16	Use Python type hints when declaring a device	254
6.17	Server logging	259
6.18	Multiple device classes (Python and C++) in a server	261

6.19	Create attributes dynamically	262
6.20	Create/Delete devices dynamically	266
6.21	Write a server (original API)	267
7	How to Contribute	275
7.1	Workflow	275
7.2	reStructuredText and Sphinx	275
7.3	Source code standard	276
7.4	Using Conda for development	276
7.5	Using Docker for development	276
7.6	Releasing a new version	276
8	Testing PyTango Devices	281
8.1	Approaches to testing Tango devices	281
8.2	Device Test Context Classes API	288
8.3	Mocking clients for Testing	291
8.4	Code coverage for Tango devices	296
9	Python and NumPy version policy	299
10	FAQ	301
11	PyTango Enhancement Proposals	303
11.1	TEP 1 - Device Server High Level API	303
11.2	TEP 2 - Tango database serverless	314
12	What's new?	319
12.1	Migration guide	319
12.2	History of changes	337
12.3	What's new in PyTango 9.5.1?	366
12.4	What's new in PyTango 9.5.0?	366
12.5	What's new in PyTango 9.4.2?	368
12.6	What's new in PyTango 9.4.1?	368
12.7	What's new in PyTango 9.4.0?	369
13	Indexes	373
	Python Module Index	375
	Index	377

GETTING STARTED

The first section show the easy way, with pre-compiled packages. If that doesn't work, or you need to compile from source, see the second section.

Contents

- *Installing pre-compiled packages (easy)*
- *Building and installing from source (hard)*
- *Basic installation check*

1.1 Installing pre-compiled packages (easy)

1.1.1 PyPI (Linux, Windows)

You can install the latest version from [PyPI](#).

PyTango has binary wheels for common platforms, so no compilation or dependencies required. However, pip needs to be at least version 19.3 in order for it to find the binary wheels:

```
$ python -m pip install --upgrade pip
```

Install PyTango with pip:

```
$ python -m pip install pytango
```

If this step downloads a `.tar.gz` file instead of a `.whl` file, then we don't have a binary package for your platform. Try Conda.

If you are going to utilize the `gevent` green mode of PyTango it is recommended to have a recent version of `gevent`. You can force `gevent` installation with the "gevent" keyword:

```
$ python -m pip install pytango[gevent]
```

1.1.2 Conda (Linux, Windows, MacOS)

You can install the latest version from [Conda-forge](#).

Conda-forge provides binary wheels for different platforms, compared to [PyPI](#). MacOS binaries are available since version 9.4.0.

If you don't already have conda, try the [Miniforge3](#) installer (an alternative installer to [Miniconda](#)).

To install PyTango in a new conda environment (you can choose a different version of Python):

```
$ conda create --channel conda-forge --name pytango-env python=3.11 pytango
$ conda activate pytango-env
```

Other useful packages on conda-forge include: `tango-test`, `jive` and `tango-database`.

1.1.3 Linux

PyTango is available on linux as an official debian/ubuntu package (however, this may not be the latest release):

For Python 3:

```
$ 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](#)

1.1.4 Windows

First, make sure [Python](#) is installed. Then follow the same instructions as for [PyPI](#) above. There are binary wheels for some Windows platforms available.

1.2 Building and installing from source (hard)

This is the more complicated option, as you need to have all the correct dependencies and build tools installed. It is possible to build in Conda environments on Linux, macOS and Windows. It is also possible to build natively on those operating system. Conda is the recommended option for simplicity. For details see the file [BUILD.md](#) in the root of the source repository.

1.3 Basic installation check

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

```
$ cd # move to a folder that doesn't contain the source code, if you_
    ↪built it
$ python -c "import tango; print(tango.Release.version)"
9.4.0
```

Next steps: Check out the [Quick tour](#).

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 structured (in the sense 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 has 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`

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  """Demo power supply tango device server"""
5
6  import time
7  import numpy
8
9  from tango import AttrQuality, AttrWriteType, DispLevel, DevState, DebugIt
10 from tango.server import Device, attribute, command, pipe, device_property
11
12
13 class PowerSupply(Device):
14     voltage = attribute(
15         label="Voltage",
16         dtype=float,
17         display_level=DispLevel.OPERATOR,
18         access=AttrWriteType.READ,
19         unit="V",
20         format="8.4f",
21         doc="the power supply voltage",
22     )
```

(continues on next page)

(continued from previous page)

```

23
24     current = attribute(
25         label="Current",
26         dtype=float,
27         display_level=DispLevel.EXPERT,
28         access=AttrWriteType.READ_WRITE,
29         unit="A",
30         format="8.4f",
31         min_value=0.0,
32         max_value=8.5,
33         min_alarm=0.1,
34         max_alarm=8.4,
35         min_warning=0.5,
36         max_warning=8.0,
37         fget="get_current",
38         fset="set_current",
39         doc="the power supply current",
40     )
41
42     noise = attribute(label="Noise", dtype=((int,)), max_dim_x=1024, max_
43     ↪dim_y=1024)
44
45     info = pipe(label="Info")
46
47     host = device_property(dtype=str)
48     port = device_property(dtype=int, default_value=9788)
49
50     def init_device(self):
51         Device.init_device(self)
52         self.__current = 0.0
53         self.set_state(DevState.STANDBY)
54
55     def read_voltage(self):
56         self.info_stream("read_voltage(%s, %d)", self.host, self.port)
57         return 9.99, time.time(), AttrQuality.ATTR_WARNING
58
59     def get_current(self):
60         return self.__current
61
62     def set_current(self, current):
63         # should set the power supply current
64         self.__current = current
65
66     def read_info(self):
67         return "Information", dict(
68             manufacturer="Tango", model="PS2000", version_number=123
69         )
70
71     @DebugIt()
72     def read_noise(self):
73         return numpy.random.random_integers(1000, size=(100, 100))
74
75     @command
76     def TurnOn(self):
77         # turn on the actual power supply here
78         self.set_state(DevState.ON)

```

(continues on next page)

(continued from previous page)

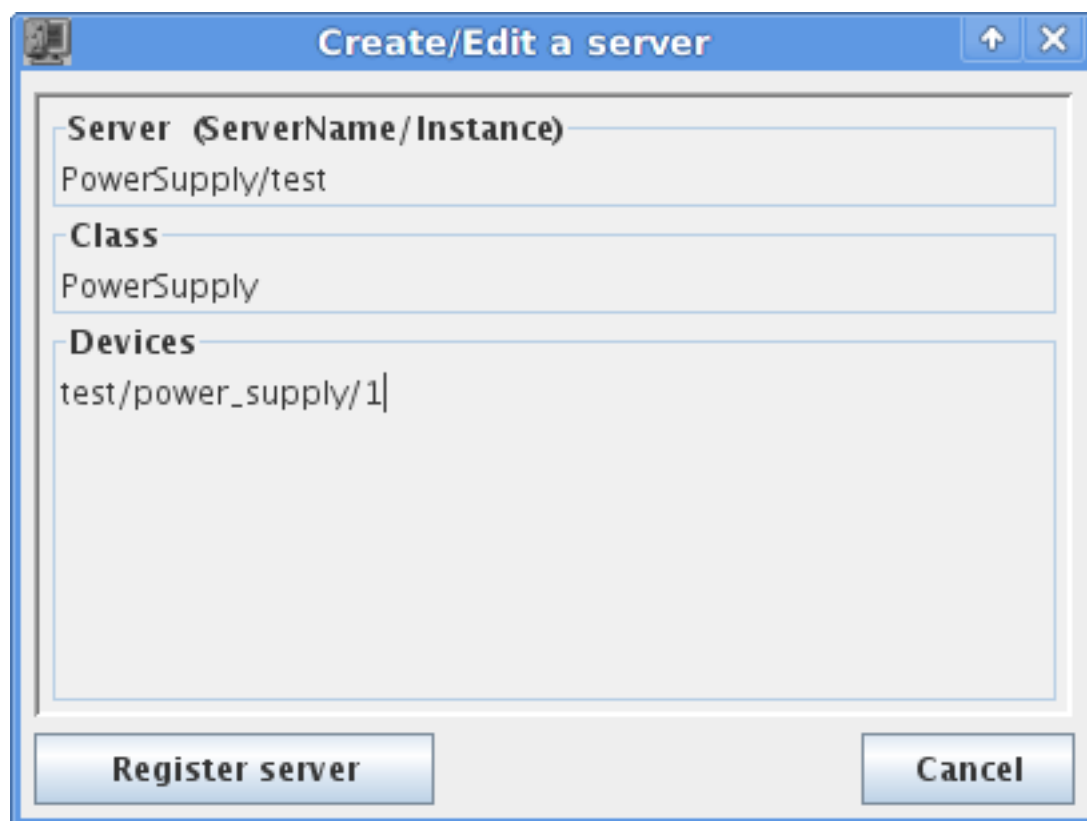
```

78
79 @command
80 def TurnOff(self):
81     # turn off the actual power supply here
82     self.set_state(DevState.OFF)
83
84 @command(
85     dtype_in=float,
86     doc_in="Ramp target current",
87     dtype_out=bool,
88     doc_out="True if ramping went well, False otherwise",
89 )
90 def Ramp(self, target_current):
91     # should do the ramping
92     return True
93
94
95 if __name__ == "__main__":
96     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"

```

(continues on next page)

(continued from previous page)

```
>>> 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.

ITANGO

ITango is a PyTango CLI based on [IPython](#). It is designed to be used as an IPython profile.



```
ITango
tcoutinho@pc151:~$ itango
ITango 8.0.0 -- An interactive Tango client.

Running on top of Python 2.7.1, IPython 0.14.dev and PyTango 8.0.0

help      -> ITango's help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

IPython profile: tango

hint: Try typing: mydev = Device("<tab>

ITango [1]:
```

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](#)

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 specify 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.

```
1 import asyncio
2 from tango.asyncio import DeviceProxy
3
4
5 async def asyncio_example():
6     dev = await DeviceProxy("sys/tg_test/1")
7     print(dev.get_green_mode())
```

(continues on next page)

(continued from previous page)

```

8
9     print(await dev.state())
10
11     # in case of high-level API read has to be awaited
12     print(await dev.long_scalar)
13     print(await dev["long_scalar"])
14     print(await getattr(dev, "long_scalar"))
15
16     # while write executed sync
17     dev.long_scalar = 1
18
19     # for low-level API both read_attribute and write_attribute have to be_
    ↪awaited
20     print(await dev.read_attribute("long_scalar"))
21     await dev.write_attribute("long_scalar", 1)
22
23
24 if __name__ == "__main__":
25     asyncio.run(asyncio_example())

```

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.

```

1  """A simple TCP server for Tango attributes.
2
3  It runs on all interfaces on port 8888:
4
5      $ python tango_tcp_server.py
6      Serving on 0.0.0.0 port 8888
7
8  It can be accessed using netcat:
9
10     $ ncat localhost 8888
11     >>> sys/tg_test/1/ampli
12     0.0
13     >>> sys/tg_test/1/state
14     RUNNING
15     >>> sys/tg_test/1/nope
16     DevFailed[
17     DevError[
18         desc = Attribute nope is not supported by device sys/tg_test/1
19         origin = AttributeProxy::real_constructor()
20         reason = API_UnsupportedAttribute
21         severity = ERR]
22     ]
23     >>> ...
24 """
25
26 import asyncio
27 from tango.asyncio import AttributeProxy
28
29
30 async def handle_echo(reader, writer):
31     # Write the cursor
32     writer.write(b">>> ")
33     # Loop over client request

```

(continues on next page)

(continued from previous page)

```

34  async for line in reader:
35      request = line.decode().strip()
36      # Get attribute value using asyncio green mode
37      try:
38          proxy = await AttributeProxy(request)
39          attr_value = await proxy.read()
40          reply = str(attr_value.value)
41          # Catch exception if something goes wrong
42          except Exception as exc:
43              reply = str(exc)
44              # Reply to client
45              writer.write(reply.encode() + b"\n" + b">>> ")
46          # Close communication
47          writer.close()
48
49
50  async def start_serving():
51      server = await asyncio.start_server(handle_echo, "0.0.0.0", 8888)
52      print("Serving on {} port {}".format(*server.sockets[0].getsockname()))
53      return server
54
55
56  async def stop_serving(server):
57      server.close()
58      await server.wait_closed()
59
60
61  def main():
62      # Start the server
63      loop = asyncio.get_event_loop()
64      server = loop.run_until_complete(start_serving())
65      # Serve requests until Ctrl+C is pressed
66      try:
67          loop.run_forever()
68      except KeyboardInterrupt:
69          pass
70      # Close the server
71      loop.run_until_complete(stop_serving(server))
72      loop.close()
73
74
75  if __name__ == "__main__":
76      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:

```
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. 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 programming 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:

```
1  """Demo Tango Device Server using asyncio green mode"""
2
3  import asyncio
4  from tango import DevState, GreenMode
5  from tango.server import Device, command, attribute
6
7
8  class AsyncioDevice(Device):
9      green_mode = GreenMode.Asyncio
10
11      async def init_device(self):
12          await super().init_device()
13          self.set_state(DevState.ON)
14
15      @command
16      async def long_running_command(self):
17          self.set_state(DevState.OPEN)
18          await asyncio.sleep(2)
19          self.set_state(DevState.CLOSE)
20
21      @command
22      async def background_task_command(self):
23          loop = asyncio.get_event_loop()
24          future = loop.create_task(self.coroutine_target())
25
```

(continues on next page)

(continued from previous page)

```
26     async def coroutine_target(self):
27         self.set_state(DevState.INSERT)
28         await asyncio.sleep(15)
29         self.set_state(DevState.EXTRACT)
30
31     @attribute
32     async def test_attribute(self):
33         await asyncio.sleep(2)
34         return 42
35
36
37 if __name__ == "__main__":
38     AsyncioDevice.run_server()
```


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 the numpy type is used for the input arguments. Also, it is recommended to use numpy arrays of the appropriate 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 (<i>New in PyTango 8.0</i>)
DEV_VOID	No data	No data
DEV_BOOLEAN	<code>bool</code>	<code>bool</code>
DEV_SHORT	<code>int</code>	<code>int</code>
DEV_LONG	<code>int</code>	<code>int</code>
DEV_LONG64	<ul style="list-style-type: none"> <code>long</code> (on a 32 bits computer) <code>int</code> (on a 64 bits computer) 	<code>int</code>
DEV_FLOAT	<code>float</code>	<code>float</code>
DEV_DOUBLE	<code>float</code>	<code>float</code>
DEV_USHORT	<code>int</code>	<code>int</code>
DEV_ULONG	<code>int</code>	<code>int</code>
DEV_ULONG64	<ul style="list-style-type: none"> <code>long</code> (on a 32 bits computer) <code>int</code> (on a 64 bits computer) 	<code>int</code>
DEV_STRING	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
DEV_ENCODED (<i>New in PyTango 8.0</i>)	sequence of two elements: 0. <code>str</code> 1. <code>bytes</code> (for any value of <i>extract_as</i>)	sequence of two elements: 0. <code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>) 1. <code>bytes</code> (for any value of <i>extract_as</i> , except <i>String</i> . In this case it is <code>str</code> (decoded with default python encoding <i>utf-8</i>))
DEV_ENUM (<i>New in PyTango 9.0</i>)	<ul style="list-style-type: none"> <code>int</code> (for value) <code>list</code> <code><str></code> (for <code>enum_labels</code>) <p>Note: direct attribute access via DeviceProxy will return enumerated type <code>enum.IntEnum</code>. This type uses the package <code>enum34</code>.</p>	<ul style="list-style-type: none"> <code>int</code> (for value) <code>list</code> <code><str></code> (for <code>enum_labels</code>) <p>Note: direct attribute access via DeviceProxy will return enumerated type <code>enum.IntEnum</code>. Python <code>< 3.4</code>, uses the package <code>enum34</code>. Python <code>>= 3.4</code>, uses standard package <code>enum</code>.</p>

For array types (SPECTRUM/IMAGE)

Tango data type	ExtractAs	Data type (Python 2.x)	Data type (Python 3.x) (<i>New in PyTango 8.0</i>)
DEV-VAR_CHARARRAY	Numpy	<code>numpy.ndarray</code> (<code>dtype= numpy.uint8</code>)	<code>numpy.ndarray</code> (<code>dtype= numpy.uint8</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>

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-VAR_SHORTARRAY or (DEV_SHORT + SPECTRUM) or (DEV_SHORT + IM- AGE)	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int></code>	<code>list <int></code>
	Tuple	<code>tuple <int></code>	<code>tuple <int></code>
	Numpy	<code>numpy.ndarray</code> (<code>dtype= numpy. uint16</code>)	<code>numpy.ndarray</code> (<code>dtype= numpy. uint16</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
DEV-VAR_LONGARRAY or (DEV_LONG + SPECTRUM) or (DEV_LONG + IM- AGE)	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int></code>	<code>list <int></code>
	Tuple	<code>tuple <int></code>	<code>tuple <int></code>
	Numpy	<code>numpy.ndarray</code> (<code>dtype= numpy. uint32</code>)	<code>numpy.ndarray</code> (<code>dtype= numpy. uint32</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
DEV-VAR_LONG64ARRAY or (DEV_LONG64 + SPECTRUM) or (DEV_LONG64 + IMAGE)	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int></code>	<code>list <int></code>
	Tuple	<code>tuple <int></code>	<code>tuple <int></code>
	Numpy	<code>numpy.ndarray</code> (<code>dtype= numpy. uint64</code>)	<code>numpy.ndarray</code> (<code>dtype= numpy. uint64</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
DEV-VAR_FLOATARRAY or (DEV_FLOAT + SPECTRUM) or (DEV_FLOAT + IM- AGE)	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int (64 bits) / long (32 bits)></code>	<code>list <int></code>
	Tuple	<code>tuple <int (64 bits) / long (32 bits)></code>	<code>tuple <int></code>
	Numpy	<code>numpy.ndarray</code> (<code>dtype= numpy. float32</code>)	<code>numpy.ndarray</code> (<code>dtype= numpy. float32</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <float></code>	<code>list <float></code>
	Tuple	<code>tuple <float></code>	<code>tuple <float></code>

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- VAR_DOUBLEARRAY or (DEV_DOUBLE + SPECTRUM) or (DEV_DOUBLE + IMAGE)	Numpy	<code>numpy.ndarray</code> (dtype= <code>numpy.float64</code>)	<code>numpy.ndarray</code> (dtype= <code>numpy.float64</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <float></code>	<code>list <float></code>
	Tuple	<code>tuple <float></code>	<code>tuple <float></code>
DEV- VAR_USHORTARRAY or (DEV_USHORT + SPECTRUM) or (DEV_USHORT + IMAGE)	Numpy	<code>numpy.ndarray</code> (dtype= <code>numpy.uint16</code>)	<code>numpy.ndarray</code> (dtype= <code>numpy.uint16</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int></code>	<code>list <int></code>
	Tuple	<code>tuple <int></code>	<code>tuple <int></code>
DEV- VAR_ULONGARRAY or (DEV_ULONG + SPECTRUM) or (DEV_ULONG + IMAGE)	Numpy	<code>numpy.ndarray</code> (dtype= <code>numpy.uint32</code>)	<code>numpy.ndarray</code> (dtype= <code>numpy.uint32</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int></code>	<code>list <int></code>
	Tuple	<code>tuple <int></code>	<code>tuple <int></code>
DEV- VAR_ULONG64ARRAY or (DEV_ULONG64 + SPECTRUM) or (DEV_ULONG64 + IMAGE)	Numpy	<code>numpy.ndarray</code> (dtype= <code>numpy.uint64</code>)	<code>numpy.ndarray</code> (dtype= <code>numpy.uint64</code>)
	Bytes	<code>bytes</code> (which is in fact equal to <code>str</code>)	<code>bytes</code>
	ByteArray	<code>bytearray</code>	<code>bytearray</code>
	String	<code>str</code>	<code>str</code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
	List	<code>list <int (64 bits) / long (32 bits)></code>	<code>list <int></code>
	Tuple	<code>tuple <int (64 bits) / long (32 bits)></code>	<code>tuple <int></code>
DEV- VAR_STRINGARRAY or (DEV_STRING + SPECTRUM) or (DEV_STRING + IMAGE)		<code>sequence<str></code>	<code>sequence<str></code> (de- coded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)

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_LONGSTRINGAR		sequence of two elements: 0. <code>numpy.ndarray(dtype= numpy.int32)</code> or <code>sequence<int></code> 1. <code>sequence<str></code>	sequence of two elements: 0. <code>numpy.ndarray(dtype= numpy.int32)</code> or <code>sequence<int></code> 1. <code>sequence<str></code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)
DEV_DOUBLESTRING.		sequence of two elements: 0. <code>numpy.ndarray(dtype= numpy.float64)</code> or <code>sequence<int></code> 1. <code>sequence<str></code>	sequence of two elements: 0. <code>numpy.ndarray(dtype= numpy.float64)</code> or <code>sequence<int></code> 1. <code>sequence<str></code> (decoded with <i>latin-1</i> , aka <i>ISO-8859-1</i>)

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. 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
# (explicit 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,)),
```

(continues on next page)

(continued from previous page)

```

        DE8=True)
)

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

import collections

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

PIPE2 = 'BlobCase2', data

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

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

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

PIPE4 = \
('BlobCase4',
 ({'name': '1DE', 'value': ('Inner', ({'name': '1_1DE', 'value': 'Grenoble
→'},
                                     {'name': '1_2DE', 'value': (
→ 'InnerInner',
                                     ({'name
→ ': '1_1_1DE', 'value': np.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 (here we use type hints, see [Use Python type hints when declaring a device](#), but we can also set `dtype=Noon` when defining the attribute - see earlier versions of this documentation):

```
import time
from enum import IntEnum

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.ANALOG

    @attribute
    def time(self) -> float:
        return time.time()

    @attribute(max_dim_x=9)
    def gmtime(self) -> tuple[int]:
        return time.gmtime()

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

    @attribute
    def display(self) -> DisplayType:
        return self.display_type

    @display.setter
    def display(self, display_type: int):
        # note that we receive an integer, not an enum instance,
        # so we have to convert that to an instance of our enum.
        self.display_type = DisplayType(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_
```

(continues on next page)

(continued from previous page)

```

→tuple
    is not present, current time as returned by localtime() is used.
    """
    return time.ctime(seconds)

@command
def mktime(self, tupl: tuple[int]) -> float:
    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 tango

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

clock = tango.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)

```

Example output:

```

$ python client.py test/clock/1
1699433430.714272
[2023  11   8   8  50  30   2 312   0]
0 AM 0
Good morning!
Wed Nov  8 09:50:30 2023
1699429830.0
0 ANALOG 0
1 DIGITAL 1

```

(continues on next page)

(continued from previous page)

```
0 ANALOG 0
1 DIGITAL 1
```

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.AsyncResult`

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_async_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_async_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 omitted 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_async (*self*, *cmd_name*) → *id*

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

command_inout_async (*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 omitted 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 reconnection 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_async(*self*, *cmd_name*, *callback*) -> *None*

`command_inout_async(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 omitted if the command should not get any argument.

`callback`

Any callable object (function, lambda...) or any object 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_async_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 omitted if the command should not get any argument.

Return

A `DeviceData` object.

Throws

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

`command_inout_reply(self, idx, timeout=None) -> 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. If optional `timeout` parameter is not provided an exception is also thrown in case of the reply is not yet arrived. If `timeout` is provided, 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**idx**

(int) Asynchronous call identifier.

timeout

(int) (optional) Milliseconds to wait for the reply.

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***freeze_dynamic_interface** ()Prevent unknown attributes to be set on this `DeviceProxy` instance.

An exception will be raised if the Python attribute set on this `DeviceProxy` instance does not already exist. This prevents accidentally writing to a non-existent Tango attribute when using the high-level API.

This is the default behaviour since PyTango 9.3.4.

See also `tango.DeviceProxy.unfreeze_dynamic_interface()`.

New in version 9.4.0.

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_async_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 indefinitely for the reply

Parameters**call_timeout***(int)* timeout in milliseconds**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 deviceDeprecated: 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* or *sequence(names)*) → AttributeInfoListEx :

Return the extended attribute configuration for a single attribute or for the list of specified attributes. To get all the attributes pass a sequence containing the constant `tango.constants.AllAttr`.

Parameters**name**

(*str*) attribute name or (sequence(*str*)) attribute names

Return

(AttributeInfoListEx) Object containing the attribute information

Throws

ConnectionFailed, *CommunicationFailed*, *DevFailed* from device

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 and sets 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 information**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 if the device has no logging targets.

Parameters

None

Return

a sequence<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 fetched

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_dynamic_interface_frozen()

Indicates if the dynamic interface for this DeviceProxy instance is frozen.

See also `tango.DeviceProxy.freeze_dynamic_interface()` and `tango.DeviceProxy.unfreeze_dynamic_interface()`.

returns

True if the dynamic interface this DeviceProxy is frozen.

rtype

`bool`

New in version 9.4.0.

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>**). Careful, 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.

Changed in version 9.4.0: For spectrum and image attributes with an empty sequence, no longer returns *DeviceAttribute.value* and *DeviceAttribute.w_value* as *None*. Instead, *DevString* and *DevEnum* types get an empty tuple, while other types get an empty *numpy.ndarray*. Using *extract_as* can change the sequence type, but it still won't be *None*.

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

read_attribute_async (*self*, *attr_name*, *callback*) → *None*

Shortcut to *self.read_attributes_async([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_async (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_async (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

AsyncCall, *AsyncReplyNotArrived*, *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

AsyncCall, *AsyncReplyNotArrived*, *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`

Reconnect 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 Green-Mode.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 Green-Mode.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.

unfreeze_dynamic_interface()

Allow new attributes to be set on this `DeviceProxy` instance.

An exception will not be raised if the Python attribute set on this `DeviceProxy` instance does not exist. Instead, the new Python attribute will be added to the `DeviceProxy` instance's dictionary of attributes. This may be useful, but a user will not get an error if they accidentally write to a non-existent Tango attribute when using the high-level API.

See also `tango.DeviceProxy.freeze_dynamic_interface()`.

New in version 9.4.0.

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 purpose 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 check 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` Green-Mode. (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_async (*attr_name*, *value*, *cb=None*)

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

Shortcut to self.write_attributes_async([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

AsyncCall, *AsyncReplyNotArrived*, *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

AsyncCall, *AsyncReplyNotArrived*, *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 Green-Mode. (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_async (*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_async (*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

`AsyncCall`, `AsyncReplyNotArrived`, `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

`AsyncCall`, `AsyncReplyNotArrived`, `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

```
tango.get_device_proxy(self, dev_name, green_mode=None, wait=True, timeout=True) →  
DeviceProxy
```

```
tango.get_device_proxy(self, dev_name, need_check_acc, green_mode=None, wait=True,  
timeout=None) → DeviceProxy
```

Returns a new *DeviceProxy*. There is no difference between using this function and the direct *DeviceProxy* constructor if you use the default kwargs.

The added value of this function becomes evident when you choose a green_mode to be *Futures* or *Gevent* or *Asyncio*. The DeviceProxy constructor internally makes some network calls which makes it *slow*. By using one of the *green modes* as 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)
- **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`

else if `green_mode` is Futures:

`concurrent.futures.Future`

else if `green_mode` is Gevent:

`gevent.event.AsyncResult`

else if `green_mode` is Asyncio:

`asyncio.Future`

Throws

- a `DevFailed` if `green_mode` is Synchronous or `wait` is `True` and there is an error creating the device.
- a `concurrent.futures.TimeoutError` if `green_mode` is Futures, `wait` is `False`, `timeout` is not `None` and the time to create the device has expired.
- a `gevent.timeout.Timeout` if `green_mode` is Gevent, `wait` is `False`, `timeout` is not `None` and the time to create the device has expired.
- a `asyncio.TimeoutError` if `green_mode` is Asyncio, `wait` is `False`, `timeout` is not `None` and the time to create the device has expired.

New in PyTango 8.1.0

5.2.2 AttributeProxy

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

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 reimplement 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, **kws)

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, **kws)

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, **kws)`

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`, `AttrConEventData` 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, **kws)

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, **kws)

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 fetched

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*, ***kws*)

This method is a simple way to do:

```
self.get_device_proxy().get_transparency_reconnection(...)
```

For convenience, here is the documentation of *Device-Proxy.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, **kwargs)

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, **kwargs)

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, **kwargs)

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, **kws*)

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, **kws*)

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, **kws)

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>**). Careful, 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.

Changed in version 9.4.0: For spectrum and image attributes with an empty sequence, no longer returns `DeviceAttribute.value` and `DeviceAttribute.w_value` as `None`. Instead, `DevString` and `DevEnum` types get an empty `tuple`, while other types get an empty `numpy.ndarray`. Using `extract_as` can change the sequence type, but it still won't be `None`.

read_async (*args, **kws)

This method is a simple way to do:

```
self.get_device_proxy().read_attribute_async(self.name(), ...)
```

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

```
read_attribute_async( self, attr_name) -> int read_attribute_async( self,
attr_name, callback) -> None
```

Shortcut to `self.read_attributes_async([attr_name], cb)`

New in PyTango 7.0.0

read_reply (*args, **kws)

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, **kws)

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, **kws)

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, **kws)

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, **kws)

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, **kws)

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, **kwargs)

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 * *Event-*
Type.ATTR_CONF_EVENT * *Event-*
Type.DATA_READY_EVENT * *Event-*
Type.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 *Green-Mode.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, **kws)

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 check that the `event_id` has been subscribed in this `DeviceProxy`.

Return

None

Throws

EventSystemFailed, *KeyError*

write (*args, **kws)

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 Device-Proxy 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_async (*args, **kwargs)

This method is a simple way to do:

self.get_device_proxy().write_attribute_async(...)

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

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

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

New in PyTango 7.0.0

write_read (*args, **kwargs)

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, **kws)

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

AsyncCall, *AsyncReplyNotArrived*,
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

AsyncCall, *AsyncReplyNotArrived*,
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_async(...))`

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

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

command_inout_async (*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_async` 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, **kws*)

Disables a group or a device element in a group.

enable (**args, **kws*)

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
```

(continues on next page)

(continued from previous page)

```
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, **kws)

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

get_name (*args, **kws)

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, **kws)

Check if a group is enabled. *New in PyTango 7.0.0*

name_equals (*args, **kws)

New in PyTango 7.0.0

name_matches (*args, **kws)

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_async(...))`

read_attribute_async (*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_async`.

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_async(...))`

read_attributes_async (*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_async`.

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_async(...))`

write_attribute_async (*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_async.

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.asyncio.DeviceProxy()`
- `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 behavior of other libraries and/or application where your library is being.

Parameters

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

`tango.asyncio.DeviceProxy(self, dev_name, wait=False, timeout=None)`

-> DeviceProxy

DeviceProxy(self, dev_name, need_check_acc, wait=False, timeout=None)

-> DeviceProxy

Creates a *asyncio* enabled DeviceProxy.

The DeviceProxy constructor internally makes some network calls which makes it *slow*. By using the *asyncio green mode* you may give the control back to the *asyncio* event loop using the *yield from* or *await* syntax.

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.
- an *asyncio.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.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.AsyncResult

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 synchronization 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_async_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_async_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_async_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

[AsyncReplyNotArrived](#). 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_async_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_ASYNC`: Return all asynchronous request number

Parameters

req
(*async_req_type*) asynchronous request type

Return

(*int*) the number of pending requests for the given type

New in PyTango 7.1.3

set_async_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 following 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 following 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 following 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 following 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 following 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 following 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 following 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 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_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
- 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_ASYNC

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

totatetime (*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

```
1 from time import time
2 from numpy.random import random_sample
3
4 from tango import AttrQuality, AttrWriteType, DispLevel
5 from tango.server import Device, attribute, command
6 from tango.server import class_property, device_property
7
8 class PowerSupply(Device):
9
10     voltage = attribute()
11
12     current = attribute(label="Current", dtype=float,
13                        display_level=DispLevel.EXPERT,
14                        access=AttrWriteType.READ_WRITE,
15                        unit="A", format="8.4f",
16                        min_value=0.0, max_value=8.5,
17                        min_alarm=0.1, max_alarm=8.4,
18                        min_warning=0.5, max_warning=8.0,
19                        fget="get_current", fset="set_current",
20                        doc="the power supply current")
21
22     noise = attribute(label="Noise", dtype=((float,)),
23                      max_dim_x=1024, max_dim_y=1024,
24                      fget="get_noise")
25
26     host = device_property(dtype=str)
27     port = class_property(dtype=int, default_value=9788)
28
29     def read_voltage(self):
30         self.info_stream("get voltage(%s, %d)" % (self.host, self.port))
31         return 10.0
32
33     def get_current(self):
34         return 2.3456, time(), AttrQuality.ATTR_WARNING
35
36     def set_current(self, current):
37         print("Current set to %f" % current)
38
39     def get_noise(self):
40         return random_sample((1024, 1024))
41
42     @command(dtype_in=float)
43     def ramp(self, value):
44         print("Ramping up...")
45
46 if __name__ == "__main__":
47     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
<code>None</code>	<code>DevVoid</code>
<code>'None'</code>	<code>DevVoid</code>
<code>DevVoid</code>	<code>DevVoid</code>
<code>'DevVoid'</code>	<code>DevVoid</code>
<code>DevState</code>	<code>DevState</code>
<code>'DevState'</code>	<code>DevState</code>
<code>bool</code>	<code>DevBoolean</code>
<code>'bool'</code>	<code>DevBoolean</code>
<code>'boolean'</code>	<code>DevBoolean</code>
<code>DevBoolean</code>	<code>DevBoolean</code>
<code>'DevBoolean'</code>	<code>DevBoolean</code>
<code>numpy.bool_</code>	<code>DevBoolean</code>
<code>'char'</code>	<code>DevUChar</code>
<code>'chr'</code>	<code>DevUChar</code>
<code>'byte'</code>	<code>DevUChar</code>
<code>chr</code>	<code>DevUChar</code>
<code>DevUChar</code>	<code>DevUChar</code>
<code>'DevUChar'</code>	<code>DevUChar</code>
<code>numpy.uint8</code>	<code>DevUChar</code>
<code>'int16'</code>	<code>DevShort</code>
<code>DevShort</code>	<code>DevShort</code>
<code>'DevShort'</code>	<code>DevShort</code>
<code>numpy.int16</code>	<code>DevShort</code>
<code>'uint16'</code>	<code>DevUShort</code>
<code>DevUShort</code>	<code>DevUShort</code>

continues on next page

Table 2 – continued from previous page

dtype argument	converts to tango type
'DevUShort '	DevUShort
<code>numpy.uint16</code>	DevUShort
'int32 '	DevLong
DevLong	DevLong
'DevLong '	DevLong
<code>numpy.int32</code>	DevLong
'uint32 '	DevULong
DevULong	DevULong
'DevULong '	DevULong
<code>numpy.uint32</code>	DevULong
<code>int</code>	DevLong64
'int '	DevLong64
'int64 '	DevLong64
DevLong64	DevLong64
'DevLong64 '	DevLong64
<code>numpy.int64</code>	DevLong64
'uint '	DevULong64
'uint64 '	DevULong64
DevULong64	DevULong64
'DevULong64 '	DevULong64
<code>numpy.uint64</code>	DevULong64
'float32 '	DevFloat
DevFloat	DevFloat
'DevFloat '	DevFloat
<code>numpy.float32</code>	DevFloat
<code>float</code>	DevDouble
'double '	DevDouble
'float '	DevDouble
'float64 '	DevDouble
DevDouble	DevDouble
'DevDouble '	DevDouble
<code>numpy.float64</code>	DevDouble
<code>str</code>	DevString
'str '	DevString
'string '	DevString
'text '	DevString
DevString	DevString
'DevString '	DevString
<code>bytearray</code>	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
DevVarLong64Array	DevVarLong64Array
'DevVarLong64Array '	DevVarLong64Array
DevVarULong64Array	DevVarULong64Array
'DevVarULong64Array '	DevVarULong64Array

continues on next page

Table 2 – continued from previous page

dtype argument	converts to tango type
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.

If you pass a reference to unbound method for read, write or is_allowed method (e.g. DeviceClass.read_function or self.__class__.read_function), during execution the corresponding bound method (self.read_function) will be used.

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*, *source=None*)

Sends the given message to the tango debug stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the debug stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added source parameter

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 the 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 fulfill 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 fulfill 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*, *source=None*)

Sends the given message to the tango error stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the error stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter

fatal_stream (*self*, *msg*, **args*, *source=None*)

Sends the given message to the tango fatal stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the fatal stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter

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 typeSequence[`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 typeSequence[`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 typeSequence[`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 typeSequence[`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*, *source=None*)

Sends the given message to the tango info stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the info stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter

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 initialization 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.

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.

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*)

Push a data ready event for the given attribute name.

The method needs the attribute name and a "counter" which will be passed 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.

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
- **free_it** (*bool*) – free Attr object flag. Default False
- **clean_db** (*bool*) – clean attribute related info in db. Default True

Raises

DevFailed –

New in version 9.5.0: *free_it* parameter. *clean_db* parameter.

remove_command (*self*, *cmd_name*, *free_it=False*, *clean_db=True*)

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.

server_init_hook ()

Tango `server_init_hook`. Called once the device server admin device (`DServer`) is exported. Default implementation does nothing.

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 fulfilled. 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 fulfilled. 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*, *source=None*)

Sends the given message to the tango warn stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the warn stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added source parameter

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()
```

(continues on next page)

(continued from previous page)

```
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	<i>AttrDataFormat</i>	SCALAR	data format
max_dim_x	int	1	maximum size for x dimension (ign
max_dim_y	int	0	maximum size for y dimension (ign
display_level	<i>DispLevel</i>	OPERATOR	display level
polling_period	int	-1	polling period
memorized	bool	False	attribute must be memorized (only
hw_memorized	bool	False	memorized value will be restored b
access	<i>AttrWriteType</i>	READ	read only / read write / write only a
fget (or fread)	str or callable	'read_<attr_name>'	read method name or method objec
fset (or fwrite)	str or callable	'write_<attr_name>'	write method name or method obje
isallowed	str or callable	'is_<attr_name>_allowed'	is allowed method name or method
label	str	'<attr_name>'	attribute label
enum_labels	sequence	None	the list of enumeration labels (enum
doc (or description)	str	''	attribute description
unit	str	''	physical units the attribute value is
standard_unit	str	''	physical standard unit
display_unit	str	''	physical display unit (hint for client
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
max_alarm	str	None	maximum value to trigger attribute
min_warning	str	None	minimum value to trigger attribute
max_warning	str	None	maximum value to trigger attribute
delta_val	str	None	
delta_t	str	None	
abs_change	str	None	minimum value change between ev
rel_change	str	None	minimum relative change between
period	str	None	
archive_abs_change	str	None	
archive_rel_change	str	None	
archive_period	str	None	
green_mode	bool	True	Default green mode for read/write
read_green_mode	bool	'green_mode' value	green mode for read function. If Tru
write_green_mode	bool	'green_mode' value	green mode for write function. If Tr
isallowed_green_mode	bool	'green_mode' value	green mode for is allowed function.
forwarded	bool	False	the attribute should be forwarded i

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

    @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, fisallowed=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):
```

(continues on next page)

(continued from previous page)

```

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.
- **fisallowed** (*str or callable*) – is allowed method for command

New in version 8.1.7: added `green_mode` option

New in version 9.2.0: added `display_level` and `polling_period` optional argument

New in version 9.4.0: added `fisallowed` option

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},

```

(continues on next page)

(continued from previous page)

```
{'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
isallowed	str or callable	'is_<pipe_name>	is allowed method name or method object
label	str	'<pipe_name>	pipe label
doc (or description)	str	""	pipe description
green_mode	bool	True	Default green mode for read/write/isallowed functions. If True: run with green mode executor, if False: run directly
read_green	bool	'green_mode' value	green mode for read function. If True: run with green mode executor, if False: run directly
write_green	bool	'green_mode' value	green mode for write function. If True: run with green mode executor, if False: run directly
isallowed_green	bool	'green_mode' value	green mode for is allowed function. If True: run with green mode executor, if False: run directly

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.

New in version 9.4.0: added `isallowed_green_mode` option

```
class tango.server.device_property(dtype=None, 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=None, 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,
pre_init_callback=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.

Examples

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)})
```

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.

Parameters

- **classes** (*Sequence[tango.server.Device] / dict*) – Defines for which Tango Device Classes the server will run. If *dict* is provided, it's key is the tango class name and value is either:
 - Device*
 - two element sequence: *DeviceClass*, *DeviceImpl*
 - three element sequence: *DeviceClass*, *DeviceImpl*, tango class name *str*
- **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
- **pre_init_callback** (*callable or tuple*) – an optional callback that is executed between the calls `Util.init` and `Util.server_init` The optional *pre_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).
- **post_init_callback** (*callable or tuple*) – an optional callback that is executed between the calls `Util.server_init` and `Util.server_run` 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).
- **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

Changed in version 9.5.0: *pre_init_callback* 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,
pre_init_callback=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.

Changed in version 9.5.0: `pre_init_callback` argument has been added

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.

If you pass a reference to unbound method for read, write or `is_allowed` method (e.g. `DeviceClass.read_function` or `self.__class__.read_function`), during execution the corresponding bound method (`self.read_function`) will be used.

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 *isallowed* 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*, *source=None*)

Sends the given message to the tango debug stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the debug stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

*New in version 9.4.2: added source parameter***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 the 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 fulfill 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 fulfill 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*, *source*=None)

Sends the given message to the tango error stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the error stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter

fatal_stream (*self*, *msg*, **args*, *source*=None)

Sends the given message to the tango fatal stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the fatal stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter

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 typeSequence[[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 typeSequence[[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 typeSequence[[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 typeSequence[[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*, *source*=None)

Sends the given message to the tango info stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the info stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter

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.

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.

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*)

Push a data ready event for the given attribute name.

The method needs the attribute name and a “counter” which will be passed 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.

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
- **free_it** (*bool*) – free Attr object flag. Default False
- **clean_db** (*bool*) – clean attribute related info in db. Default True

Raises

DevFailed –

New in version 9.5.0: *free_it* parameter. *clean_db* parameter.

remove_command (*self*, *cmd_name*, *free_it=False*, *clean_db=True*)

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 –

server_init_hook (*self*)

Hook method.

This method is called once the device server admin device is exported. This allows for instance for the different devices to subscribe to events at server startup on attributes from other devices of the same device server with stateless parameter set to false.

This method can be redefined in sub-classes in case of the default behaviour does not fullfill the needs

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*, *source=None*)

Sends the given message to the tango warn stream.

Since PyTango 7.1.3, the same can be achieved with:

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

Parameters

- **msg** (*str*) – the message to be sent to the warn stream
- ***args** – Arguments to format a message string.
- **source** (*Callable*) – Function that will be inspected for filename and lineno in the log message.

New in version 9.4.2: added *source* parameter**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

add_wiz_class_prop (*self*, *name*, *desc*) → *None***add_wiz_class_prop** (*self*, *name*, *desc*, *def*) → *None*

For internal usage only

Parameters

- **name** (*str*) – class property name
- **desc** (*str*) – class property description
- **def** (*str*) – class property default value

Returns*None***add_wiz_dev_prop** (*self*, *name*, *desc*) → *None*

add_wiz_dev_prop (*self*, *name*, *desc*, *def*) → *None*

For internal usage only

Parameters

- **name** (*str*) – device property name
- **desc** (*str*) – device property description
- **def** (*str*) – device property default value

Returns

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 (*str*) – 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 wish 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 fulfilled.

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 fulfilled.

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_level*)

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).

By default the setpoint will be written to the attribute during initialisation! Use method `set_memorized_init()` with `False` as argument if you don't want this feature.

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 fulfilled.

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 fulfilled. 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 files in the Attribute-Config/ 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 (**args*)

set_value (*self, data*)

set_value (*self, str_data, data*)

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

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.

Parameters

- **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.

- **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*) – [DEPRECATED] the attribute x length. Default value is 1
- **dim_y** (*int*) – [DEPRECATED] the attribute y length. Default value is 0

set_value_date_quality (*args)

set_value_date_quality (self, data, time_stamp, quality)

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

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

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.

Parameters

- **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.
- **str_data** (*str*) – special variation for DevEncoded data type. In this case ‘data’ must be a str or an object with the buffer interface.
- **time_stamp** (*double*) – the time stamp
- **quality** (*AttrQuality*) – the attribute quality factor
- **dim_x** (*int*) – [DEPRECATED] the attribute x length. Default value is 1
- **dim_y** (*int*) – [DEPRECATED] the attribute y length. Default value is 0

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

Retrieve the new value for writable attribute.

Parameters

extract_as (*ExtractAs*) – defaults to *ExtractAs.Numpy*

Returns

the attribute write value.

Return type

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.

set_write_value (*self*, *data*, *dim_x*=1, *dim_y*=0)

Set the writable attribute value.

Parameters

- **data** – the data to be set. 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
- **dim_x** (*int*) – optional, the attribute set value x length
- **dim_y** (*int*) – optional, the attribute set value y length

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 message 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 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 maximum 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 responsibility 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():
```

(continues on next page)

(continued from previous page)

```
util = 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 maximum 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 triggered”

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 triggered”

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*, *servername*, *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_infol = DbDevInfo()
dev_infol.name = 'my/own/device'
dev_infol._class = 'MyDevice'
dev_infol.server = 'MyServer/test'
db.add_server(dev_infol.server, dev_infol, 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 specified 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.iior = <the real iior>
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 specified 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 specified 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 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 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 database 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.iior)
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 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 the '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: 'l-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 specified 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 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 specified 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 instantiated 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 database 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.iior)
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 specified 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 within 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):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.int16)
    data = numpy.array((data,data,data))
    enc.encode_gray16(data)
    return 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):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.byte)
    data = numpy.array((data,data,data))
    enc.encode_gray8(data)
    return 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):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.byte)
    data = numpy.array((data,data,data))
    enc.encode_jpeg_gray8(data)
    return 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 containing 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):
    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)
    return 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 containing 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`)
-

Note: Encoding with transparency information required `cppTango` built with `TANGO_USE_JPEG` options see installation instructions of `cppTango`

Example

:

```
def read_myattr(self):
    enc = tango.EncodedAttribute()
    data = numpy.arange(100, dtype=numpy.int32)
    data = numpy.array((data,data,data))
    enc.encode_jpeg_rgb32(data)
    return enc
```

encode_rgb24 (*rgb24*, *width=0*, *height=0*)

Encode a 24 bit color image (no compression)

param rgb24

an object containing 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):
    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)
    return 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 Dev-Long 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_name='MyDS')
```

Parameters

- **min_version** (*None*, *str*, *Version*) – minimum PyTango version [default: *None*, meaning no minimum required]. If a string is given, it must be in the valid version number format (see: *Version*)
- **conflicts** (*seq*<*str*/*Version*>) – 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 cppTango 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_name='MyDS')
```

Parameters

- **min_version** (*None*, *str*, *Version*) – minimum Tango version [default: *None*, meaning no minimum required]. If a string is given, it must be in the valid version number format (see: *Version*)
- **conflicts** (*seq*<*str*/*Version*>) – 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.

```
1 import tango
2
3 # How to protect the script from exceptions raised by the Tango
4 try:
5     # Get proxy on a non existing device should throw an exception
6     device = tango.DeviceProxy("non/existing/device")
7 except DevFailed as df:
8     print("Failed to create proxy to non/existing/device:\n%s" % df)
```


5.7.2 Throwing exception in a device server

The C++ `tango::Except` class with its most important methods have been wrapped to Python. Therefore, in a Python device server, you have the following methods to throw, re-throw or print a `Tango::DevFailed` exception :

- `throw_exception()` which is a static method
- `re_throw_exception()` which is also a static method
- `print_exception()` which is also a static method

The following code is an example of a command method requesting a command on a sub-device and re-throwing the exception in case of:

```

1 try:
2     dev.command_inout("SubDevCommand")
3 except tango.DevFailed as df:
4     tango.Except.re_throw_exception(df,
5         "MyClass_CommandFailed",
6         "Sub device command SubdevCommand failed",
7         "Command() ")

```

line 2

Send the command to the sub device in a try/catch block

line 4-6

Re-throw the exception and add a new level of information in the exception stack

5.7.3 Exception API

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

Bases:

A container 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 severity (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_DeviceTimedOu	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_CommunicationF	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 subscribed 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 argument to the `write_attributes()` method
- The error stack

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__
'9.5.0'
>>> tango.__version_info__
(9, 5, 0)
```

and the Tango C++ library version that PyTango was compiled with:

```
>>> import tango
>>> tango.constants.TgLibVers
'9.5.0'
```

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 `--port`, 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:

```

1 from tango import DeviceProxy
2
3 # Get proxy on the tango_test1 device
4 print("Creating proxy to TangoTest device...")
5 tango_test = DeviceProxy("sys/tg_test/1")
6
7 # ping it
8 print(tango_test.ping())
9
10 # get the state
11 print(tango_test.state())

```

6.7 Read and write attributes

Basic read/write attribute operations:

```

1 from tango import DeviceProxy
2
3 # Get proxy on the tango_test1 device
4 print("Creating proxy to TangoTest device...")
5 tango_test = DeviceProxy("sys/tg_test/1")
6
7 # Read a scalar attribute. This will return a tango.DeviceAttribute
8 # Member 'value' contains the attribute value
9 scalar = tango_test.read_attribute("long_scalar")
10 print(f"Long_scalar value = {scalar.value}")
11
12 # PyTango provides a shorter way:
13 scalar = tango_test.long_scalar
14 print(f"Long_scalar value = {scalar}")
15
16 # Read a spectrum attribute
17 spectrum = tango_test.read_attribute("double_spectrum")
18 # ... or, the shorter version:
19 spectrum = tango_test.double_spectrum
20
21 # Write a scalar attribute
22 scalar_value = 18
23 tango_test.write_attribute("long_scalar", scalar_value)
24
25 # PyTango provides a shorter way:
26 tango_test.long_scalar = scalar_value
27
28 # Write a spectrum attribute
29 spectrum_value = [1.2, 3.2, 12.3]
30 tango_test.write_attribute("double_spectrum", spectrum_value)
31 # ... or, the shorter version:
32 tango_test.double_spectrum = spectrum_value
33
34 # Write an image attribute
35 image_value = [ [1, 2], [3, 4] ]

```

(continues on next page)

(continued from previous page)

```
36 tango_test.write_attribute("long_image", image_value)
37 # ... or, the shorter version:
38 tango_test.long_image = image_value
```

Note that the values got when reading a spectrum or an image are 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:

```
1 import numpy
2 from tango import DeviceProxy
3
4 # Get proxy on the tango_test1 device
5 print("Creating proxy to TangoTest device...")
6 tango_test = DeviceProxy("sys/tg_test/1")
7
8 data_1d_long = numpy.arange(0, 100, dtype=numpy.int32)
9
10 tango_test.long_spectrum = data_1d_long
11
12 data_2d_float = numpy.zeros((10,20), dtype=numpy.float64)
13
14 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 automatically manages the data types, and writing scripts is quite easy:

```
1 from tango import DeviceProxy
2
3 # Get proxy on the tango_test1 device
4 print("Creating proxy to TangoTest device...")
5 tango_test = DeviceProxy("sys/tg_test/1")
6
7 # First use the classical command_inout way to execute the DevString_
8   ↪ command
9 # (DevString in this case is a command of the Tango_Test device)
10
11 result = tango_test.command_inout("DevString", "First hello to device")
12 print(f"Result of execution of DevString command = {result}")
13
14 # the same can be achieved with a helper method
15 result = tango_test.DevString("Second Hello to device")
16 print(f"Result of execution of DevString command = {result}")
17
18 # Please note that argin argument type is automatically managed by python
19 result = tango_test.DevULong(12456)
20 print(f"Result of execution of DevULong command = {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:

```

1 from tango import DeviceProxy
2
3 # Get proxy on the tango_test1 device
4 print("Creating proxy to TangoTest device...")
5 tango_test = DeviceProxy("sys/tg_test/1")
6
7 # The input argument is a DevVarLongStringArray so create the argin
8 # variable containing an array of longs and an array of strings
9 argin = ([1,2,3], ["Hello", "TangoTest device"])
10
11 result = tango_test.DevVarLongStringArray(argin)
12 print(f"Result of execution of DevVarLongArray command = {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:

```

1 from tango import Database, DbDevInfo
2
3 # A reference on the DataBase
4 db = Database()
5
6 # The 3 devices name we want to create
7 # Note: these 3 devices will be served by the same DServer
8 new_device_name1 = "px1/tdl/mouse1"
9 new_device_name2 = "px1/tdl/mouse2"
10 new_device_name3 = "px1/tdl/mouse3"
11
12 # Define the Tango Class served by this DServer
13 new_device_info_mouse = DbDevInfo()
14 new_device_info_mouse._class = "Mouse"
15 new_device_info_mouse.server = "ds_Mouse/server_mouse"
16
17 # add the first device

```

(continues on next page)

(continued from previous page)

```

18 print("Creating device: %s" % new_device_name1)
19 new_device_info_mouse.name = new_device_name1
20 db.add_device(new_device_info_mouse)
21
22 # add the next device
23 print("Creating device: %s" % new_device_name2)
24 new_device_info_mouse.name = new_device_name2
25 db.add_device(new_device_info_mouse)
26
27 # add the third device
28 print("Creating device: %s" % new_device_name3)
29 new_device_info_mouse.name = new_device_name3
30 db.add_device(new_device_info_mouse)

```

6.12.1 Setting up device properties

A more complex example using python subtleties. 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:

```

1 from tango import DeviceProxy
2
3 # connecting to the motor axis device
4 axis1 = DeviceProxy("microxas/motorisation/galilbox")
5
6 # Getting Device Properties
7 property_names = ["AxisBoxAttachement",
8                  "AxisEncoderType",
9                  "AxisNumber",
10                 "CurrentAcceleration",
11                 "CurrentAccuracy",
12                 "CurrentBacklash",
13                 "CurrentDeceleration",
14                 "CurrentDirection",
15                 "CurrentMotionAccuracy",
16                 "CurrentOvershoot",
17                 "CurrentRetry",
18                 "CurrentScale",
19                 "CurrentSpeed",
20                 "CurrentVelocity",
21                 "EncoderMotorRatio",
22                 "logging_level",
23                 "logging_target",
24                 "UserEncoderRatio",
25                 "UserOffset"]
26
27 axis_properties = axis1.get_property(property_names)
28 for prop in axis_properties.keys():
29     print("%s: %s" % (prop, axis_properties[prop][0]))
30
31 # Changing Properties
32 axis_properties["AxisBoxAttachement"] = ["microxas/motorisation/galilbox"]
33 axis_properties["AxisEncoderType"] = ["1"]
34 axis_properties["AxisNumber"] = ["6"]
35 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_CallTimeout`, `unidentifiable C++ exception`. Therefore, when you start a new process you must reset CORBA connection:

```

1  import time
2  import tango
3
4  from multiprocessing import Process
5
6
7  class Worker(Process):
8
9      def __init__(self):
10         Process.__init__(self)
11
12     def run(self):
13         # reset CORBA connection
14         tango.ApiUtil.cleanup()
15
16         proxy = tango.DeviceProxy('test/tserver/1')
17
18         stime = time.time()
19         etime = stime
20         while etime - stime < 1.:
21             try:
22                 proxy.read_attribute("Value")
23             except Exception as e:
24                 print(str(e))
25             etime = time.time()
26
27
28  def runworkers():
29     workers = [Worker() for _ in range(6)]
30     for wk in workers:
31         wk.start()
32     for wk in workers:
33         wk.join()
34
35
36  db = tango.Database()
37  dp = tango.DeviceProxy('test/tserver/1')
38
39  for i in range(4):
40     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 Multithreading - clients and servers

When performing Tango I/O from user-created threads, there can be problems. This is often more noticeable with event subscription/unsubscription, and when pushing events, but it could affect any Tango I/O.

A client subscribing and unsubscribing to events via a user thread may see a crash, a deadlock, or Event channel is not responding anymore errors.

A device server pushing events from a user-created thread (including asyncio callbacks) might see Not able to acquire serialization (dev, class or process) monitor errors.

As PyTango wraps the cppTango library, we need 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 9.4.1) 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 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:

```

1  import tango
2  from threading import Thread
3  from time import sleep
4
5
6  def thread_task():
7      with tango.EnsureOmniThread():
8          eid = dp.subscribe_event(
9              "double_scalar", tango.EventType.PERIODIC_EVENT, cb)
10         while running:
11             print(f"num events stored {len(cb.get_events())}")
12             sleep(1)
13         dp.unsubscribe_event(eid)
14
15
16 cb = tango.utils.EventCallback() # print events to stdout
17 dp = tango.DeviceProxy("sys/tg_test/1")
18 dp.poll_attribute("double_scalar", 1000)
19 thread = Thread(target=thread_task)
20 running = True
21 thread.start()
22 sleep(5)
23 running = False
24 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, device_property, attribute, command, pipe
3
4
5  class Clock(Device):
6
7      model = device_property(dtype=str)
8
9      @attribute
10     def time(self):
11         return time.time()
12
13     @command(dtype_in=str, dtype_out=str)
14     def strftime(self, format):
15         return time.strftime(format)
16
17     @pipe
18     def info(self):
19         return ('Information',
20                dict(manufacturer='Tango',
21                    model=self.model,
22                    version_number=123))
23
24
25  if __name__ == "__main__":
26     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
definition of the *model* property. Check the `device_property` for the complete list of options

line 9-11
definition of the *time* attribute. By default, attributes are double, scalar, read-only. Check the `attribute` for the complete list of attribute options.

line 13-15

the method *strftime* is exported as a Tango command. It 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 17-22

definition of the *info* pipe. Check the *pipe* for the complete list of pipe options.

line 26

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 *host* device property
2. a *port* class property
3. the standard initialisation method called *init_device*
4. a read/write double scalar expert attribute *current*
5. a read-only double scalar attribute called *voltage*
6. a read-only double image attribute called *noise*
7. a read/write float scalar attribute *range*, defined with pythonic-style decorators, which can be always read, but conditionally written
8. a read/write float scalar attribute *compliance*, defined with alternative decorators
9. an *output_on_off* command

```
1 from time import time
2 from numpy.random import random_sample
3
4 from tango import AttrQuality, AttrWriteType, DevState, DispLevel, _
   ↳ AttrReqType
5 from tango.server import Device, attribute, command
6 from tango.server import class_property, device_property
7
8
9 class PowerSupply(Device):
10     _my_current = 2.3456
11     _my_range = 0.0
12     _my_compliance = 0.0
13     _output_on = False
14
15     host = device_property(dtype=str)
16     port = class_property(dtype=int, default_value=9788)
17
18     def init_device(self):
19         super().init_device()
20         self.info_stream(f"Power supply connection details: {self.host}:
   ↳ {self.port}")
21         self.set_state(DevState.ON)
22         self.set_status("Power supply is ON")
23
24     current = attribute(
25         label="Current",
26         dtype=float,
```

(continues on next page)

(continued from previous page)

```

27         display_level=DispLevel.EXPERT,
28         access=AttrWriteType.READ_WRITE,
29         unit="A",
30         format="8.4f",
31         min_value=0.0,
32         max_value=8.5,
33         min_alarm=0.1,
34         max_alarm=8.4,
35         min_warning=0.5,
36         max_warning=8.0,
37         fget="get_current",
38         fset="set_current",
39         doc="the power supply current",
40     )
41
42     noise = attribute(
43         label="Noise",
44         dtype=((float,)),
45         max_dim_x=1024,
46         max_dim_y=1024,
47         fget="get_noise",
48     )
49
50     @attribute
51     def voltage(self):
52         return 10.0
53
54     def get_current(self):
55         return self._my_current
56
57     def set_current(self, current):
58         print("Current set to %f" % current)
59         self._my_current = current
60
61     def get_noise(self):
62         return random_sample((1024, 1024))
63
64     range = attribute(label="Range", dtype=float)
65
66     @range.setter
67     def range(self, new_range):
68         self._my_range = new_range
69
70     @range.getter
71     def current_range(self):
72         return self._my_range, time(), AttrQuality.ATTR_WARNING
73
74     @range.is_allowed
75     def can_range_be_changed(self, req_type):
76         if req_type == AttReqType.WRITE_REQ:
77             return not self._output_on
78         return True
79
80     compliance = attribute(label="Compliance", dtype=float)
81
82     @compliance.read

```

(continues on next page)

(continued from previous page)

```

83     def compliance(self):
84         return self._my_compliance
85
86     @compliance.write
87     def new_compliance(self, new_compliance):
88         self._my_compliance = new_compliance
89
90     @command(dtype_in=bool, dtype_out=bool)
91     def output_on_off(self, on_off):
92         self._output_on = on_off
93         return self._output_on
94
95
96 if __name__ == "__main__":
97     PowerSupply.run_server()

```

6.16 Use Python type hints when declaring a device

Note: This is an experimental feature, API may change in further releases!

Starting from PyTango 9.5.0 the data type of properties, attributes and commands in high-level API device servers can be declared using Python type hints.

This is the same simple *PowerSupply* device server, but using type hints in various ways:

```

1  from time import time
2  from numpy.random import random_sample
3
4  from tango import AttrQuality, AttrWriteType, DevState, DispLevel, _
   ↳AttrReqType
5  from tango.server import Device, attribute, command
6  from tango.server import class_property, device_property
7
8
9  class PowerSupply(Device):
10     _my_current = 2.3456
11     _my_range = 0
12     _my_compliance = 0.0
13     _output_on = False
14
15     host: str = device_property()
16     port: int = class_property(default_value=9788)
17
18     def init_device(self):
19         super().init_device()
20         self.info_stream(f"Power supply connection details: {self.host}:
   ↳{self.port}")
21         self.set_state(DevState.ON)
22         self.set_status("Power supply is ON")
23
24     current: float = attribute(
25         label="Current",
26         display_level=DispLevel.EXPERT,

```

(continues on next page)

(continued from previous page)

```

27         access=AttrWriteType.READ_WRITE,
28         unit="A",
29         format="8.4f",
30         min_value=0.0,
31         max_value=8.5,
32         min_alarm=0.1,
33         max_alarm=8.4,
34         min_warning=0.5,
35         max_warning=8.0,
36         fget="get_current",
37         fset="set_current",
38         doc="the power supply current",
39     )
40
41     noise: list[list[float]] = attribute(
42         label="Noise", max_dim_x=1024, max_dim_y=1024, fget="get_noise"
43     )
44
45     @attribute
46     def voltage(self) -> float:
47         return 10.0
48
49     def get_current(self):
50         return self._my_current
51
52     def set_current(self, current):
53         print("Current set to %f" % current)
54         self._my_current = current
55
56     def get_noise(self):
57         return random_sample((1024, 1024))
58
59     range = attribute(label="Range")
60
61     @range.getter
62     def current_range(self) -> tuple[float, float, AttrQuality]:
63         return self._my_range, time(), AttrQuality.ATTR_WARNING
64
65     @range.setter
66     def range(self, new_range: float):
67         self._my_range = new_range
68
69     @range.is_allowed
70     def can_range_be_changed(self, req_type):
71         if req_type == AttReqType.WRITE_REQ:
72             return not self._output_on
73         return True
74
75     compliance = attribute(label="Compliance")
76
77     @compliance.read
78     def compliance(self) -> float:
79         return self._my_compliance
80
81     @compliance.write
82     def new_compliance(self, new_compliance: float):

```

(continues on next page)

(continued from previous page)

```

83         self._my_compliance = new_compliance
84
85     @command
86     def output_on_off(self, on_off: bool) -> bool:
87         self._output_on = on_off
88         return self._output_on
89
90
91 if __name__ == "__main__":
92     PowerSupply.run_server()

```

Note: To defining DevEncoded attribute you can use type hints `tuple[str, bytes]` and `tuple[str, bytearray]` (or `tuple[str, bytes, float, AttrQuality]` and `tuple[str, bytearray, float, AttrQuality]`).

Type hints `tuple[str, str]` (or `tuple[str, str, float, AttrQuality]`) will be recognized as SPECTRUM DevString attribute with `max_dim_x=2`

If you want to create DevEncoded attribute with `(str, str)` return you have to use dtype kwarg

Properties

To define device property you can use:

```
host: str = device_property()
```

If you want to create list property you can use `tuple[], list[]` or `numpy.typing.NDArray[]` annotation:

```
channels: tuple[int] = device_property()
```

or

```
channels: list[int] = device_property()
```

or

```
channels: numpy.typing.NDArray[np.int_] = device_property()
```

Attributes For the attributes you can use one of the following patterns:

```
voltage: float = attribute()
```

or

```

voltage = attribute()

def read_voltage(self) -> float:
    return 10.0

```

or

```

voltage = attribute(fget="query_voltage")

def query_voltage(self) -> float:
    return 10.0

```

or

```
@attribute
def voltage(self) -> float:
    return 10.0
```

For writable (`AttrWriteType.READ_WRITE` and `AttrWriteType.WRITE`) attributes you can also define the type in write functions.

Note: Defining the type hint of a `READ_WRITE` attribute *only* in the write function is not recommended as it can lead to inconsistent code.

```
data_to_save = attribute(access=AttrWriteType.WRITE)

# since WRITE attribute can have only write method,
# its type can be defined here
def write_data_to_save(self, data: float)
    self._hardware.save(value)
```

Note: If you provide a type hint in several places (e.g., `dtype` kwarg and read function): there is no check, that types are the same and attribute type will be taken according to the following priority:

1. `dtype` kwarg
2. attribute assignment
3. read function
4. write function

E.g., if you create the following attribute:

```
voltage: int = attribute(dtype=float)

def read_voltage(self) -> str:
    return 10
```

the attribute type will be float

SPECTRUM and IMAGE attributes

As for the case of properties, the SPECTRUM and IMAGE attributes can be defined by `tuple[], list[]` or `numpy.typing.NDArray[]` annotation.

Note: Since there isn't yet official support for `numpy.typing.NDArray[]` shape definitions (as at 12 October 2023: <https://github.com/numpy/numpy/issues/16544>) you **must** provide a `dformat` kwarg as well as `max_dim_x` (and, if necessary, `max_dim_y`):

```
@attribute(dformat=AttrDataFormat.SPECTRUM, max_dim_x=3)
def get_time(self) -> numpy.typing.NDArray[np.int_]:
    return hours, minutes, seconds
```

In case of `tuple[], list[]` you can automatically specify attribute dimension:

```
@attribute
def get_time(self) -> tuple[int, int, int]:
    return hours, minutes, seconds
```

or you can use `max_x_dim(max_y_dim)` kwarg with just one element in tuple/list:

```
@attribute(max_x_dim=3)
def get_time(self) -> list[int]: # can be also tuple[int]
    return hours, minutes, seconds
```

Note: If you provide both `max_x_dim(max_y_dim)` kwarg and use `tuple[]` annotation, kwarg will have priority

Note: Mixing element types within a `spectrum(image)` attribute definition is not supported by Tango and will raise a `RuntimeError`.

e.g., attribute

```
@attribute(max_x_dim=3)
def get_time(self) -> tuple[float, str]:
    return hours, minutes, seconds
```

will result in `RuntimeError`

Dimension of SPECTRUM attributes can be also taken from annotation:

```
@attribute()
def not_matrix(self) -> tuple[tuple[bool, bool], tuple[bool, bool]]:
    return [[False, True], [True, False]]
```

Note: `max_y` will be len of outer tuple (or list), `max_x` - len of the inner. Note, that all inner tuples(lists) must be the same length

e.g.,

```
tuple[tuple[bool, bool], tuple[bool, bool], tuple[bool, bool]]
```

will result in `max_y=3, max_x=2`

while

```
tuple[tuple[bool, bool], tuple[bool], tuple[bool]]
```

will result in `RuntimeError`

Commands

Declaration of commands is the same as declaration of attributes with decorators:

```
@command
def set_and_check_voltage(self, voltage_to_set: float) -> float:
    device.set_voltage(voltage_to_set)
    return device.get_voltage()
```

Note: If you provide both type hints and dtype kwargs, the kwargs take priority:

e.g.,

```
@command(dtype_in=float, dtype_out=float)
def set_and_check_voltage(self, voltage_to_set: str) -> str:
```

(continues on next page)

(continued from previous page)

```
device.set_voltage(voltage_to_set)
return device.get_voltage()
```

will be a command that accepts float and returns float.

As in case of attributes, the SPECTRUM commands can be declared with *tuple[]* or *list[]* annotation:

```
@command
def set_and_check_voltages(self, voltages_set: tuple[float, float]) ->
tuple[float, float]:
    device.set_voltage(channel1, voltages_set[0])
    device.set_voltage(channel2, voltages_set[1])
    return device.get_voltage(channel=1), device.get_voltage(channel=2)
```

Note: Since commands do not have dimension parameters, length of tuple/list does not matter. If the type hints indicates 2 floats in the input, PyTango does not check that the input for each call received arrived with length 2.

6.17 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.

6.17.1 Basic logging

The most basic way to write a log message on your device is to use the *Device* logging related methods:

- `debug_stream()`
- `info_stream()`
- `warn_stream()`
- `error_stream()`
- `fatal_stream()`

Example:

```
def read_voltage(self):
    self.info_stream("read voltage attribute")
    # ...
    return voltage_value
```

This will print a message like:

```
1282206864 [-1215867200] INFO test/power_supply/1 read voltage attribute
```

every time a client asks to read the *voltage* attribute value.

The logging methods support argument list feature (since PyTango 8.1). Example:

```
def read_voltage(self):
    self.info_stream("read_voltage(%s, %d)", self.host, self.port)
    # ...
    return voltage_value
```

6.17.2 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 referred 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
```

6.17.3 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()
```

6.18 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`:

```
1 # PLC.py
2
3 from tango.server import Device
4
5 class PLC(Device):
6
7     # bla, bla my PLC code
8
9 if __name__ == "__main__":
10     PLC.run_server()
```

... and a `IRMirror` in a `IRMirror.py`:

```
1 # IRMirror.py
2
3 from tango.server import Device
4
5 class IRMirror(Device):
6
7     # bla, bla my IRMirror code
8
9 if __name__ == "__main__":
10     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     util = 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

6.19 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 (and will not call `initialize_dynamic_attributes()` either).

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: If the `fget` (`fread`), `fset` (`fwrite`) and `fisallowed` are given as `str(name)` they must be methods that exist on your Device class. If you want to use plain functions, or functions belonging to a different class, you should pass a callable.

Which arguments you have to provide depends on the type of the attribute. For example, a `WRITE` attribute does not need a read method.

Note: Starting from PyTango 9.4.0 the read methods for dynamic attributes can also be implemented with the high-level API. Prior to that, only the low-level API was available.

For the read function it is possible to use one of the following signatures:

```
def low_level_read(self, attr):
    attr.set_value(self.attr_value)

def high_level_read(self, attr):
    return self.attr_value
```

For the write function there is only one signature:

```
def low_level_write(self, attr):
    self.attr_value = attr.get_write_value()
```

Here is an example of a device which creates a dynamic attribute on startup:

```
1 from tango import AttrWriteType
2 from tango.server import Device, attribute
3
4 class MyDevice(Device):
5
6     def initialize_dynamic_attributes(self):
7         self._values = {"dyn_attr": 0}
8         attr = attribute(
9             name="dyn_attr",
10            dtype=int,
```

(continues on next page)

(continued from previous page)

```

11         access=AttrWriteType.READ_WRITE,
12         fget=self.generic_read,
13         fset=self.generic_write,
14         fisallowed=self.generic_is_allowed,
15     )
16     self.add_attribute(attr)
17
18     def generic_read(self, attr):
19         attr_name = attr.get_name()
20         value = self._values[attr_name]
21         return value
22
23     def generic_write(self, attr):
24         attr_name = attr.get_name()
25         value = attr.get_write_value()
26         self._values[attr_name] = value
27
28     def generic_is_allowed(self, request_type):
29         # note: we don't know which attribute is being read!
30         # request_type will be either AttReqType.READ_REQ or AttReqType.
31         ↪WRITE_REQ
32         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:

```

1  from tango import AttrWriteType
2  from tango.server import Device, attribute, command
3
4  class MyDevice(Device):
5
6      def init_device(self):
7          super(MyDevice, self).init_device()
8          self._values = {}
9
10     @command(dtype_in=str)
11     def CreateFloatAttribute(self, attr_name):
12         if attr_name not in self._values:
13             self._values[attr_name] = 0.0
14             attr = attribute(
15                 name=attr_name,
16                 dtype=float,
17                 access=AttrWriteType.READ_WRITE,
18                 fget=self.generic_read,
19                 fset=self.generic_write,
20             )
21             self.add_attribute(attr)
22             self.info_stream("Added dynamic attribute %r", attr_name)
23         else:
24             raise ValueError(f"Already have an attribute called {repr(attr_
25 ↪name) }")
26
27     def generic_read(self, attr):
28         attr_name = attr.get_name()

```

(continues on next page)

(continued from previous page)

```

28         self.info_stream("Reading attribute %s", attr_name)
29         value = self._values[attr.get_name()]
30         attr.set_value(value)
31
32     def generic_write(self, attr):
33         attr_name = attr.get_name()
34         value = attr.get_write_value()
35         self.info_stream("Writing attribute %s - value %s", attr_name,
→value)
36         self._values[attr.get_name()] = 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:

```

1  from tango import Attr, AttrWriteType
2  from tango.server import Device, command
3
4  class MyOldDevice(Device):
5
6      @command(dtype_in=str)
7      def CreateFloatAttribute(self, attr_name):
8          attr = Attr(attr_name, tango.DevDouble, AttrWriteType.READ_WRITE)
9          self.add_attribute(attr, self.read_General, self.write_General)
10
11     def read_General(self, attr):
12         self.info_stream("Reading attribute %s", attr.get_name())
13         attr.set_value(99.99)
14
15     def write_General(self, attr):
16         self.info_stream("Writing attribute %s - value %s", attr.get_
→name(), attr.get_write_value())

```

Dynamic attributes with type hint

Note: Starting from PyTango 9.5.0 dynamic attribute type can be defined by type hints in the read/write methods.

Usage of type hints is described in *Use Python type hints when declaring a device*. The only difference in case of dynamic attributes is, that there is no option to use type hint in attribute at assignment

e.g., the following code won't work:

```

def initialize_dynamic_attributes(self):
    voltage: float = attribute() # CANNOT BE AN OPTION FOR DYNAMIC_
→ATTRIBUTES!!!!!!!
    self.add_attribute(attr)

```

6.20 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()`.

6.20.1 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):

```

1 from tango import Util
2 from tango.server import Device, command
3
4 class MyDevice(Device):
5
6     @command(dtype_in=[str])
7     def CreateDevice(self, pars):
8         klass_name, dev_name = pars
9         util = Util.instance()
10        util.create_device(klass_name, dev_name, alias=None, cb=None)
11
12    @command(dtype_in=[str])
13    def DeleteDevice(self, pars):
14        klass_name, dev_name = pars
15        util = Util.instance()
16        util.delete_device(klass_name, dev_name)

```

An optional callback can be registered that will be executed after the device is registered 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.

6.20.2 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`:

```

1 class MyDevice(tango.Device):
2
3     def fill_new_device_properties(self, dev_name):
4         prop_names = db.get_device_property_list(self.get_name(), "*")
5         prop_values = db.get_device_property(self.get_name(), prop_names.
        ↪value_string)

```

(continues on next page)

(continued from previous page)

```

6         db.put_device_property(dev_name, prop_values)
7
8         # do the same for attributes...
9         ...
10
11     def Clone(self, dev_name):
12         klass = self.get_device_class()
13         klass.create_device(dev_name, alias=None, cb=self.fill_new_device_
→properties)
14
15     def DeleteSibling(self, dev_name):
16         klass = self.get_device_class()
17         klass.delete_device(dev_name)

```

Note that the `cb` parameter is optional. In the example it is given for demonstration purposes only.

6.21 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).

6.21.1 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

6.21.2 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,
6     ↪ "Array of string" ],
7                                       [ tango.ArgType.DevVarStringArray,
8     ↪ "This reversed array" ] ],
9     }
10
11     attr_list = { 'Long_attr' : [ [ tango.ArgType.DevLong ,
12     ↪ tango.AttrDataFormat.SCALAR ,
13                                     tango.AttrWriteType.READ],
14                                     { 'min alarm' : 1000, 'max alarm' : 1500,
15     ↪ } ],
16
17         'Short_attr_rw' : [ [ tango.ArgType.DevShort,
18                               tango.AttrDataFormat.SCALAR,
19                               tango.AttrWriteType.READ_WRITE ] ]
20     }
```

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 constructor is

```
def __init__(self, name):
    tango.DeviceClass.__init__(self, name)
    self.set_type("TestDevice")
```

The device type is set at line 3.

6.21.3 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 optionally 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

6.21.4 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 maximum 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

6.21.5 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:

```

1 class PyDsExp(tango.Device):
2
3     def __init__(self, cl, name):
4         tango.Device.__init__(self, cl, name)
5         self.info_stream('In PyDsExp.__init__')
6         PyDsExp.init_device(self)
7
8     def init_device(self):
9         self.info_stream('In Python init_device method')
10        self.set_state(tango.DevState.ON)
11        self.attr_short_rw = 66
12        self.attr_long = 1246
13
14        #-----
15
16    def delete_device(self):
17        self.info_stream('PyDsExp.delete_device')
18
19        #-----
20        # COMMANDS
21        #-----
22
23    def is_IOLong_allowed(self):
24        return self.get_state() == tango.DevState.ON
25
```

(continues on next page)

(continued from previous page)

```

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
<code>init_device</code>	None	None	Yes
<code>delete_device</code>	None	None	No
<code>always_executed_hook</code>	None	None	No
<code>signal_handler</code>	<code>int</code>	None	No
<code>read_attr_hardware</code>	sequence< <code>int</code> >	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
<code>is_<Cmd_name>_allowed</code>	None	Python boolean	No
<code>Cmd_name</code>	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`:

```
def is_IOLong_allowed(self):
    self.debug_stream("in is_IOLong_allowed")
    return self.get_state() == tango.DevState.ON

def IOLong(self, in_data):
    self.info_stream('IOLong', in_data)
    in_data = in_data * 2
    self.info_stream('IOLong returns', in_data)
    return in_data
```

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 *AttrReqtype*.

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*:

```
def read_Long_attr(self, the_attr):
    self.info_stream("read attribute name Long_attr")
    the_attr.set_value(self.attr_long)
```

Line 1

Method declaration with “the_attr” 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:

```
def write_Short_attr_rw(self, the_attr):
    self.info_stream("In write_Short_attr_rw for attribute ", the_attr.get_
    ↪name())
    self.attr_short_rw = the_attr.get_write_value(data)
```

Line 1

Method declaration with “the_attr” 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

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 (work in a virtualenv):

- `$ cd /path/to/pytango`
- `$ python -m pip install -r doc/requirements.txt`
- `$ 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 gevent numpy packaging pillow psutil 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 [pre-commit](#) which runs [ruff](#), a Python linter written in Rust. `pre-commit` is run as first job in every gitlab-ci pipeline and will fail if errors are detected.

It is recommended to install [pre-commit](#) locally to check code quality on every commit, before to push to GitLab. This is a one time operation:

- Install [pre-commit](#). `pipx` is a good way if you use it. Otherwise, see the [official documentation](#).
- Run `pre-commit install` at the root of your `pytango` repository.

That's it. `pre-commit` will now run automatically on every commit. If errors are reported, the commit will be aborted. You should fix them and try to commit again.

Note that you can also configure your editor to run `ruff`. See [ruff README](#).

7.4 Using Conda for development

For local development, it is recommended to work in a [Conda environment](#).

To run the tests locally (after activating your Conda environment):

- `$ pytest`

To run only some tests, use a filter argument, `-k`:

- `$ pytest -k test_ping`

7.5 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 development, Docker images with PyTango already installed are available from the [Square Kilometre Array Organisation's repository](#).

For example:

- `docker run --rm -ti artefact.skao.int/ska-tango-images-tango-pytango:9.4.3`

7.6 Releasing a new version

Starting from 9.4.2 `pytango` tries to follow `cpptango` releases with the delay up to ~1 month. The basic steps to make a new release are as follows:

Pick a version number

- A 3-part version numbering scheme is used: `<major>.<minor>.<patch>`
- Note that PyTango **does not** follow [Semantic Versioning](#). API changes can occur at minor releases (but avoid them if at all possible).
- The major and minor version fields (e.g., 9.4) 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:**

- 9.4.4.devN or 9.4.4rcN (current development branch)
- changes to 9.4.4 (the actual release)
- changes to 9.4.5.dev0 (bump the patch version at the end of the release process)

- **Minor release example:**

- 9.4.4.devN or 9.4.4rcN (current development branch)
- changes to 9.5.0 (the actual release)
- changes to 9.5.1.dev0 (bump the patch version at the end of the release process)

Check which versions of Python should this release support

- Follow the [version policy](#) and modify correspondingly *requires-python*, *classifiers*, and minimum runtime *dependencies* for NumPy in *pyproject.toml*.

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
- [] Release candidate testing and fixes complete
- [] 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
- [] Push stable to head of develop
- [] Make sure the documentation is updated for release (readthedocs)
- [] Check the new version was automatically uploaded 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-v9.4.4`.
- 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. A command like this could be used to see all the MR numbers, just change the initial version: `git log --ancestry-path v9.4.3..develop | grep "merge request" | sort`
- **Find the versions of the dependencies included in our binary PyPI packages, and update this in `docs/news.rst`.**
 - For Linux, see [PyTango CI wheel-linux config](#), and [pytango-builder tags](#),

- For Windows: See [cppTango CI config](#), [zmq-windows-ci CI config](#), and [PyTango CI wheel-win config](#).
- For macOS: see [PyTango CI output](#), and [cpptango conda-forge feedstock CI output](#) (for tango-idl).
- Bump the versions (`tango/release.py`, `pyproject.toml` and `CMakeLists.txt`). E.g. `version_info = (9, 4, 4)`, `version = "9.4.4"`, and `VERSION 9.4.4` for a final release. Or, for a release candidate: `version_info = (9, 4, 4, "rc", 1)`, `version = "9.4.4.rc1"`, and `VERSION 9.4.4`.
- Create a merge request to get these changes reviewed and merged before proceeding.

Make sure CI is OK on `develop` branch

- On Gitlab CI 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 `v9.4.4`)
- *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 9.4.4):**
 - `$ git checkout develop`
 - `$ git pull`
 - `$ git tag -a -m "tag v9.4.4" v9.4.4`
 - `$ git push -v origin refs/tags/v9.4.4`

Push `stable` to head of `develop`

- **Skip this step for release candidates!**
- Merge `stable` into the latest `develop`. It is recommended to do a fast-forward merge in order to avoid a confusing merge commit. This can be done by simply pushing `develop` to `stable` using this command:

```
$ git push origin develop:stable
```

This way the release tag corresponds to the actual release commit both on the `stable` and `develop` branches.
- In general, the `stable` branch should point to the latest release.

Upload the new version to PyPI

- The source tarball and binary wheels are automatically uploaded to PyPI by Gitlab CI on tag.

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, 4, 4)` to `(9, 4, 5, "dev", 0)`.
- In `pyproject.toml`, change `version`, e.g. from `"9.4.4"` to `"9.4.5.dev0"`.
- In `CMakeLists.txt`, change `VERSION`, e.g. from `9.4.4` to `9.4.5.0`.
- 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.

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 Databases 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 Databases 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.

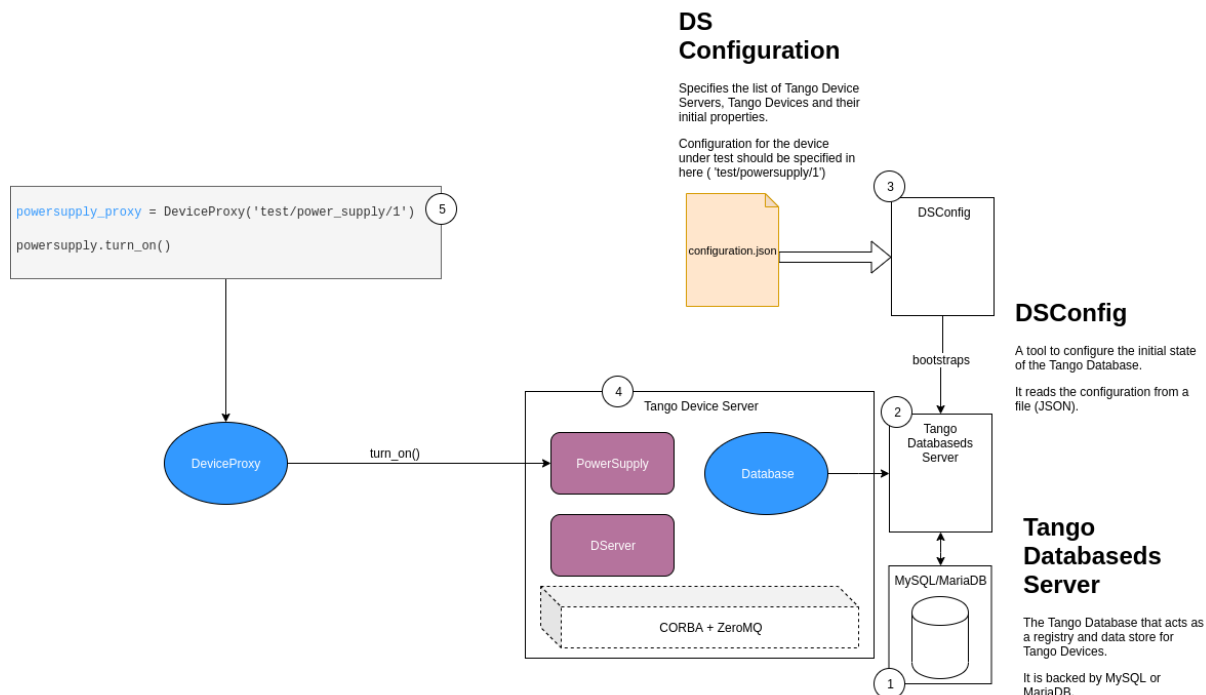


Figure 1. A schematic diagram showing the agents involved when testing a Tango device using the real Tango database and their interactions.

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.

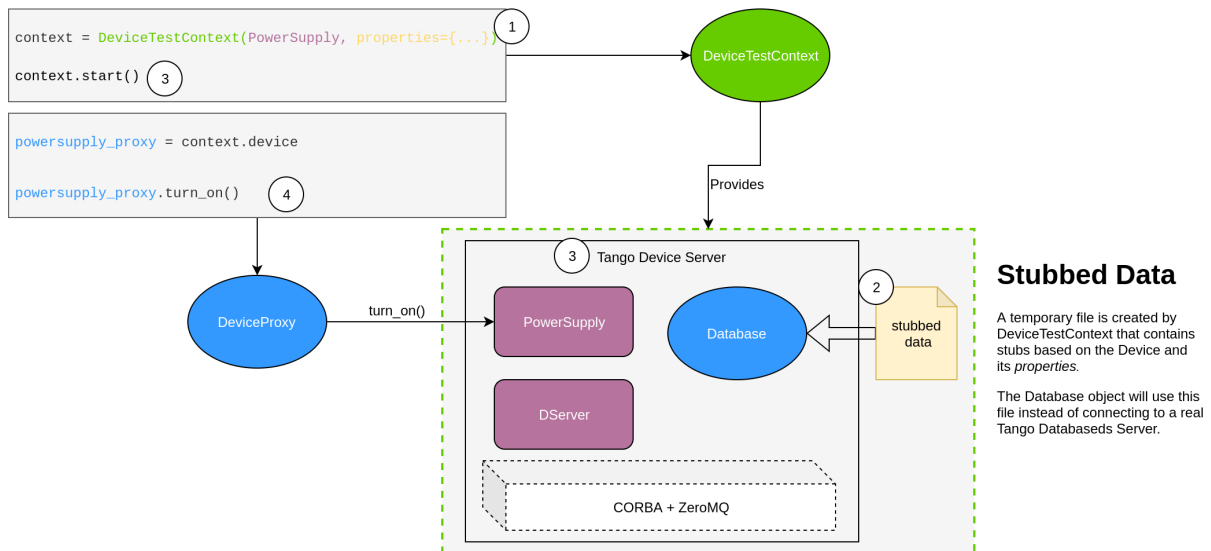


Figure 2. A schematic diagram showing the agents involved when testing a Tango device using the `DeviceTestContext` and their interactions.

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
<code>powersupply_proxy.turn_on()</code>	Tango Command	An action that the Tango Device performs.
<code>powersupply_proxy.voltage</code>	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`.

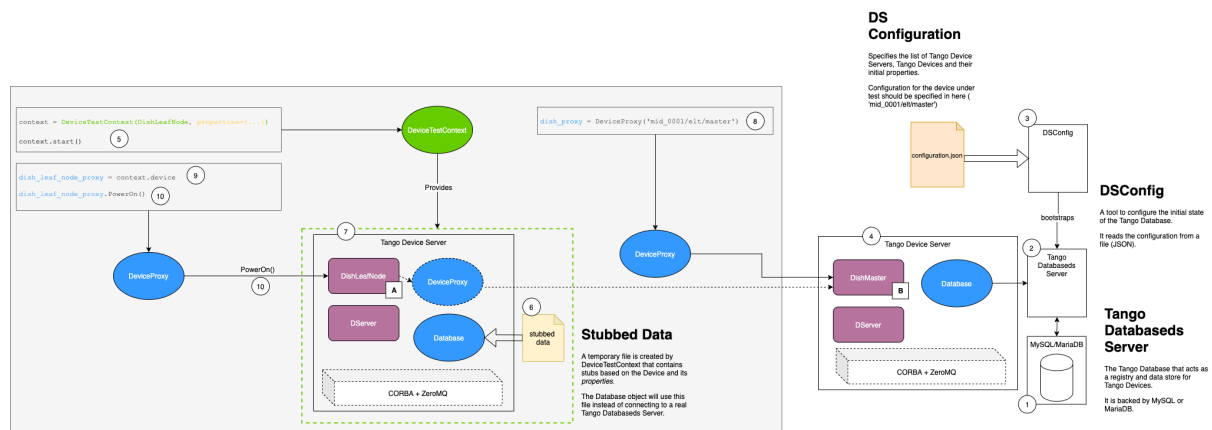


Figure 3. A schematic diagram showing the agents involved when testing multiple Tango devices using the `DeviceTestContext` together with the real Tango database and their interactions.

Examples:

- `DishLeafNode/conftest.py`

Testing with multiple DeviceTestContexts

Warning: 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 `DeviceTestContext`s 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.

From PyTango 9.5.0, this will require setting `process=True` on the nested `DeviceTestContext` instances.

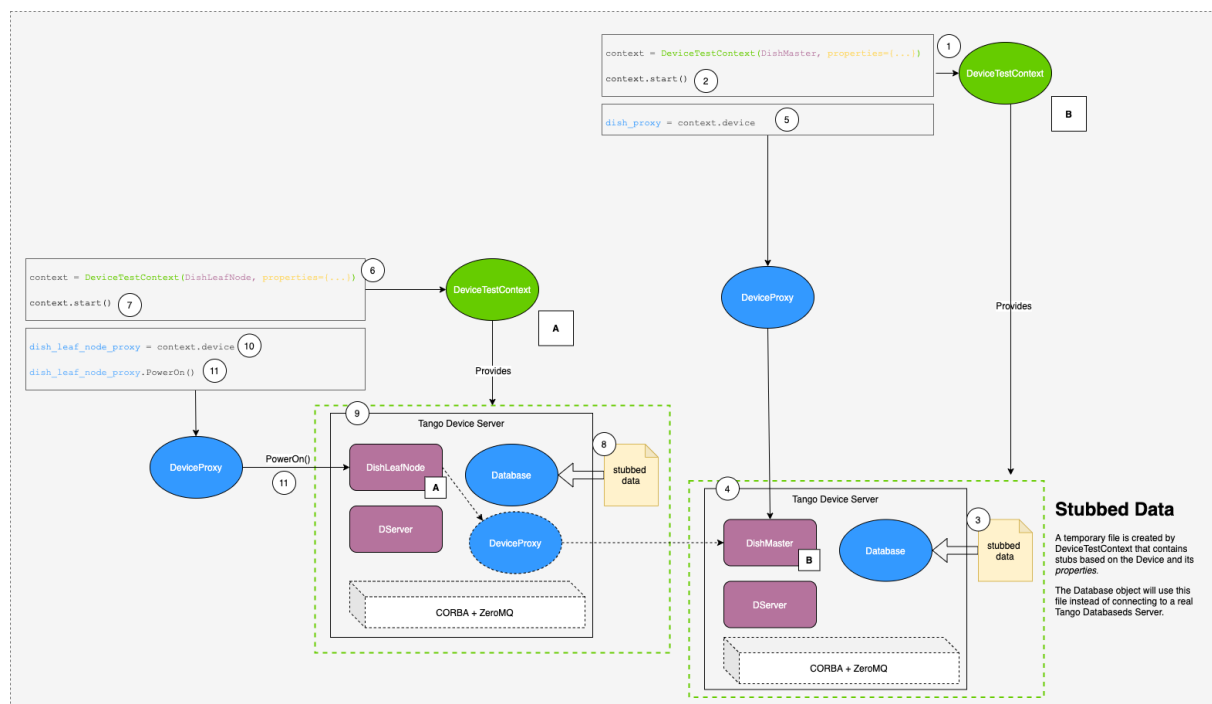


Figure 4. A schematic diagram showing the agents involved when testing multiple Tango devices using the *DeviceTestContext* and their interactions.

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 *MultiDeviceTestContext* 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.

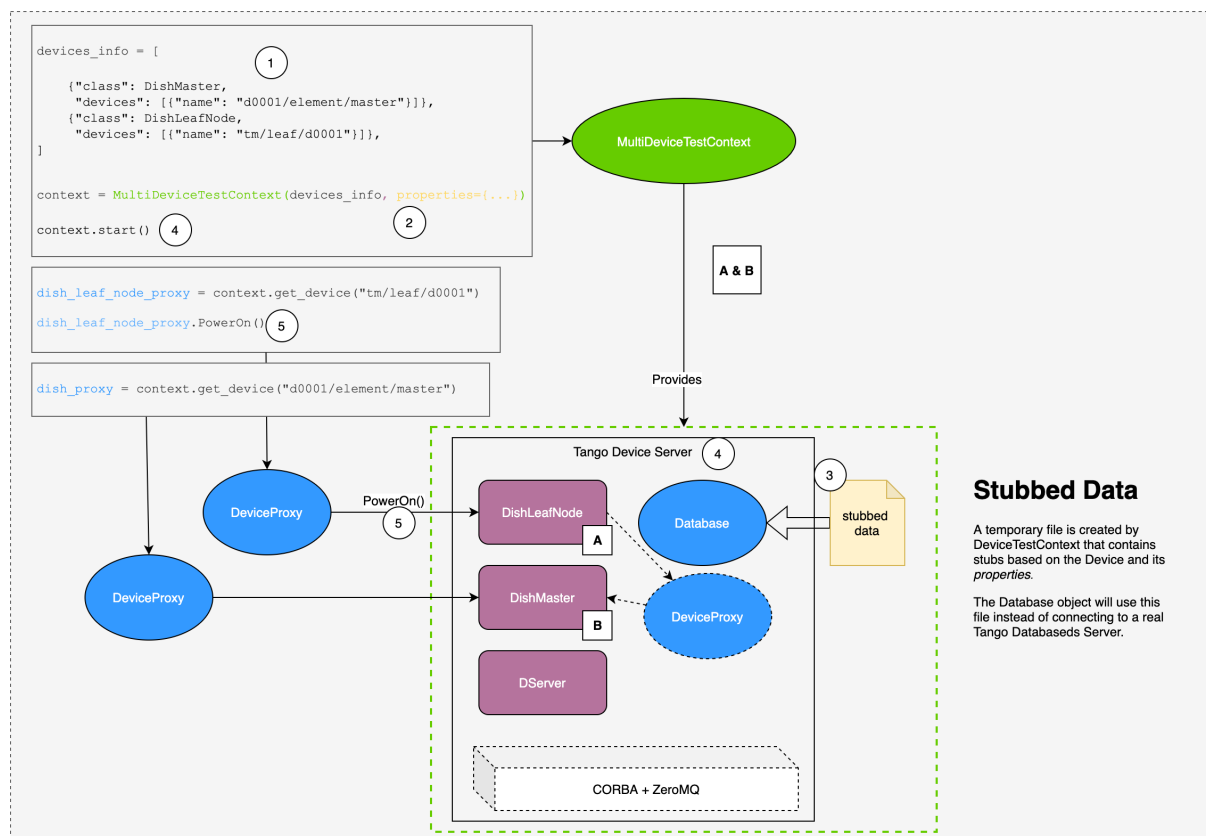


Figure 5. A schematic diagram showing the agents involved when testing multiple Tango devices using the *MultiDeviceTestContext* and their interactions.

Examples:

- *MultiDeviceTestContext* with fixtures

8.1.2 Issues

1. A single process that attempts to use a *DeviceTestContext* multiple times in threaded mode (so `kwargs.process=False`, or unspecified), will get a segmentation fault on the second usage. The segfault can be avoided using the `pytest-forked` plugin to run tests in separate processes (Linux and macOS, but not Windows). Either mark individual tests with the `@pytest.forked` decorator, or use the `pytest --forked` command line option to run every test in a new process. Running each test in a new process slows things down, so consider if all tests or just some tests need this.
2. Another way to avoid the segfault with multiple uses of *DeviceTestContext* is by setting the `kwargs.process=True`. In this case we don't need the forked execution, but consider the disadvantages in the previous section.
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.
5. 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.

Note: The same issues apply to *MultiDeviceTestContext*.

8.1.3 The `process` kwarg: thread vs. subprocess modes

When using *DeviceTestContext* or (*MultiDeviceTestContext*) with the `kwargs.process=False` (default), the Tango Device server runs in the same operating system process as the code that starts the *DeviceTestContext* (normally the test runner). With `process=True`, a new subprocess is created and the device server runs in that subprocess. In other words, a different operating system process to the test runner. In both cases, the test runner can communicate with the device via a client like *DeviceProxy* or *AttributeProxy*.

There is a subtle detail to note when using the `process=True` option with a nested device class. Consider the following example:

```
1 from tango import DevState
2 from tango.server import Device
3 from tango.test_context import DeviceTestContext
4
5
6 def test_empty_device_in_state_unknown():
7     class TestDevice(Device):
8         pass
9
10    with DeviceTestContext(TestDevice, process=False) as proxy:
11        assert proxy.state() == DevState.UNKNOWN
```

This will work fine, using thread-based mode. However, if we use the subprocess mode, i.e., `process=True`, we will get an error about serialisation of the class using pickle, like: `AttributeError: Can't pickle local object 'test_empty_device_in_state_unknown.<locals>.TestDevice'`. It fails when Python creates a subprocess using multiprocessing - the nested class definition cannot be passed to the subprocess.

One solution is to move the test class out of the function:

```

1 from tango import DevState
2 from tango.server import Device
3 from tango.test_context import DeviceTestContext
4
5
6 class TestEmptyDevice(Device):
7     pass
8
9
10 def test_empty_device_in_state_unknown():
11     with DeviceTestContext(TestEmptyDevice, process=True) as proxy:
12         assert proxy.state() == DevState.UNKNOWN

```

The next detail to consider is that the memory address space for two processes is independent. If we use `process=False` the device under test is running in the same process as the test runner, so we can access variables inside the class being tested (or even the device instance, if we keep a reference to it). With `process=True` the test runner cannot access the device server's memory.

Example of accessing class variables and device internals (only possible with `process=False`):

```

1 from weakref import WeakValueDictionary
2
3 from tango.server import Device
4 from tango.test_context import DeviceTestContext
5
6
7 class TestDeviceInternals(Device):
8     class_variable = 0
9     instances = WeakValueDictionary()
10
11     def init_device(self):
12         super().init_device()
13         TestDeviceInternals.class_variable = 123
14         TestDeviceInternals.instances[self.get_name()] = self
15         self._instance_variable = 456
16
17
18 def test_class_and_device_internals_accessible_with_process_false():
19     with DeviceTestContext(TestDeviceInternals, process=True) as proxy:
20         assert TestDeviceInternals.class_variable == 123
21
22         device_instance = TestDeviceInternals.instances[proxy.dev_name()]
23         assert device_instance._instance_variable == 456

```

The `weakref.WeakValueDictionary` isn't critical to this test (it could have been a standard dict), but it is shown as a way to avoid reference cycles in the `instances` dict. For example, if a device server were creating and deleting device instances at runtime. The reference cycles would prevent the cleanup of device instances by Python's garbage collector.

8.1.4 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, *root_atts*=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:

```
1 from time import sleep
2
3 from tango.server import Device, attribute, command
4 from tango.test_context import DeviceTestContext
5
6 class PowerSupply (Device) :
7
8     @attribute (dtype=float)
9     def voltage (self) :
10         return 1.23
11
12     @command
13     def calibrate (self) :
14         sleep (0.1)
15
16 def test_calibrate () :
17     '''Test device calibration and voltage reading.'''
18     with DeviceTestContext (PowerSupply, process=True) as proxy:
19         proxy.calibrate ()
20         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*.

New in version 9.2.1.

New in version 9.3.3: added *memorized* parameter.

New in version 9.3.6: added *green_mode* parameter.

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 them 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:

```

1 from tango import DeviceProxy
2 from tango.server import Device, attribute
3 from tango.test_context import MultiDeviceTestContext
4
5
6 class Device1(Device):
7     @attribute(dtype=int)
8     def attr1(self):
9         return 1
10
11
12 class Device2(Device):
13     @attribute(dtype=int)
14     def attr2(self):
15         dev1 = DeviceProxy("test/device/1")
16         return dev1.attr1 * 2
17
18
19 devices_info = (
20     {

```

(continues on next page)

(continued from previous page)

```

21     "class": Device1,
22     "devices": [
23         {"name": "test/device/1"},
24     ],
25 },
26 {
27     "class": Device2,
28     "devices": [
29         {
30             "name": "test/device/2",
31         },
32     ],
33 },
34 )
35
36
37 def test_devices():
38     with MultiDeviceTestContext(devices_info, process=True) as context:
39         proxy1 = context.get_device("test/device/1")
40         proxy2 = context.get_device("test/device/2")
41         assert proxy1.attr1 == 1
42         assert proxy2.attr2 == 2

```

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:
 - * a : 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 some such class
 - “devices” which value is a sequence of dicts with the following keys:
 - * “name” (str)
 - * “properties” (dict)
 - * “memorized” (dict)
 - * “root_atts” (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 the loopback IP address, 127.0.0.1.
- **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 multiple times, it may seg fault if the thread mode is chosen. See the [issues](#) and [process kwarg](#) discussion in the docs. 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.

New in version 9.3.2.

New in version 9.3.3: Added support for *memorized* key to “devices” field in *devices_info*. Added support for literal names for “class” field in *devices_info*.

New in version 9.3.3: added *green_mode* parameter.

Changed in version 9.5.0: By default, devices launched by a test context can be accessed using short names with [AttributeProxy](#), [DeviceProxy](#), and [Group](#). This can be disabled by setting the *enable_test_context_tango_host_override* class/instance attribute to *False* before starting the test context. Added support for *root_atts* key to “devices” field in *devices_info*.

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 them 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

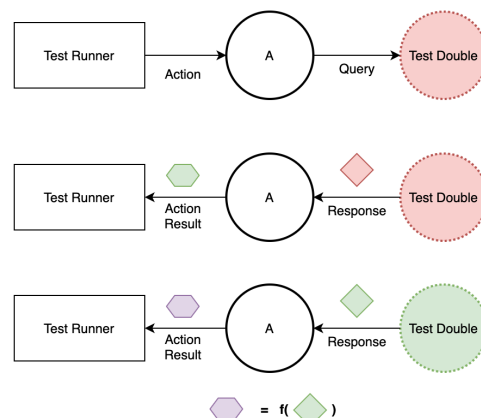
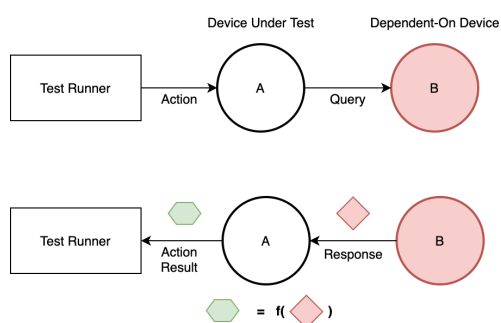
Suppose we want to test a Tango Device, **Device A**. In particular, we want to assert that when a certain *action* is invoked on **Device A**, it should produce an expected result. This will prove to us that **Device A** 's implementation sufficiently manifests the behaviour we would like it to have.

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.



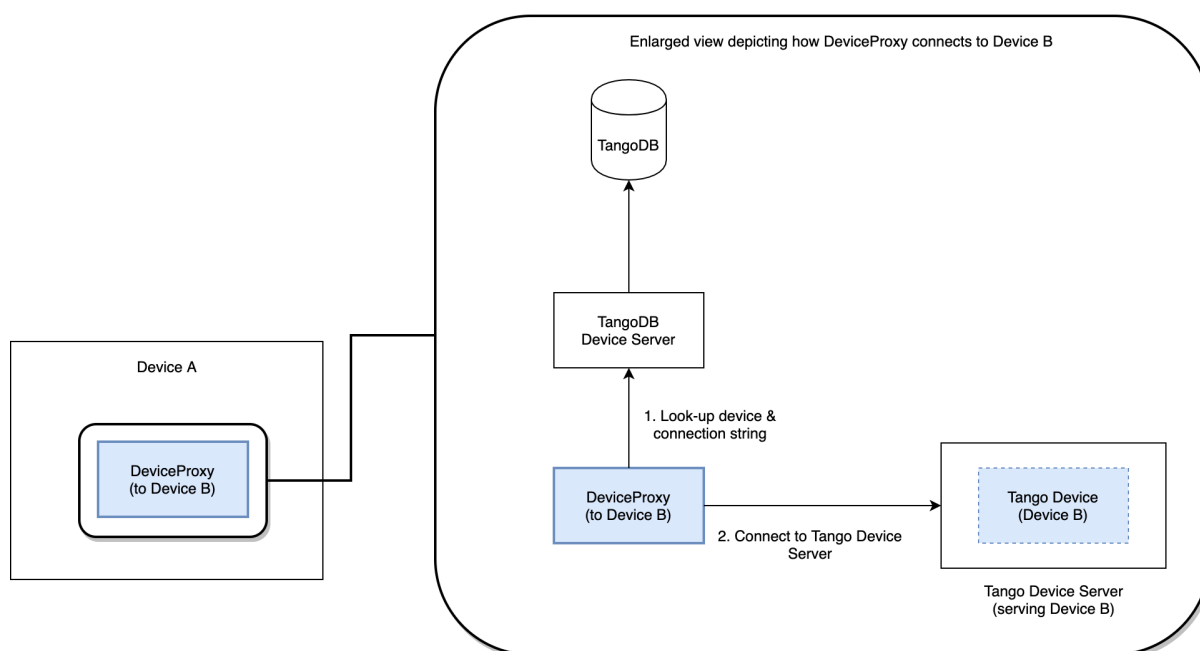
8.3.2 Tango's DeviceProxys

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.

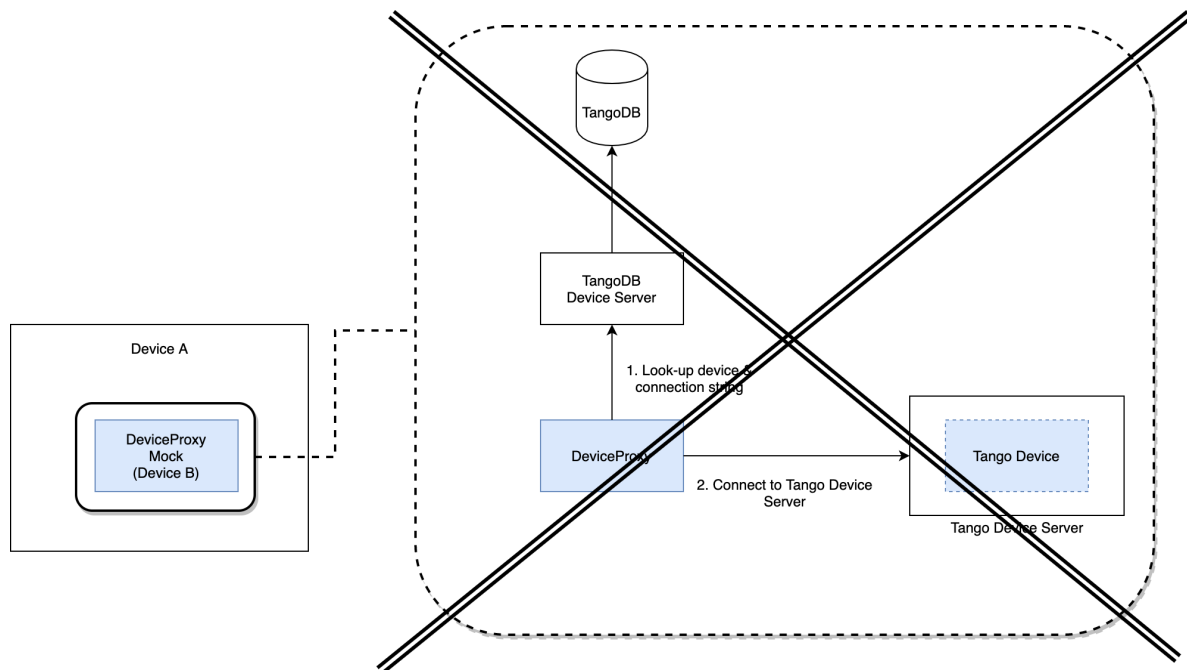
Thus, any Tango device (say, **Device A**) that depends on a secondary Tango device (**Device B**) will need to use a **DeviceProxy** to connect and communicate with the secondary device (**Device B**).

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:

```

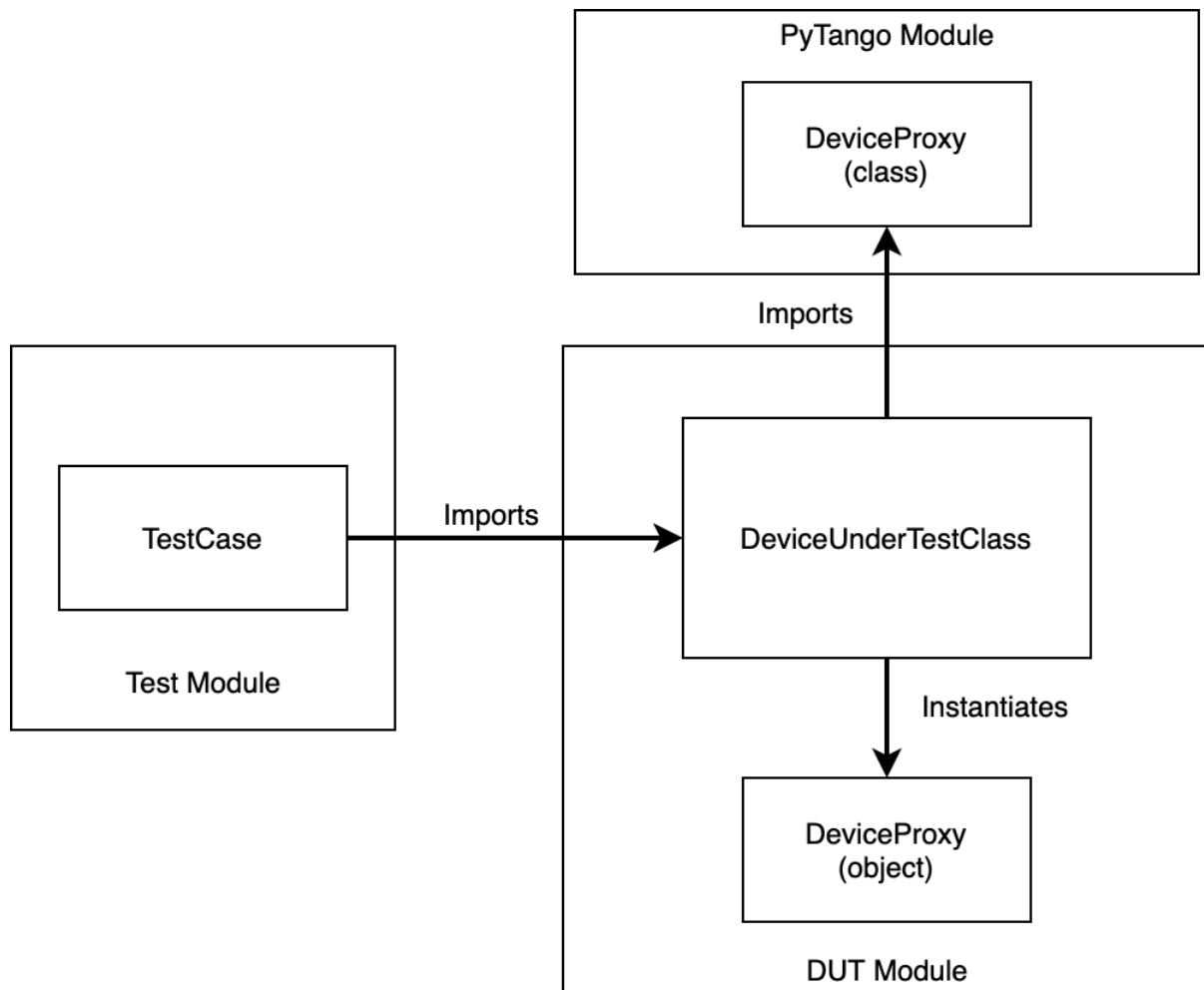
1 class CentralNode (SKABaseDevice) :
2     ...
3     def init_device(self) :
4         ...
5         self._leaf_device_proxy.append(DeviceProxy(self._dish_leaf_node_
6         ↪ devices[name]))
7         ...
8         self._csp_master_leaf_proxy = DeviceProxy(self.CspMasterLeafNodeFQDN)
9         ...
10        self._sdp_master_leaf_proxy = DeviceProxy(self.SdpMasterLeafNodeFQDN)
11        ...
12        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.

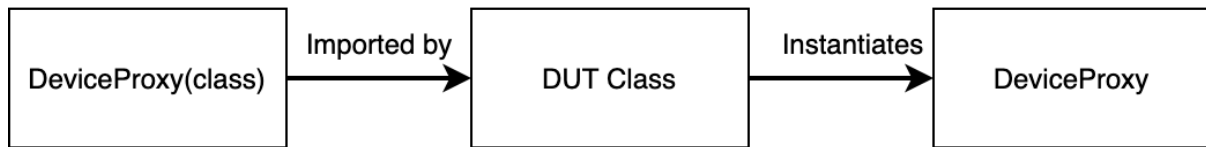
The diagram below illustrates the relationship between the **TestCase**, **DUT** and its transitive import of the **DeviceProxy** class from the **PyTango** module:



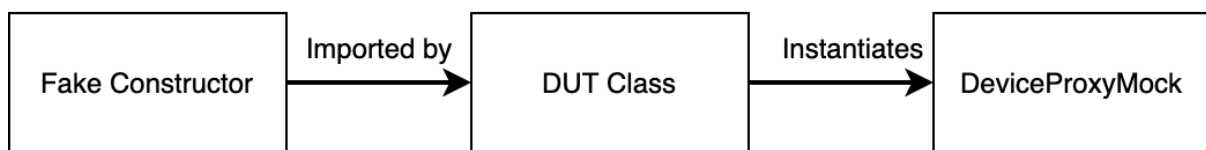
So, we want to replace the imported **DeviceProxy** class with our own Fake Constructor that will provide a Mocked Device Proxy for the **DUT** during tests.

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:

Original Implementation



Mocked Implementation



8.3.4 Solution

This can be achieved by using the `unittest.mock` library that comes with Python 3.

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:

```

1 with mock.patch(device_proxy_import_path) as patched_constructor:
2     patched_constructor.side_effect = lambda device_fqdn: proxies_to_mock.
   ↪ get(device_fqdn, Mock())
3     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 succinct 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](#).

8.4 Code coverage for Tango devices

8.4.1 Test coverage

A common tool for measuring code coverage is [Coverage.py](#). From their docs:

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

This is a very useful technique improve the quality of source code - both implementation and tests.

8.4.2 How to run Coverage.py for a PyTango high-level device

The recommended approach is to use `pytest`, with the `pytest-forked` and `pytest-cov` plugins. See the [issues](#) for notes on why the `pytest-forked` plugin, or subprocesses in general are necessary. The `pytest-cov` plugin specifically supports tests run in subprocesses.

For example:

```
pytest --forked --cov --cov-branch tests/my_tests.py
```

Warning: `coverage run -m pytest --forked tests/my_tests.py` will underestimate the code coverage due to the use of subprocesses.

Note: If checking coverage using the built-in feature of an IDE like PyCharm, note that it may start tests with `coverage` first, so the same problems with tests in a forked subprocess apply. Try disabling the forked plugin and running a single test at a time.

8.4.3 PyTango run-time patching to support Coverage.py

New in version 9.4.2.

Coverage.py works by using Python's `sys.settrace` function to record the execution of every line of code. If you are interested, you can read more about [how it works](#). Unfortunately, this mechanism doesn't automatically work for the callbacks from the cppTango layer. E.g., when a command is executed or an attribute is read, the Python method in your Tango device is generally not called in a thread that Python is aware of. If you were to call `threading.current_thread()` in these callbacks you would see `DummyThread` with a name like `Dummy-2`. The threads are created by cppTango (using omniORB), not with Python's `threading` module.

In order to get coverage to work, PyTango does the following:

1. Detect if Coverage.py is currently running (when importing `tango/server.py`).
2. If a Coverage.py session is active, and the feature isn't disabled (see environment variable below), then patch all the server methods that would get callbacks from the cppTango layer. This includes `init_device`, `always_executed_hook`, command methods, attribute read/write methods, is allowed methods, etc.
3. The patch calls `sys.settrace(threading._trace_hook)` to install the Coverage.py handler before calling your method. This allows these methods to be analysed for code coverage.

You can opt out of the patching, by setting the `PYTANGO_DISABLE_COVERAGE_TRACE_PATCHING=1` environment variable. The value it is set to doesn't matter. The presence of the variable disables the patching.

Note: This patching is only implemented for high-level API devices, in other words, those inheriting from `Device`. Low-level API devices, inheriting from `LatestDeviceImpl` (or earlier), do not benefit from this patching.

PYTHON AND NUMPY VERSION POLICY

Python and NumPy version policy

Supported versions are determined based on each PyTango release's anticipated release date, as follows:

1. All minor versions of Python released 42 months prior to that date, and at minimum the two latest minor versions.
2. All minor versions of NumPy released at that date that meet the requirements in [oldest-supported-numpy](#) for the corresponding Python version and platform.

As Python minor versions are released annually, this means that PyTango will drop support for the oldest minor Python version every year, and also gain support for a new minor version every year.

Note: NumPy's [NEP 29](#) policy requires that dependency versions are only changed on minor or major releases, however, as PyTango does not follow semantic versioning we allow changing the dependencies on any release, including a patch release. If PyTango ever changes to semantic versioning, then we can avoid such dependency updates on patch releases.

For example, a 9.4.2 PyTango release would support:

Python	Platform	NumPy
3.9	x86_64, win_amd64, win32, aarch64	>=1.19.3
3.9	arm64 (macOS)	>=1.21.0
3.10	x86_64, win_amd64, win32, aarch64, arm64	>=1.21.6
3.11	x86_64, win_amd64, win32, aarch64, arm64	>=1.23.2

A release after 5 April 2024 would require at least Python 3.10, and support Python 3.12.

The related discussion can be found <https://gitlab.com/tango-controls/pytango/-/issues/527>

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 libboost_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/python3.10/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](#).

PYTANGO ENHANCEMENT PROPOSALS

11.1 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

11.1.1 Abstract

This TEP aims to define a new high level API for writting device servers.

11.1.2 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,)
```

11.1.3 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 additionally `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()`

11.1.4 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 inherit 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)

```

11.1.5 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_executed_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()

```

11.1.6 References

tango.server

11.1.7 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

11.1.8 Copyright

This document has been placed in the public domain.

11.2 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

11.2.1 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).

11.2.2 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 writing the first version of this TEP I still didn't solve the problem! Shame on me!

Also at the same tango meeting 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 DataBases without the need for a mysql server.

11.2.3 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

11.2.4 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](#)):

mod- ule	Na- tive	Plat- forms	API	Database	Description
Native python 2.x					
<code>pickle</code>	Yes	all	dump/lc	file	python serialization/marshalling module
<code>shelve</code>	Yes	all	dict	file	high level persistent, dictionary-like object
<code>marshasha</code>	Yes	all	dump/lc	file	Internal Python object serialization
<code>anydbm</code>	Yes	all	dict	file	Generic access to DBM-style databases. Wrapper for <code>dbhash</code> , <code>gdbm</code> , <code>dbm</code> or <code>dumbdbm</code>
<code>dbm</code>	Yes	all	dict	file	Simple “database” interface
<code>gdbm</code>	Yes	unix	dict	file	GNU’s reinterpretation of <code>dbm</code>
<code>dbhash</code>	Yes	unix?	dict	file	DBM-style interface to the BSD database library (needs <code>bsddb</code>). Removed in python 3
<code>bsddb</code>	Yes	unix?	dict	file	Interface to Berkeley DB library. Removed in python 3
<code>dumbdb</code>	Yes	all	dict	file	Portable DBM implementation
<code>sqlite</code>	Yes	all	DBAPI2	file, mem- ory	DB-API 2.0 interface for SQLite databases
Native Python 3.x					
<code>pickle</code>	Yes	all	dump/lc	file	python serialization/marshalling module
<code>shelve</code>	Yes	all	dict	file	high level persistent, dictionary-like object
<code>marshasha</code>	Yes	all	dump/lc	file	Internal Python object serialization
<code>dbm</code>	Yes	all	dict	file	Interfaces to Unix “databases”. Wrapper for <code>dbm.gnu</code> , <code>dbm.ndbm</code> , <code>dbm.dumb</code>
<code>dbm.gnu</code>	Yes	unix	dict	file	GNU’s reinterpretation of <code>dbm</code>
<code>dbm.ndbm</code>	Yes	unix	dict	file	Interface based on <code>ndbm</code>
<code>dbm.dumb</code>	Yes	all	dict	file	Portable DBM implementation
<code>sqlite</code>	Yes	all	DBAPI2	file, mem- ory	DB-API 2.0 interface for SQLite databases

third-party DBAPI2

- `pyodbc`
- `mxCODBC`
- `kinterbasdb`
- `mxCODBC 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

11.2.5 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.

11.2.6 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).

11.2.7 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).

11.2.8 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.

11.2.9 Development

The development is being done in PyTango SVN trunk in the `tango.databases` module.

You can checkout with:

```
$ svn co https://tango-cs.svn.sourceforge.net/svnroot/tango-cs/bindings/  
→PyTango/trunk PyTango-trunk
```

11.2.10 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).

11.2.11 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>

WHAT'S NEW?

The sections below will give you the most relevant news from the PyTango releases. For help moving to a new release, or for the complete list of changes, see the following links:

12.1 Migration guide

This chapter describes how to migrate between the different PyTango API versions.

12.1.1 Moving to v9.4

This chapter describes how to migrate to PyTango versions 9.4.x from 9.3.x and earlier.

Dependencies and installation

Dependencies

PyTango v9.4.0 is the first release which **only** supports Python 3.6 or higher. If you haven't moved all your clients and devices to Python 3, now is the time!

PyTango v9.4.0 moved from [cppTango](#) 9.3.x to at least [cppTango](#) 9.4.1. It will not run with [cppTango](#) 9.4.0 or earlier.

In most cases, your existing PyTango devices and clients will continue to work as before, however there are important changes. In the other sections of the migration guide, you can find the incompatibilities and the necessary migration steps.

Installation

Environments created with Python 2 need to be ported to Python 3. You will need at least Python 3.6 and [cppTango](#) 9.4.1. Python dependencies will be installed automatically, including [numpy](#) - this is no longer optional, and doesn't have to be installed before installing PyTango.

The binary wheels on [PyPI](#) and [Conda-forge](#) makes installation very simple on many platforms. No need for compilation. See [Getting started](#).

Empty spectrum and image attributes

Warning: This is the most significant API change. It could cause errors in existing Tango clients and devices.

Both server-side writing, and client-side reading could be affected. There are differences depending on an attribute's data type and its access type (read/read-write). We go into detail below. The goal of the changes was to make usage more intuitive, and to provide a more consistent API.

Contents

- *Writing - server side*
- *Reading - client side*
 - *High-level API reads*
 - *Low-level API reads*
 - *Low-level API reads of set point (write value)*

Writing - server side

When an empty sequence is written to a spectrum or image attribute of type `DevString` the write function used to receive a `None` value, but now it will receive an empty `list`. For other types, the behaviour is unchanged - they were already receiving an empty `numpy.ndarray`.

Example device:

```
from tango import AttrWriteType
from tango.server import Device, attribute

class Test(Device):
    str1d = attribute(dtype=(str,), max_dim_x=4, access=AttrWriteType.
→WRITE)
    int1d = attribute(dtype=(int,), max_dim_x=4, access=AttrWriteType.
→WRITE)
    str2d = attribute(dtype=((str,)), max_dim_x=4, max_dim_y=4,
→access=AttrWriteType.WRITE)
    int2d = attribute(dtype=((int,)), max_dim_x=4, max_dim_y=4,
→access=AttrWriteType.WRITE)

    def write_str1d(self, values):
        print(f"Writing str1d: values={values!r}, type={type(values)}")

    def write_int1d(self, values):
        print(f"Writing int1d: values={values!r}, type={type(values)}")

    def write_str2d(self, values):
        print(f"Writing str2d: values={values!r}, type={type(values)}")

    def write_int2d(self, values):
        print(f"Writing int2d: values={values!r}, type={type(values)}")
```

If a client writes empty data to the device:

```
>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test#dbase=no
↳")
>>> dp.str1d = []
>>> dp.int1d = []
>>> dp.str2d = [[]]
>>> dp.int2d = [[]]
```

Old: The output from a v9.3.x PyTango device would be:

```
Writing str1d: values=None, type=<class 'NoneType'>
Writing int1d: values=array([], dtype=int64), type=<class 'numpy.ndarray'>
Writing str2d: values=None, type=<class 'NoneType'>
Writing int2d: values=array([], shape=(1, 0), dtype=int64), type=<class
↳ 'numpy.ndarray'>
```

New: The output from a v9.4.x PyTango device would be:

```
Writing str1d: values=[], type=<class 'list'>
Writing int1d: values=array([], dtype=int64), type=<class 'numpy.ndarray'>
Writing str2d: values=[], type=<class 'list'>
Writing int2d: values=array([], shape=(1, 0), dtype=int64), type=<class
↳ 'numpy.ndarray'>
```

Notice the change in values received for the `str1d` and `str2d` attributes. If your existing devices have special handling for `None`, then they may need to change.

Reading - client side

When clients read from spectrum and image attributes with empty values, clients will now receive an empty sequence instead of a `None` value. For `DevString` and `DevEnum` types, the *sequence* is a `tuple`, while other types get a `numpy.ndarray` by default.

There is a subtle inconsistency in PyTango 9.3.x - empty **read-only** spectrum and image attributes always returned `None` values, but **read-write** spectrum attributes *can* return empty sequences instead of `None` values. It depends on the set point (written value) stored for the attribute - if it is non-empty, then the client gets an empty sequence. This is shown in the examples below. From PyTango 9.4.x, the behaviour is more consistent.

Warning: Reading attributes of any type can still produce a `None` value if the quality is `ATTR_INVALID`. Client-side code should always be prepared for this. This behaviour is unchanged in PyTango 9.4.x (an exception being the fix for enumerated types using the high-level API, so that they also return `None`).

Note: It is possible to get the data returned in other container types using the `extract_as` argument with `tango.DeviceProxy.read_attribute()`.

This change affects values received via both the high-level API, and the low-level method it uses, `tango.DeviceProxy.read_attribute()`. It also affects all related read methods: `tango.DeviceProxy.read_attributes()`, `tango.DeviceProxy.read_attribute_async()`, `tango.DeviceProxy.read_attribute_reply()`, `tango.DeviceProxy.read_attributes_async()`, and `tango.DeviceProxy.read_attributes_reply()`.

The read attribute methods return data via `tango.DeviceAttribute` objects. These include a `value` field for the read value, and a `w_value` field for the last set point (i.e., last value written). Both of these

fields are affected by this change.

Example device:

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

class AnEnum(IntEnum):
    A = 0
    B = 1

class Test(Device):
    @attribute(dtype=(str,), max_dim_x=4)
    def str1d(self):
        return []

    @attribute(dtype=(AnEnum,), max_dim_x=4)
    def enum1d(self):
        return []

    @attribute(dtype=(int,), max_dim_x=4, access=AttrWriteType.READ)
    def int1d(self):
        return []

    @attribute(dtype=(int,), max_dim_x=4, access=AttrWriteType.READ_WRITE)
    def int1d_rw(self):
        return []

    @int1d_rw.write
    def int1d_rw(self, values):
        print(f"Writing int1d_rw: values={values!r}, type={type(values)!r}")

    @attribute(dtype=((str,),), max_dim_x=4, max_dim_y=4)
    def str2d(self):
        return [[]]

    @attribute(dtype=((int,),), max_dim_x=4, max_dim_y=4)
    def int2d(self):
        return [[]]
```

High-level API reads

Old: A PyTango 9.3.x client reads the empty data from the device using the high-level API:

```
>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test#dbase=no
↳")

>>> value = dp.str1d
>>> value, type(value)
(None, <class 'NoneType'>)

>>> value = dp.enum1d # broken in PyTango 9.3.x
Traceback: ... ValueError: None is not a valid enum1d

>>> value = dp.int1d # read-only attribute
>>> value, type(value)
```

(continues on next page)

(continued from previous page)

```
(None, <class 'NoneType'>)

>>> value = dp.int1d_rw # read-write attribute (default set point is [0])
>>> value, type(value)
(array([], dtype=int64), <class 'numpy.ndarray'>)
>>> dp.int1d_rw = [] # write empty value (set point changed to empty)
>>> value = dp.int1d_rw # read again, after set point changed
>>> value, type(value)
(None, <class 'NoneType'>)

>>> value = dp.str2d
>>> value, type(value)
(None, <class 'NoneType'>)

>>> value = dp.int2d
>>> value, type(value)
(None, <class 'NoneType'>)
```

In the above example, notice that high-level API reads of enumerated spectrum types fail under PyTango 9.3.x. Also see the difference in behaviour between read-only attributes like `int1d` and read-write attributes like `int1d_rw`. Read-write spectrum attributes behave inconsistently with empty data prior to PyTango 9.4.x.

New: A PyTango 9.4.x client reads the empty data from the device using the high-level API (device server has been restarted after previous example):

```
>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test#dbase=no
→")

>>> value = dp.str1d
>>> value, type(value)
((), <class 'tuple'>)

>>> value = dp.enum1d
>>> value, type(value)
((), <class 'tuple'>)

>>> value = dp.int1d # read-only attribute
>>> value, type(value)
(array([], dtype=int64), <class 'numpy.ndarray'>)

>>> value = dp.int1d_rw # read-write attribute (default set point is [0])
>>> value, type(value)
(array([], dtype=int64), <class 'numpy.ndarray'>)
>>> dp.int1d_rw = [] # write empty value (set point changed to empty)
>>> value = dp.int1d_rw # read again, after set point changed
>>> value, type(value)
(array([], dtype=int64), <class 'numpy.ndarray'>)

>>> value = dp.str2d
>>> value, type(value)
((), <class 'tuple'>)

>>> value = dp.int2d
>>> value, type(value)
(array([], shape=(1, 0), dtype=int64), <class 'numpy.ndarray'>)
```

Low-level API reads

In these examples, focus on the `value` field (the value read back), which changes as above, and the `type` field. Using PyTango 9.3.x, the `type` is number 100, which indicates an unknown type, while with PyTango 9.4.0, the type stays correct even when the value is empty. The change in `type` is something updated in [cppTango 9.4.1](#).

Old: A PyTango 9.3.x client reads the empty data from the device using the low-level API:

```
>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test
↳#dbase=no")

>>> print(dp.read_attribute("str1d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.SPECTRUM
    dim_x = 0
    dim_y = 0
has_failed = False
is_empty = True
    name = 'str1d'
    nb_read = 0
    nb_written = 0
    quality = tango._tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    time = TimeVal(tv_nsec = 0, tv_sec = 1676068470, tv_usec =
↳650091)
    type = tango._tango.CmdArgType(100)
    value = None
    w_dim_x = 0
    w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    w_value = None]

>>> print(dp.read_attribute("int1d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.SPECTRUM
    dim_x = 0
    dim_y = 0
has_failed = False
is_empty = False
    name = 'int1d'
    nb_read = 0
    nb_written = 1
    quality = tango._tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    time = TimeVal(tv_nsec = 0, tv_sec = 1676068478, tv_usec =
↳597279)
    type = tango._tango.CmdArgType.DevLong64
    value = array([], dtype=int64)
    w_dim_x = 1
    w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 1, dim_y = 0)
    w_value = array([0])

>>> print(dp.read_attribute("str2d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.IMAGE
    dim_x = 0
```

(continues on next page)

(continued from previous page)

```

        dim_y = 1
        has_failed = False
        is_empty = True
        name = 'str2d'
        nb_read = 0
        nb_written = 0
        quality = tango._tango.AttrQuality.ATTR_VALID
        r_dimension = AttributeDimension(dim_x = 0, dim_y = 1)
        time = TimeVal(tv_nsec = 0, tv_sec = 1676068484, tv_usec = ↵
↵896408)
        type = tango._tango.CmdArgType(100)
        value = None
        w_dim_x = 0
        w_dim_y = 0
        w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
        w_value = None]

>>> print(dp.read_attribute("int2d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.IMAGE
    dim_x = 0
    dim_y = 1
    has_failed = False
    is_empty = True
    name = 'int2d'
    nb_read = 0
    nb_written = 0
    quality = tango._tango.AttrQuality.ATTR_VALID
    r_dimension = AttributeDimension(dim_x = 0, dim_y = 1)
    time = TimeVal(tv_nsec = 0, tv_sec = 1676068489, tv_usec = ↵
↵330193)
    type = tango._tango.CmdArgType(100)
    value = None
    w_dim_x = 0
    w_dim_y = 0
    w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    w_value = None]

```

New: A PyTango 9.4.x client reads the empty data from the device using the low-level API:

```

>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test#dbase=no
↵")

>>> print(dp.read_attribute("str1d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.SPECTRUM
    dim_x = 0
    dim_y = 0
    has_failed = False
    is_empty = True
    name = 'str1d'
    nb_read = 0
    nb_written = 0
    quality = tango._tango.AttrQuality.ATTR_VALID
    r_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    time = TimeVal(tv_nsec = 0, tv_sec = 1676068550, tv_usec = 333749)

```

(continues on next page)

(continued from previous page)

```

        type = tango._tango.CmdArgType.DevString
        value = ()
        w_dim_x = 0
        w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
        w_value = ()]

>>> print(dp.read_attribute("int1d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.SPECTRUM
        dim_x = 0
        dim_y = 0
has_failed = False
is_empty = False
        name = 'int1d'
        nb_read = 0
nb_written = 1
        quality = tango._tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
        time = TimeVal(tv_nsec = 0, tv_sec = 1676068554, tv_usec = 243413)
        type = tango._tango.CmdArgType.DevLong64
        value = array([], dtype=int64)
        w_dim_x = 1
        w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 1, dim_y = 0)
        w_value = array([0])]

>>> print(dp.read_attribute("str2d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.IMAGE
        dim_x = 0
        dim_y = 1
has_failed = False
is_empty = True
        name = 'str2d'
        nb_read = 0
nb_written = 0
        quality = tango._tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 0, dim_y = 1)
        time = TimeVal(tv_nsec = 0, tv_sec = 1676068558, tv_usec = 191433)
        type = tango._tango.CmdArgType.DevString
        value = ()
        w_dim_x = 0
        w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
        w_value = ()]

>>> print(dp.read_attribute("int2d"))
DeviceAttribute[
data_format = tango._tango.AttrDataFormat.IMAGE
        dim_x = 0
        dim_y = 1
has_failed = False
is_empty = True
        name = 'int2d'
        nb_read = 0

```

(continues on next page)

(continued from previous page)

```

nb_written = 0
quality = tango._tango.AttrQuality.ATTR_VALID
r_dimension = AttributeDimension(dim_x = 0, dim_y = 1)
    time = TimeVal(tv_nsec = 0, tv_sec = 1676068562, tv_usec = 50107)
    type = tango._tango.CmdArgType.DevLong64
    value = array([], shape=(1, 0), dtype=int64)
w_dim_x = 0
w_dim_y = 0
w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
w_value = array([], shape=(0, 0), dtype=int64)]

```

Low-level API reads of set point (write value)

In these examples, focus on the `w_value` field which is the set point, or last written value.

Old: A PyTango 9.3.x client changes the set point and reads using the low-level API:

```

>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test
↳#dbase=no")

>>> dp.int1d_rw = [1, 2]
>>> print(dp.read_attribute("int1d_rw"))
DeviceAttribute[
    ...
    type = tango._tango.CmdArgType.DevLong64
    w_dim_x = 2
    w_dim_y = 0
    w_dimension = AttributeDimension(dim_x = 2, dim_y = 0)
    w_value = array([1, 2])]

>>> dp.int1d_rw = []
>>> print(dp.read_attribute("int1d_rw"))
DeviceAttribute[
    ...
    type = tango._tango.CmdArgType(100)
    w_dim_x = 0
    w_dim_y = 0
    w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    w_value = None]

```

New: A PyTango 9.4.x client changes the set point and reads using the low-level API:

```

>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/test
↳#dbase=no")

>>> dp.int1d_rw = [1, 2]
>>> print(dp.read_attribute("int1d_rw"))
DeviceAttribute[
    ...
    type = tango._tango.CmdArgType.DevLong64
    w_dim_x = 2
    w_dim_y = 0
    w_dimension = AttributeDimension(dim_x = 2, dim_y = 0)
    w_value = array([1, 2])]

```

(continues on next page)

(continued from previous page)

```
>>> dp.int1d_rw = []
>>> print(dp.read_attribute("int1d_rw"))
DeviceAttribute[
    ...
    type = tango._tango.CmdArgType.DevLong64
    w_dim_x = 0
    w_dim_y = 0
    w_dimension = AttributeDimension(dim_x = 0, dim_y = 0)
    w_value = array([], dtype=int64)]
```

Changes to `extract_as` when reading spectrum and image attributes

When a client reads a spectrum or image attribute, the default behaviour is to return the data in a `tuple` for strings and enumerated types, or a `numpy.ndarray` for other types. This can be changed using the `extract_as` argument passed to `tango.DeviceProxy.read_attribute()` (and similar methods).

Changes in 9.4.x affect the results when using `tango.ExtractAs.Bytes`, `tango.ExtractAs.ByteArray`, and `tango.ExtractAs.String`. The methods are a little unusual as they try to provide the raw data values as bytes or strings, rather than the original data type.

Keep value and `w_value` separate

Prior to 9.4.x, the data in the `tango.DeviceAttribute` value and `w_value` fields would be concatenated (respectively) and returned in the `value` field. For a read-only attributes this was reasonable, but not for read-write attributes.

In 9.4.x, the data in the two fields remains independent.

Disabled types

Extracting data to bytes and strings has been disabled for attributes of type `DevString` because concatenating the data from all the strings is not clearly defined. E.g., should null termination characters be included for each item? This may be re-enabled in future, once the solution becomes clear.

Problems with UTF-8 encoding

For arbitrary binary data, it is not always possible to convert it into a Python `str`. The conversion from bytes will attempt to decode the data assuming UTF-8 encoding, so many byte sequences are invalid. This is not a new problem in version 9.4.x, but will be noticeable if moving from Python 2 to Python 3. For arbitrary data, including numeric types, rather extract to `bytes` or `bytearray`.

Non-bound user functions for read/write/isallowed and commands

When providing the user functions that are executed on attribute and command access, the general requirement was that they had to be methods on the `tango.server.Device` class. This has led to some confusion when developers try a plain function instead. There were some ways to work around this, e.g., by patching the Python device instance `__dict__`. From Pytango 9.4.x this is no longer necessary. User functions can now be defined outside of the device class, if desired.

This feature applies to both static and dynamic attributes/commands:

- attribute read method (`fget/fread` kwarg)

- attribute write method (fset/fwrite kwarg)
- attribute is allowed method (fisallowed kwarg)
- command is allowed method (fisallowed kwarg)

The first argument to these functions will be a reference to the device instance.

For example, using static attributes and commands:

```
from tango import AttrWriteType, AttReqType
from tango.server import Device, command, attribute

global_data = {"example_attr1": 100}

def read_example_attr1(device):
    print(f"read from device {device.get_name()}")
    return global_data["example_attr1"]

def write_example_attr1(device, value):
    print(f"write to device {device.get_name()}")
    global_data["example_attr1"] = value

def is_example_attr1_allowed(device, req_type):
    print(f"is_allowed attr for device {device.get_name()}")
    assert req_type in (AttReqType.READ_REQ, AttReqType.WRITE_REQ)
    return True

def is_cmd1_allowed(device):
    print(f"is_allowed cmd for device {device.get_name()}")
    return True

class Test(Device):

    example_attr1 = attribute(
        fget=read_example_attr1,
        fset=write_example_attr1,
        fisallowed=is_example_attr1_allowed,
        dtype=int,
        access=AttrWriteType.READ_WRITE
    )

    @command(dtype_in=int, dtype_out=int, fisallowed=is_cmd1_allowed)
    def identity1(self, value):
        return value
```

Another example, using dynamic attributes and commands:

```
from tango import AttrWriteType, AttReqType
from tango.server import Device, command, attribute

global_data = {"example_attr2": 200}

def read_example_attr2(device, attr):
```

(continues on next page)

(continued from previous page)

```
print(f"read from device {device.get_name()} attr {attr.get_name()}")
return global_data["example_attr2"]

def write_example_attr2(device, attr):
    print(f"write to device {device.get_name()} attr {attr.get_name()}")
    value = attr.get_write_value()
    global_data["example_attr2"] = value

def is_example_attr2_allowed(device, req_type):
    print(f"is_allowed attr for device {device.get_name()}")
    assert req_type in (AttReqType.READ_REQ, AttReqType.WRITE_REQ)
    return True

def is_cmd2_allowed(device):
    print(f"is_allowed cmd for device {device.get_name()}")
    return True

class Test(Device):
    def initialize_dynamic_attributes(self):
        attr = attribute(
            name="example_attr2",
            dtype=int,
            access=AttrWriteType.READ_WRITE,
            fget=read_example_attr2,
            fset=write_example_attr2,
            fisallowed=is_example_attr2_allowed,
        )
        self.add_attribute(attr)
        cmd = command(
            f=self.identity2,
            dtype_in=int,
            dtype_out=int,
            fisallowed=is_cmd2_allowed,
        )
        self.add_command(cmd)

    def identity2(self, value):
        return value
```

High-level API for dynamic attributes

The read methods for dynamic attributes can now be coded in a more Pythonic way, similar to the approach used with static attributes. This is a new feature so existing code does not have to be modified.

Prior to 9.4.x, the methods had to look something like:

```
def low_level_read(self, attr):
    value = self._values[attr.get_name()]
    attr.set_value(value)
```

From 9.4.x, the read method can simply return the result:

```
def high_level_read(self, attr):
    value = self._values[attr.get_name()]
    return value
```

Further details are discussed in *Create attributes dynamically*.

High-level API support for DevEnum spectrum and image attributes

Prior to 9.4.x there were problems with both the client-side and server-side implementation of spectrum and image attribute of type DevEnum.

On the client side, the high-level API would raise an exception when such attributes were read. The low-level API worked correctly.

On the server side, a Python enumerated type couldn't be used to defined spectrum or image attributes in a PyTango device.

Both of these issues have been fixed.

Example of server:

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

class DisplayType(IntEnum):
    ANALOG = 0
    DIGITAL = 1

class Clock(Device):

    _display_types = [DisplayType(0)]

    displays = attribute(dtype=(DisplayType,), max_dim_x=4,
        ↳access=AttrWriteType.READ_WRITE)

    def read_displays(self):
        return self._display_types

    def write_displays(self, values):
        display_types = [DisplayType(value) for value in values]  #
        ↳optional conversion from int values
        self._display_types = display_types
```

Example of client-side usage:

```
>>> dp = tango.DeviceProxy("tango://127.0.0.1:8888/test/nodb/clock#dbase=no
↳")

>>> dp.displays
(<displays.ANALOG: 0>,)

>>> display = dp.displays[0]  # useful as client-side copy of the IntEnum
↳class
>>> display.__class__.__members__
mappingproxy({'ANALOG': <displays.ANALOG: 0>, 'DIGITAL': <displays.
↳DIGITAL: 1>})
```

(continues on next page)

(continued from previous page)

```

>>> dp.displays = ["ANALOG", "DIGITAL", "ANALOG"] # string assignment
>>> dp.displays
(<displays.ANALOG: 0>, <displays.DIGITAL: 1>, <displays.ANALOG: 0>)

>>> dp.displays = [display.ANALOG, "DIGITAL", display.ANALOG, 1] # mixed_
↳assignment
>>> dp.displays
(<displays.ANALOG: 0>, <displays.DIGITAL: 1>, <displays.ANALOG: 0>,
↳<displays.DIGITAL: 1>)

```

Optionally add Python attributes to DeviceProxy instances

Prior to PyTango 9.3.4, developers could add arbitrary Python attributes to a *DeviceProxy* instance. From version 9.3.4 PyTango raises an exception if this is attempted. To aid backwards compatibility, where the old use case was beneficial, some new methods have been added to the *DeviceProxy*. We use the term *dynamic interface* to refer this. When the *dynamic interface* is frozen (the default) it cannot be changed, and you get an exception if you try. Unfreeze it by calling *unfreeze_dynamic_interface()* if you want to make these kinds of changes. Here is an example:

```

>>> import tango
>>> dp = tango.DeviceProxy("sys/tg_test/1")
>>> dp.is_dynamic_interface_frozen()
True
>>> dp.non_tango_attr = 123
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/lib/python3.11/site-packages/tango/device_proxy.py", line 484, in_
↳__DeviceProxy__setattr
    raise e from cause
  File "/lib/python3.11/site-packages/tango/device_proxy.py", line 478, in_
↳__DeviceProxy__setattr
    raise AttributeError(
AttributeError: Tried to set non-existent attr 'non_tango_attr' to 123.
The DeviceProxy object interface is frozen and cannot be modified - see_
↳tango.DeviceProxy.freeze_dynamic_interface for details.
>>> dp.unfreeze_dynamic_interface()
/lib/python3.11/site-packages/tango/device_proxy.py:302: UserWarning:_
↳Dynamic interface unfrozen on DeviceProxy instance TangoTest(sys/tg_test/
↳1) id=0x102a4ea20 - arbitrary Python attributes can be set without_
↳raising an exception.
    warnings.warn(
>>> dp.non_tango_attr = 123
>>> dp.non_tango_attr
123
>>> dp.freeze_dynamic_interface()
>>> dp.is_dynamic_interface_frozen()
True

```


New attribute decorators

In order to improve with readability and better match Python’s `property` syntax, the `tango.server.attribute` decorator has a few new options. No changes are required for existing code. These are the new options:

- `@<attribute>.getter`
- `@<attribute>.read`
- `@<attribute>.is_allowed`

The `getter` and `read` methods are aliases for each other and are applicable to reading of attributes. They provide symmetry with the decorators that handle writing to attributes: `setter` and `write`. The new `is_allowed` decorator provides a consistent way to define the “is allowed” method for an attribute.

Unlike Python properties, the names of the decorated methods do not have to match.

A simple example:

```
from tango import AttReqType, DevState
from tango.server import Device, attribute

class Test(Device):
    _simulated_voltage = 0.0

    voltage = attribute(dtype=float)

    @voltage.getter
    def voltage(self):
        return self._simulated_voltage

    @voltage.setter
    def voltage(self, value):
        self._simulated_voltage = value

    @voltage.is_allowed
    def voltage_can_be_changed(self, req_type):
        if req_type == AttReqType.WRITE_REQ:
            return self.get_state() == DevState.ON
        else:
            return True
```

There are many variations possible when using these. See more in [Write a server](#).

Broken logging with “%”

Device log streams calls that included literal % symbols but no args used to raise an exception. For example, `self.debug_stream("I want to log a %s symbol")`, or when when logging an exception traceback that happened to include a format method. This is fixed in PyTango 9.4.x, so if you had code that was doing additional escaping for the % characters before logging, this can be removed.

If you do pass additional args to a logging function after the message string, and the number of args doesn’t match the number of % string formatting characters an exception will still be raised.

The updated methods are:

- `tango.server.Device.debug_stream()`
- `tango.server.Device.info_stream()`
- `tango.server.Device.warn_stream()`

- `tango.server.Device.error_stream()`
- `tango.server.Device.fatal_stream()`
- `tango.server.Device.fatal_stream()`
- `tango.Logger.debug()`
- `tango.Logger.info()`
- `tango.Logger.warn()`
- `tango.Logger.error()`
- `tango.Logger.fatal()`
- `tango.Logger.log()`
- `tango.Logger.log_unconditionally()`

Where `tango.Logger` is the type returned by `tango.server.Device.get_logger()`.

12.1.2 Moving to v9.5

This chapter describes how to migrate to PyTango versions 9.5.x from 9.4.x and earlier.

Dependencies and installation

In most cases, your existing PyTango devices and clients will continue to work as before, however there are important changes. In the other sections of the migration guide, you can find the incompatibilities and the necessary migration steps.

Dependencies

PyTango v9.5.0 requires Python 3.9 or higher.

PyTango v9.5.0 moved from [cppTango](#) 9.4.x to at least 9.5.0. It will not run with earlier versions. `cppTango`'s dependencies have also changed, most notably, `omniORB` 4.3.x is required, instead of 4.2.x.

OmniORB 4.3 troubleshooting

If you get an error like:

```
omniORB: (0) 2023-10-23 13:18:11.872312: ORB_init failed: Bad parameter_
↳ (2097152 # 2 MBytes.)
for ORB configuration option giopMaxMsgSize, reason: Invalid value, expect_
↳ n >= 8192 or n == 0
```

This is probably because your system has the string `2097152 # 2 MBytes.` in a configuration file, and this is incorrectly parsed by `omniORB` 4.3. For example, if the Linux system package `libomniORB4-2` is installed then configuration file `/etc/omniORB.cfg` has this problem.

You can edit the file and remove the comment from that line, changing it to `2097152`. Alternatively, uninstall the `libomniORB4-2` package if it isn't required on your system. PyTango's binary Python wheels include their own copy of `omniORB`, so they don't require the system package. Similarly, the conda-forge packages come with their own copy of `omniORB`.

Installation

Similar to the 9.4.x series, the binary wheels on [PyPI](#) and [Conda-forge](#) make installation very simple on many platforms. No need for compilation. See [Getting started](#).

If you are compiling from source, you may notice that the build system has changed completely. We now use [scikit-build-core](#), and use [CMake](#) for the compilation on all platforms. See [building from source](#).

Removal of DevInt

In PyTango v9.5.0 the DevInt data type was removed from PyTango because the corresponding DEV_INT type was [removed from cppTango](#). The DevInt didn't function correctly, but the nearest replacement is DevLong.

Short-name access to TestContext devices

Description

When a device server is launched via the [DeviceTestContext](#) or [MultiDeviceTestContext](#), it is run without a Tango database. This means that access to any of its devices must be done via a fully-qualified [Tango Resource Locator](#) (FQTRL). I.e., we need something like `tango://172.17.0.3:48493/test/device/1#dbase=no`, instead of just the short name: `test/device/1`. Requiring the FQTRL is inconvenient.

A simple example:

```
from tango import AttrWriteType, DeviceProxy
from tango.server import Device, attribute
from tango.test_context import MultiDeviceTestContext

class Device1(Device):
    _attr1 = 100

    @attribute(access=AttrWriteType.READ_WRITE)
    def attr1(self):
        return self._attr1

    def write_attr1(self, value):
        self._attr1 = value

def test_multi_short_name_device_proxy_access_without_tango_db():
    devices_info = ({ "class": Device1, "devices": [{"name": "test/device1/1"}] },)

    with MultiDeviceTestContext(devices_info):
        proxy1 = tango.DeviceProxy("test/device1/1")
        assert proxy1.name() == "test/device1/1"
        assert proxy1.attr1 == 100
```

Previously, we would have had to use a call like `context.get_device_access("test/device1/1")` to get the FQTRL, before instantiating a [DeviceProxy](#).

A more detailed [example](#) is available.

Limitations

- Group patterns (* wildcard) are not supported.
- Group device names will be FQTRLs, rather than the short name that was originally supplied.
- Launching two TestContexts in the same process will not work correctly without FQTRLs. The TestContext that is started second will overwrite the global variable set by the first.

Implementation details

From PyTango 9.5.0, the test device server's FQTRL is stored in a global variable while the TextContext is active. Then, in order to make device access with short names transparent, the *DeviceProxy*, *AttributeProxy* and *Group* classes have been modified to take this global variable into account. If it is set, and a short (i.e., unresolved) TRL is used for the proxy constructor or group add method, then the name is rewritten to a fully qualified (resolved) TRL. If this FQTRL is not accessible, the short name is tried again (except for *Group*) - this allows real devices to be accessed via short TRLs, if necessary.

There should not be a noticeable performance degradation for this outside of test environments. It is just a single variable lookup per proxy constructor or group add invocation.

This new TestContext functionality is enabled by default. It can be disabled by setting the *enable_test_context_tango_host_override* attribute to *False* before starting the TestContext.

Example:

```
def test_multi_short_name_access_fails_if_override_disabled():
    devices_info = ({ "class": Device1, "devices": [{"name": "test/device1/a"
→ "}] } }, )

    context = MultiDeviceTestContext(devices_info)
    context.enable_test_context_tango_host_override = False
    context.start()
    try:
        with pytest.raises(DevFailed):
            _ = tango.DeviceProxy("test/device1/a")
    finally:
        context.stop()
```

Port detection and pre_init_callback

As before, if the user doesn't specify a port number when instantiating the TestContext, omniORB will select an ephemeral port automatically when the device server starts. We need to determine this port number before any device calls its *init_device* method, in case it wants to create clients (*DeviceProxy*, etc.) at that early stage.

The solution to this is the addition of a *pre_init_callback* in the *run()* and *run_server()* methods. This is called after *Util.init()* (when omniORB has started and bound to a TCP port for the GIOP traffic, and the event system has bound two ports for ZeroMQ traffic), but before *Util.server_init()* (which initialises all classes in the device server).

During *pre_init_callback* the TestContext probes the process's open ports to determine the GIOP port number. We didn't find a way to determine the port number without probing each port.

12.2 History of changes

Contributors

PyTango Team - see [source repo history](#) for full details.

Last Update

Mar 29, 2024

12.2.1 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. Ounsy
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
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. Michel
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

continues on

Table 1 – continued from previous page

Date	Revision	Description	Author
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
15/02/23	9.4.0	9.4.0 Release	A. Joubert
15/03/23	9.4.1	9.4.1 Release	A. Joubert
27/07/23	9.4.2	9.4.2 Release	Y. Matveev
23/11/23	9.5.0	9.5.0 Release	A. Joubert
28/03/24	9.5.1	9.5.1 Release	A. Joubert

12.2.2 Version history

Version	Changes
9.5.1	<p>9.5.1</p> <p>Backports: !688: 9.5.x maintenance: Backport MRs 631, 644, 645, 664, 673, 674, 677</p> <p>Changed:</p> <ul style="list-style-type: none"> • !684: 9.5.x maintenance: restrict NumPy to 1.x for 9.5.1 release • Backport !631: Improve error message if cannot convert value to Dev-Boolean type • Backport !645: Extend pydevd debugging and coverage to dynamic attributes and commands • Backport !673: Improve error message if python float value if written to int attribute for python >= 3.10 <p>Fixed:</p> <ul style="list-style-type: none"> • Backport !644: Fix high-level attribute read for asyncio DeviceProxies • Backport !664: Use attribute lock ATTR_BY_KERNEL for push_event (fix a crash with asyncio green modes) • Backport !674: Fix typing compatibility with numpy 1.20 • Backport !677: Fix class_property inheritance in high-level Device <p>Documentation:</p> <ul style="list-style-type: none"> • !689: 9.5.x maintenance: update docs and migration guide for 9.5.1-rc1 • !690: 9.5.x maintenance: update docs bump for 9.5.1 <p>More details in the full changelog 9.5.0...9.5.1</p>

continues on next page

Table 2 – continued from previous page

Version	Changes
9.5.0	<p>9.5.0</p> <p>Changed:</p> <ul style="list-style-type: none"> !558: Check if user’s class methods are coroutines in Async mode !614: Require cppTango 9.5.0, bump to 9.5.0.dev0, doc fixes !617: Use 127.0.0.1 as default TestContext host instead of external IP <p>Added:</p> <ul style="list-style-type: none"> !388: Enable short-name access to TestContext devices !568: Declaration of properties, attributes and command type with typing hints !580: IMAGEs support added to set_write_value !581: Support forwarded attributes in TestContext !582: Add support for EncodedAttribute in high-level API device !592: Expose complete API of DeviceImpl.remove_attribute() !616: Support server debugging with PyCharm and VS Code (pydevd) !618: Resolve “Python 3.12 support” <p>Fixed:</p> <ul style="list-style-type: none"> !591: Handle spaces in Python path in winsetup (Windows only) !600: Fix green_mode bug in TestContext !612: Close socket in get_host_ip() (as used by DeviceTestContext) !625: Ignore event when using TestContext if not installed !627: Fix problem if self has a type hint !634: Fix SegFault if set_value was called with None <p>Removed:</p> <ul style="list-style-type: none"> !602: Remove CmdArgType.DevInt (cppTango DEV_INT) !604: Deprecated signature for WAttribute.get_write_value() removed <p>Documentation:</p> <ul style="list-style-type: none"> !615: Update docs and migration guide for 9.5.0-rc1 !628: Update docs and bump for 9.5.0rc2 !636: Update docs and bump for

Table 2 – continued from previous page

Version	Changes
9.4.2	<p>9.4.2 release.</p> <p>Features:</p> <ul style="list-style-type: none"> !578: Add server init hook to high-level and low-level devices !562: Check code coverage !577: Implement new python and NumPy version policy <p>Bug fixes and changes:</p> <ul style="list-style-type: none"> !551: Handle unsupported DeviceTestContext properties gracefully !556: Fix source location recorded by logging decorators !564: <i>Asyncio server doesn't change state to ALARM with AttrQuality</i> <https://gitlab.com/tango-controls/pytango/-/merge_requests/564> !557: Fix DevEncoded attributes and commands !565: Raise UnicodeError instead of segfaulting when Latin-1 encoding fails !570: Fix linter problem in win-setup.py !579: Extend “empty string workaround” to sequences for DeviceTestContext properties <p>Doc fixes:</p> <ul style="list-style-type: none"> !571: Update new build system doc !572: Improve docs for push_data_ready_event and EnsureOmniThread !587: Update docs and bump version for 9.4.2rc1 !595: Fixed history of changes <p>DevOps changes:</p> <ul style="list-style-type: none"> !563: Skip log location tests in AppVeyor CI !566: Add AppVeyor Windows builds for Python 3.9 to 3.11, Boost 1.82.0 !575: Add job to test main cpptango branch !574: Added test for checking default and non-default units !576: Add macOS wheels + gitlab-ci cleaning

12.2. History of changes

- !585: Move to cppTango 9.4.2, drop Python<3.9 on Win, update wheel deps
- !588: Skip failing test in Winodws

Table 2 – continued from previous page

Version	Changes
9.4.1	<p>9.4.1 release.</p> <p>Bug fixes and changes:</p> <ul style="list-style-type: none"> !547: Fix attributes with device inheritance and repeated method wrapping regression in 9.4.0 !548: Fix decorated attribute methods regression in 9.4.0 <p>Doc fixes:</p> <ul style="list-style-type: none"> !546: Add note about pip version for binary packages !544: Bump version to 9.4.1dev0 !555: Update docs and bump version for 9.4.0rc1 !559: Groom docstrings !560: Bump for 9.4.1 <p>Deprecation fixes:</p> <ul style="list-style-type: none"> !553: Remove compiler version check from setup.py <p>DevOps changes:</p> <ul style="list-style-type: none"> !545: Run black on repo and add to pre-commit-config !554: Update to omniorb 4.2.5 for Linux wheels !549: Use new tango-controls group runners !550: Update mambaforge image and use conda instead of apt packages in CI !552: Run gitlab-triage to update old issues/MRs <p>More details in the full changelog 9.4.0...9.4.1</p>

continues on next page

Table 2 – continued from previous page

Version	Changes
9.4.0	<p>9.4.0 release (not recommended due to significant regressions).</p> <p>Features:</p> <ul style="list-style-type: none"> !522: Support of non-bound methods for attributes !535: Allow developer to optionally add attributes to a DeviceProxy instance !515: DevEnum spectrum and image attributes support added !502: Provide binary wheels on PyPI using pytango-builder images !510: Added high level API for dynamic attributes !511: Added <code>is_allowed</code> kwarg for static/dynamic commands and <code>is_allowed</code> method for dynamic commands !528: Added <code>getter</code>, <code>read</code> and <code>is_allowed</code> attribute decorators !542: Improve device types autocompletion in IDEs <p>Changes:</p> <ul style="list-style-type: none"> !490: Drop Python 2.7 and 3.5 support !486: Switch support from cppTango 9.3 to 9.4 !536: Require <code>cppTango</code> $\geq 9.4.1$ to import the library !489: Make <code>numpy</code> a hard requirement !493: Improve spectrum and image attribute behaviour with empty lists (breaking change to API!) !492: Change <code>DServer</code> inheritance from <code>Device_4Impl</code> to <code>Device_5Impl</code> !514: Remove Python 2 compatibility code !539: Update CI to <code>cppTango</code> 9.4.1, change default ORBendpoint host to 0.0.0.0, fix tests !541: Workaround <code>cppTango</code>#1055 for <code>DatabaseDS</code> startup <p>Bug fixes:</p> <ul style="list-style-type: none"> !495: Fix log streams with <code>%</code> and no args !516: Resolve “Crash when writing <code>numpy.array</code> to <code>DeviceProxy</code> string array attributes” !533: Fix high-level enum read exception when quality is <code>ATTR_INVALID</code>

Table 2 – continued from previous page

Version	Changes
9.3.6	<p>9.3.6 release.</p> <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #482: Use cpptango 9.3.5 for Widows wheels (except Py27 x64) <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #477: Resolve “Dynamic attribute in 9.3.5 fails” • Pull Request #479: Fix green mode usage from run method kwarg • Pull Request #480: Resolve “read-only dynamic attribute with dummy write function fails in 9.3.5”
9.3.5	<p>9.3.5 release.</p> <p>Features:</p> <ul style="list-style-type: none"> • Pull Request #470: Add set_data_ready_event method to Device <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #471: Fail if mixed green modes used in device server <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #461: Fix handling of -ORBEndPointX command line options • Pull Request #462: Ensure PY-TANGO_NUMPY_VERSION is stringized to support newer C++ compilers • Pull Request #465: Restore dynamic attribute functionality with unbound methods • Pull Request #466: Explicit boost::python::optional template usage to fix compilation with gcc>10 <p>Doc fixes:</p> <ul style="list-style-type: none"> • Pull Request #467: Better MultiDeviceTestContext workaround • Pull Request #474: Update documentation for tango.Database <p>DevOps features:</p> <ul style="list-style-type: none"> • Pull Request #473: Make universal dockerfile

continues on next page

Table 2 – continued from previous page

Version	Changes
9.3.4	<p>9.3.4 release.</p> <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #430: Raise when setting non-existent DeviceProxy attr • Pull Request #444: Add “friendly” argparser for device server arguments (#132, #354) <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #401: Fix read/write/is_allowed not called for dynamic attribute in async mode server (#173) • Pull Request #417: Fix DeviceProxy constructor reference cycle (#412) • Pull Request #418: Release GIL in DeviceProxy and AttributeProxy dtor • Pull Request #434: Fix Device green_mode usage in MultiDeviceTestContext • Pull Request #436: Fix MSVC 9 syntax issue with shared pointer deletion • Pull Request #438: Add unit tests for device server logging • Pull Request #446: Allow pipes to be inherited by Device subclasses (#439) <p>Deprecation fixes:</p> <ul style="list-style-type: none"> • Pull Request #414: Fix deprecated warning with numpy 1.20 • Pull Request #424: tango/pytango_pprint.py: Use correct syntax for comparing object contents • Pull Request #425: Fix some and silence some C++ compiler warnings • Pull Request #439: Fix asyncio Python 3.10 compatibility (#429) • Pull Request #449: Use Py_ssize_t for all CPython indexing <p>Doc fixes:</p> <ul style="list-style-type: none"> • Pull Request #404: Typo on Sphinx documentation (#173) • Pull Request #406: Fix docs - missing DbDevExportInfos and DbDevImportInfos • Pull Request #420: Fix broken link: no s in gevent • Pull Request #422: Uncomment docs of tango.Util.instance() and bus345 docs for other static methods • Pull Request #426: [docs] Fixed arguments name when calling command

Table 2 – continued from previous page

Version	Changes
9.3.3	<p>9.3.3 release.</p> <p>Features:</p> <ul style="list-style-type: none"> • Pull Request #378: Add string support for MultiDeviceTestContext devices_info class field • Pull Request #384: Add test context support for memorized attributes • Pull Request #395: Fix Windows build and add CI test suite (#355, #368, #369) <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #365: Preserve cause of exception when getting/setting attribute in DeviceProxy (#364) • Pull Request #385: Improve mandatory + default device property error message (#380) • Pull Request #397: Add std namespace prefix in C++ code <p>Bug/doc fixes:</p> <ul style="list-style-type: none"> • Pull Request #360: Fix convert2array for Unicode to DevVarStringArray (Py3) (#361) • Pull Request #386: Fix DeviceProxy repr/str memory leak (#298) • Pull Request #352: Fix sphinx v3 warning • Pull Request #359: MultiDeviceTestContext example • Pull Request #363: Change old doc links from ESRF to RTD • Pull Request #370: Update CI to use cppTango 9.3.4rc6 • Pull Request #389: Update CI and dev Docker to cpptango 9.3.4 • Pull Request #376: Update Windows CI and dev containers to boost 1.73.0 • Pull Request #377: VScode remote development container support • Pull Request #391: Add documentation about testing • Pull Request #393: Fix a typo in get_server_info documentation (#392)

continues on next page

Table 2 – continued from previous page

Version	Changes
9.3.2	<p>9.3.2 release.</p> <p>Features:</p> <ul style="list-style-type: none"> • Pull Request #314: Add MultiDeviceTestContext for testing more than one Device • Pull Request #317: Add get_device_attribute_list and missing pipe methods to Database interface (#313) • Pull Request #327: Add EnsureOmniThread and is_omni_thread (#307, #292) <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #316: Reduce six requirement from 1.12 to 1.10 (#296) • Pull Request #326: Add Docker development container • Pull Request #330: Add enum34 to Python 2.7 docker images • Pull Request #329: Add test to verify get_device_properties called on init • Pull Request #341: Build DevFailed origin from format_exception (#340) <p>Bug/doc fixes:</p> <ul style="list-style-type: none"> • Pull Request #301: Fix documentation error • Pull Request #334: Update green mode docs and asyncio example (#333) • Pull Request #335: Generalise search for libboost_python on POSIX (#300, #310) • Pull Request #343: Extend the info on dependencies in README • Pull Request #345: Fix power_supply client example PowerOn -> TurnOn • Pull Request #347: Fix memory leak for DevEncoded attributes • Pull Request #348: Fix dynamic enum attributes created without labels (#56)

continues on next page

Table 2 – continued from previous page

Version	Changes
9.3.1	<p>9.3.1 release.</p> <p>Changes:</p> <ul style="list-style-type: none">• Pull Request #277: Windows builds using AppVeyor (#176)• Pull Request #290: Update docs: int types maps to DevLong64 (#282)• Pull Request #293: Update exception types in proxy docstrings <p>Bug fixes:</p> <ul style="list-style-type: none">• Pull Request #270: Add six ≥ 1.12 requirement (#269)• Pull Request #273: DeviceAttribute.is_empty not working correctly with latest cpp tango version (#271)• Pull Request #274: Add unit tests for spectrum attributes, including empty (#271)• Pull Request #281: Fix DevEncoded commands on Python 3 (#280)• Pull Request #288: Make sure we only convert to string python unicode/str/bytes objects (#285)• Pull Request #289: Fix compilation warnings and conda build (#286)

continues on next page

Table 2 – continued from previous page

Version	Changes
9.3.0	<p>9.3.0 release.</p> <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #242: Improve Python version check for enum34 install • Pull Request #250: Develop 9.3.0 • Pull Request #258: Change Travis CI builds to xenial <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #245: Change for collections abstract base class • Pull Request #247: Use IP address instead of hostname (fix #246) • Pull Request #252: Fix wrong link to tango dependency (#235) • Pull Request #254: Fix mapping of AttrWriteType WT_UNKNOWN • Pull Request #257: Fix some docs and docstrings • Pull Request #260: add ApiUtil.cleanup() • Pull Request #262: Fix compile error under Linux • Pull Request #263: Fix #251: Python 2 vs Python 3: DevString with bytes

continues on next page

Table 2 – continued from previous page

Version	Changes
9.2.5	<p>9.2.5 release.</p> <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #212: Skip databaseds backends in PyTango compatibility module • Pull Request #221: DevEnum attributes can now be directly assigned labels • Pull Request #236: Cleanup db_access module • Pull Request #237: Add info about how to release a new version <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #209 (issue #207): Fix documentation warnings • Pull Request #211: Yet another fix to the gevent threadpool error wrapping • Pull Request #214 (issue #213): DevEncoded attribute should produce a bytes object in python 3 • Pull Request #219: Fixing icons in documentation • Pull Request #220: Fix 'DevFailed' object does not support indexing • Pull Request #225 (issue #215): Fix exception propagation in python 3 • Pull Request #226 (issue #216): Add missing converter from python bytes to char* • Pull Request #227: Gevent issue #1260 should be fixed by now • Pull Request #232: use special case-insensitive weak values dictionary for Tango nodes

continues on next page

Table 2 – continued from previous page

Version	Changes
9.2.4	<p>9.2.4 release.</p> <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #194 (issue #188): Easier access to DevEnum attribute using python enum • Pull Request #199 (issue #195): Support python enum as dtype argument for attributes • Pull Request #205 (issue #202): Python 3.7 compatibility <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #193 (issue #192): Fix a gevent green mode memory leak introduced in v9.2.3
9.2.3	<p>9.2.3 release.</p> <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #169: Use tango-controls theme for the documentation • Pull Request #170 (issue #171): Use a private gevent ThreadPool • Pull Request #180: Use same default encoding for python2 and python3 (utf-8) <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #178 (issue #177): Make CmdDoneEvent.argout writable • Pull Request #178: Add GIL control for ApiUtil.get_asynch_replies • Pull Request #187 (issue #186): Fix and extend client green mode

continues on next page

Table 2 – continued from previous page

Version	Changes
9.2.2	<p>9.2.2 release.</p> <p>Features:</p> <ul style="list-style-type: none"> • Pull Request #104: Pipe Events • Pull Request #106: Implement pipe write (client and server, issue #9) • Pull Request #122: Dynamic commands • Pull Request #124: Add forward attribute • Pull Request #129: Implement mandatory property (issue #30) <p>Changes:</p> <ul style="list-style-type: none"> • Pull Request #109: Device Interface Change Events • Pull Request #113: Adding asyncio green mode documentation and a how-to on contributing • Pull Request #114: Added PEP8-ified files in tango module. • Pull Request #115: Commands polling tests (client and server) • Pull Request #116: Attribute polling tests (client and server) • Pull Request #117: Use official tango-controls conda channel • Pull Request #125: Forward attribute example • Pull Request #134: Linting pytango (with pylint + flake8) • Pull Request #137: Codacy badge in README and code quality policy in How to Contribute • Pull Request #143: Added missing PipeEventData & DevIntr-ChangeEventData <p>Bug fixes:</p> <ul style="list-style-type: none"> • Pull Request #85 (issue #84): Fix Gevent ThreadPool exceptions • Pull Request #94 (issue #93): Fix issues in setup file (GCC-7 build) • Pull Request #96: Filter badges from the long description • Pull Request #97: Fix/linker options • Pull Request #98: Refactor green mode for client and server APIs • Pull Request #101 (issue #100) check for None and • Pull Request #102: Update server tests

Table 2 – continued from previous page

Version	Changes
9.2.1	<p>9.2.1 release.</p> <p>Features:</p> <ul style="list-style-type: none"> • Pull Requests #70: Add <code>test_context</code> and <code>test_utils</code> modules, used for py-tango unit-testing <p>Changes:</p> <ul style="list-style-type: none"> • Issue #51: Refactor platform specific code in setup file • Issue #67: Comply with PEP 440 for pre-releases • Pull Request #70: Add unit-testing for the server API • Pull Request #70: Configure Travis CI for continuous integration • Pull Request #76: Add unit-testing for the client API • Pull Request #78: Update the python version classifiers • Pull Request #80: Move tango object server to its own module • Pull Request #90: The metaclass definition for tango devices is no longer mandatory <p>Bug fixes:</p> <ul style="list-style-type: none"> • Issue #24: Fix <code>dev_status</code> dangling pointer bug • Issue #57: Fix <code>dev_state/status</code> to be gevent safe • Issue #58: Server gevent mode internal call hangs • Pull Request #62: Several fixes in <code>tango.databases</code> • Pull Request #63: Follow up on issue #21 (Fix <code>Group.get_device</code> method) • Issue #64: Fix <code>AttributeProxy.__dev_proxy</code> to be initialized with python internals • Issue #74: Fix hanging with an asynchronous tango server fails to start • Pull Request #81: Fix <code>DeviceImpl</code> documentation • Issue #82: Fix attribute completion for device proxies with IPython ≥ 4 • Issue #84: Fix gevent threadpool exceptions

continues on next page

Table 2 – continued from previous page

Version	Changes
9.2.0	<p>9.2.0 release.</p> <p>Features:</p> <ul style="list-style-type: none"> • Issue #37: Add <code>display_level</code> and <code>polling_period</code> as optional arguments to command decorator <p>Bug fixes:</p> <ul style="list-style-type: none"> • Fix cache problem when using <i>DeviceProxy</i> through an <i>AttributeProxy</i> • Fix compilation on several platforms • Issue #19: Defining new members in <i>DeviceProxy</i> has side effects • Fixed bug in <i>beacon.add_device</i> • Fix for <i>get_device_list</i> if <i>server_name</i> is '*' • Fix <i>get_device_attribute_property2</i> if <i>prop_attr</i> is not <i>None</i> • Accept <i>StdStringVector</i> in <i>put_device_property</i> • Map 'int' to <i>DevLong64</i> and 'uint' to <i>DevULong64</i> • Issue #22: Fix <i>push_data_ready_event()</i> deadlock • Issue #28: Fix compilation error for <i>constants.cpp</i> • Issue #21: Fix <i>Group.get_device</i> method • Issue #33: Fix internal server documentation <p>Changes:</p> <ul style="list-style-type: none"> • Move <i>ITango</i> to another project • Use <i>setuptools</i> instead of <i>distutils</i> • Add <i>six</i> as a requirement • Refactor directory structure • Rename <i>PyTango</i> module to <i>tango</i> (<i>import PyTango</i> still works for backward compatibility) • Add a ReST readme for GitHub and PyPI <p>ITango changes (moved to another project):</p> <ul style="list-style-type: none"> • Fix itango event logger for python 3 • Avoid deprecation warning with IPython 4.x • Use entry points instead of scripts

continues on next page

Table 2 – continued from previous page

Version	Changes
9.2.0a	<p>9.2 alpha release. Missing:</p> <ul style="list-style-type: none"> writable pipes (client and server) dynamic commands (server) device interface change event (client and server) pipe event (client and server) <p>Bug fixes:</p> <ul style="list-style-type: none"> 776: [pytango][8.1.8] SyntaxError: invalid syntax
8.1.9	<p>Features:</p> <ul style="list-style-type: none"> PR #2: asyncio support for both client and server API PR #6: Expose AutoTangoMonitor and AutoTangoAllowThreads <p>Bug fixes:</p> <ul style="list-style-type: none"> PR #31: Get -l flags from pkg-config PR #15: Rename itango script to itango3 for python3 PR #14: Avoid deprecation warning with IPython 4.x

continues on next page

Table 2 – continued from previous page

Version	Changes
8.1.8	<p>Features:</p> <ul style="list-style-type: none"> • PR #3: Add a <code>run_server</code> class method to <code>Device</code> • PR #4: Add device inheritance • 110: device property with auto update in database <p>Bug fixes:</p> <ul style="list-style-type: none"> • 690: Description attribute property • 700: [pytango] useless files in the source distribution • 701: Memory leak in command with list argument • 704: Assertion failure when calling command with string array input type • 705: Support <code>boost_python</code> lib name on Gentoo • 714: Memory leak in PyTango for direct server command calls • 718: <code>OverflowErrors</code> with float types in 8.1.6 • 724: PyTango <code>DeviceProxy.command_inout(<str>)</code> memory leaks • 736: pytango FTBFS with python 3.4 • 747: PyTango event callback in <code>gevent</code> mode gets called in non main thread
8.1.6	<p>Bug fixes:</p> <ul style="list-style-type: none"> • 698: <code>PyTango.Util</code> discrepancy • 697: <code>PyTango.server.run</code> does not accept old <code>Device</code> style classes
8.1.5	<p>Bug fixes:</p> <ul style="list-style-type: none"> • 687: [pytango] 8.1.4 unexpected files in the source package • 688: PyTango 8.1.4 new style server commands don't work

continues on next page

Table 2 – continued from previous page

Version	Changes
8.1.4	<p>Features:</p> <ul style="list-style-type: none"> • 107: Nice to check Tango/PyTango version at runtime <p>Bug fixes:</p> <ul style="list-style-type: none"> • 659: segmentation fault when unsubscribing from events • 664: problem while installing PyTango 8.1.1 with pip (using pip 1.4.1) • 678: [pytango] 8.1.2 unexpected files in the source package • 679: PyTango.server tries to import missing <code>__builtin__</code> module on Python 3 • 680: Cannot import PyTango.server.run • 686: Device property substitution for a multi-device server
8.1.3	<i>SKIPPED</i>
8.1.2	<p>Features:</p> <ul style="list-style-type: none"> • 98: PyTango.server.server_run needs additional <code>post_init_callback</code> parameter • 102: DevEncoded attribute should support a python buffer object • 103: Make creation of *EventData objects possible in PyTango <p>Bug fixes:</p> <ul style="list-style-type: none"> • 641: python3 error handling issue • 648: PyTango unicode method parameters fail • 649: <code>write_attribute</code> of <code>spectrum/image</code> fails in PyTango without numpy • 650: [pytango] 8.1.1 not compatible with ipyton 1.2.0-rc1 • 651: PyTango segmentation fault when run a DS that use <code>attr_data.py</code> • 660: <code>command_inout_async</code> (polling mode) fails • 666: PyTango shutdown sometimes blocks.

continues on next page

Table 2 – continued from previous page

Version	Changes
8.1.1	<p>Features:</p> <ul style="list-style-type: none"> • Implemented tango C++ 8.1 API <p>Bug fixes:</p> <ul style="list-style-type: none"> • 527: <code>set_value()</code> for <code>ULong64</code> • 573: [pytango] python3 error with unregistered device • 611: URGENT fail to write attribute with PyTango 8.0.3 • 612: [pytango][8.0.3] failed to build from source on s390 • 615: Threading problem when setting a <code>DevULong64</code> attribute • 622: PyTango broken when running on Ubuntu 13 • 626: <code>attribute_history</code> extraction can raised an exception • 628: Problem in installing PyTango 8.0.3 on Scientific Linux 6 • 635: Reading of <code>ULong64</code> attributes does not work • 636: PyTango log messages are not filtered by level • 637: [pytango] segfault doing <code>write_attribute</code> on Group
8.1.0	SKIPPED
8.0.3	<p>Features:</p> <ul style="list-style-type: none"> • 88: Implement <code>Util::server_set_event_loop</code> method in python <p>Bug fixes:</p> <ul style="list-style-type: none"> • 3576353: [pytango] segfault on 'Restart-Server' • 3579062: [pytango] Attribute missing methods • 3586337: [pytango] Some <code>DeviceClass</code> methods are not python safe • 3598514: <code>DeviceProxy.__setattr__</code> break python's descriptors • 3607779: [pytango] IPython 0.10 error • 598: Import DLL by PyTango failed on windows • 605: [pytango] use <code>distutils.version</code> module

continues on next page

Table 2 – continued from previous page

Version	Changes
8.0.2	Bug fixes: <ul style="list-style-type: none"> • 3570970: [pytango] problem during the python3 building • 3570971: [pytango] itango does not work without qtconsole • 3570972: [pytango] warning/error when building 8.0.0 • 3570975: [pytango] problem during use of python3 version • 3574099: [pytango] compile error with gcc < 4.5
8.0.1	<i>SKIPPED</i>
8.0.0	Features: <ul style="list-style-type: none"> • Implemented tango C++ 8.0 API • Python 3k compatible Bug fixes: <ul style="list-style-type: none"> • 3023857: DevEncoded write attribute not supported • 3521545: [pytango] problem with tango profile • 3530535: PyTango group writting fails • 3564959: EncodedAttribute.encode_xxx() methods don't accept bytearray
7.2.4	Bug fixes: <ul style="list-style-type: none"> • 551: [pytango] Some DeviceClass methods are not python safe
7.2.3	Features: <ul style="list-style-type: none"> • 3495607: DeviceClass.device_name_factory is missing Bug fixes: <ul style="list-style-type: none"> • 3103588: documentation of PyTango.Attribute.Group • 3458336: Problem with pytango 7.2.2 • 3463377: PyTango memory leak in read encoded attribute • 3487930: [pytango] wrong python dependency • 3511509: Attribute.set_value_date_quality for encoded does not work • 3514457: [pytango] TANGO_HOST multi-host support • 3520739: command_history(...) in PyTango

continues on next page

Table 2 – continued from previous page

Version	Changes
7.2.2	<p>Features:</p> <ul style="list-style-type: none"> • 3305251: DS dynamic attributes discards some Attr properties • 3365792: DeviceProxy.<cmd_name> could be documented • 3386079: add support for ipython 0.11 • 3437654: throw python exception as tango exception • 3447477: spock profile installation <p>Bug fixes:</p> <ul style="list-style-type: none"> • 3372371: write attribute of DevEncoded doesn't work • 3374026: [pytango] pyflakes warning • 3404771: PyTango.MultiAttribute.get_attribute_list missing • 3405580: PyTango.MultiClassAttribute missing
7.2.1	<i>SKIPPED</i>
7.2.0	<p>Features:</p> <ul style="list-style-type: none"> • 3286678: Add missing EncodedAttribute JPEG methods
7.1.6	<p>Bug fixes:</p> <ul style="list-style-type: none"> • 7.1.5 distribution is missing some files
7.1.5	<p>Bug fixes:</p> <ul style="list-style-type: none"> • 3284174: 7.1.4 does not build with gcc 4.5 and tango 7.2.6 • 3284265: [pytango][7.1.4] a few files without licence and copyright • 3284318: copyleft vs copyright • 3284434: [pytango][doc] few ERROR during the doc generation • 3284435: [pytango][doc] few warning during the doc generation • 3284440: [pytango][spock] the profile can't be installed • 3285185: PyTango Device Server does not load Class Properties values • 3286055: PyTango 7.1.x DS using Tango C++ 7.2.x seg faults on exit

continues on next page

Table 2 – continued from previous page

Version	Changes
7.1.4	<p>Features:</p> <ul style="list-style-type: none"> • 3274309: Generic Callback for events <p>Bug fixes:</p> <ul style="list-style-type: none"> • 3011775: Seg Faults due to removed dynamic attributes • 3105169: PyTango 7.1.3 does not compile with Tango 7.2.X • 3107243: spock profile does not work with python 2.5 • 3124427: PyTango.WAttribute.set_max_value() changes min value • 3170399: Missing documentation about is_attr_allowed method • 3189082: Missing get_properties() for Attribute class • 3196068: delete_device() not called after server_admin.Kill() • 3257286: Binding crashes when reading a WRITE string attribute • 3267628: DP.read_attribute(, extract=List/tuple) write value is wrong • 3274262: Database.is_multi_tango_host missing • 3274319: EncodedAttribute is missing in PyTango (<= 7.1.3) • 3277269: read_attribute(DevEncoded) is not numpy as expected • 3278946: DeviceAttribute copy constructor is not working <p>Documentation:</p> <ul style="list-style-type: none"> • Added <i>The Utilities API</i> chapter • Added <i>Encoded API</i> chapter • Improved <i>Write a server (original API)</i> chapter

continues on next page

Table 2 – continued from previous page

Version	Changes
7.1.3	<p>Features:</p> <ul style="list-style-type: none">• tango logging with print statement• tango logging with decorators• from sourceforge:• 3060380: ApiUtil should be exported to Py-Tango <p>Bug fixes:</p> <ul style="list-style-type: none">• added licence header to all source code files• spock didn't work without TANGO_HOST env. variable (it didn't recognize tangorc)• spock should give a proper message if it tries to be initialized outside ipython• 3048798: licence issue GPL != LGPL• 3073378: DeviceImpl.signal_handler raising exception crashes DS• 3088031: Python DS unable to read Dev-VarBooleanArray property• 3102776: PyTango 7.1.2 does not work with python 2.4 & boost 1.33.0• 3102778: Fix compilation warnings in linux

continues on next page

Table 2 – continued from previous page

Version	Changes
7.1.2	<p>Features:</p> <ul style="list-style-type: none"> • 2995964: Dynamic device creation • 3010399: The DeviceClass.get_device_list that exists in C++ is missing • 3023686: Missing DeviceProxy.<attribute name> • 3025396: DeviceImpl is missing some CORBA methods • 3032005: IPython extension for PyTango • 3033476: Make client objects pickable • 3039902: PyTango.Util.add_class would be useful <p>Bug fixes:</p> <ul style="list-style-type: none"> • 2975940: DS command with DevVarCharArray return type fails • 3000467: DeviceProxy.unlock is LOCKING instead of unlocking! • 3010395: Util.get_device_* methods don't work • 3010425: Database.dev_name does not work • 3016949: command_inout_async callback does not work • 3020300: PyTango does not compile with gcc 4.1.x • 3030399: Database put(delete)_attribute_alias generates segfault
7.1.1	<p>Features:</p> <ul style="list-style-type: none"> • Improved setup script • Interfaced with PyPI • Cleaned build script warnings due to unclean python C++ macro definitions • 2985993: PyTango numpy command support • 2971217: PyTango.GroupAttrReplyList slicing <p>Bug fixes:</p> <ul style="list-style-type: none"> • 2983299: Database.put_property() deletes the property • 2953689: can not write_attribute scalar/spectrum/image • 2953030: PyTango doc installation

continues on next page

Table 2 – continued from previous page

Version	Changes
7.1.0	<div>Features:</div> <ul style="list-style-type: none">• 2908176: read_*, write_* and is_*_allowed() methods can now be defined• 2941036: TimeVal conversion to time and datetime• added str representation on Attr, Attribute, DeviceImpl and DeviceClass <div>Bug fixes:</div> <ul style="list-style-type: none">• 2903755: get_device_properties() bug reading DevString properties• 2909927: PyTango.Group.read_attribute() return values• 2914194: DevEncoded does not work• 2916397: PyTango.DeviceAttribute copy constructor does not work• 2936173: PyTango.Group.read_attributes() fails• 2949099: Missing PyTango.Except.print_error_stack

continues on next page

Table 2 – continued from previous page

Version	Changes
7.1.0rc1	<p>Features:</p> <ul style="list-style-type: none"> • <code>v = image_attribute.get_write_value()</code> returns square sequences (arrays of arrays, or numpy objects) now instead of flat lists. Also for spectrum attributes a numpy is returned by default now instead. • <code>image_attribute.set_value(v)</code> accepts numpy arrays now or square sequences instead of just flat lists. So, <code>dim_x</code> and <code>dim_y</code> are useless now. Also the numpy path is faster. • new enum <code>AttrSerialModel</code> • <code>Attribute</code> new methods: <code>set(get)_attr_serial_model</code>, <code>set_change_event</code>, <code>set_archive_event</code>, <code>is_change_event</code>, <code>is_check_change_event</code>, <code>is_archive_criteria</code>, <code>is_check_archive_criteria</code>, <code>remove_configuration</code> • added support for numpy scalars in tango operations like <code>write_attribute</code> (ex: now a <code>DEV_LONG</code> attribute can receive a <code>numpy.int32</code> argument in a <code>write_attribute</code> method call) <p>Bug fixes:</p> <ul style="list-style-type: none"> • <code>DeviceImpl.set_value</code> for scalar attributes • <code>DeviceImpl.push_***_event</code> • server commands with <code>Dev-Var***StringArray</code> as parameter or as return type • in windows, a bug in <code>PyTango.Util</code> prevented servers from starting up • <code>DeviceImpl.get_device_properties</code> for string properties assigns only first character of string to object member instead of entire string • added missing methods to <code>Util</code> • exported <code>SubDevDiag</code> class • error in read/events of attributes of type <code>DevBoolean</code> <code>READ_WRITE</code> • error in automatic unsubscribe events of <code>DeviceProxy</code> when the object disappears (happens only on some compilers with some optimization flags) • fix possible bug when comparing attribute names in <code>DeviceProxy</code> • pretty print of <code>DevFailed</code> -> fix deprecation warning in python 2.6 • device class properties where not properly fetched when there is no property value defined
12.2. History of changes	<ul style="list-style-type: none"> • memory leak when converting <code>DevFailed</code> exceptions from C++ to python

Table 2 – continued from previous page

Version	Changes
---------	---------

12.3 What's new in PyTango 9.5.1?

Date: 2024-03-28 9.5.1

Type: minor release

12.3.1 Changed

- Restricted NumPy to 1.x, since we do not have NumPy 2.0 support yet.
- Improved some error message related to invalid types passed to `DeviceProxy`.
- Extended pydevd debugging and coverage to dynamic attributes and commands.

12.3.2 Fixed

- High-level attribute reads using asyncio DeviceProxies can now be awaited.
- Asyncio green mode devices no longer crash when an attribute is read at the same time as an event is being pushed.
- Numpy 1.20.0 no longer causes an import error.
- High-level Device class inheritance now supports `class_property`.

12.4 What's new in PyTango 9.5.0?

Date: 2023-11-23 9.5.0

Type: major release

12.4.1 Changed

- PyTango requires at least `cppTango 9.5.0`. See the *migration guide*.
- When using the asyncio green mode, a `PyTangoUserWarning` will be emitted during the `init_device()` call, if the user's `Device` methods are not coroutines (i.e., defined with `async def`).
- Use `127.0.0.1` as the default host for (Multi)DeviceTestContext instead of trying to find an external IP. This allows tests to work on systems that only have a loopback interface, and also reduces firewall warnings when running tests (at least on macOS). If using it from the command line like, `python -m tango.test_context MyDS.MyDevice`, an external IP is still the default.
- All warnings generated by PyTango are now instances of `PyTangoUserWarning`, which inherits from Python's `UserWarning`.
- Some of the dependencies packaged with the binary wheels on PyPI have changed. The bundled versions are:

Dependency	Linux	Windows	MacOS
cpptango	9.5.0	9.5.0	9.5.0
omniorb / omniorb-libs	4.3.1	4.3.0	4.3.1
libzmq / zeromq	4.3.5	4.0.5-2	4.3.5
cppzmq	4.10.0	4.7.1	4.10.0
libjpeg-turbo	3.0.0	2.0.3	3.0.0
tango-idl	5.1.1	5.1.2	5.1.2
boost	1.82.0	1.83.0	1.82.0

12.4.2 Added

- Short-name access can be used for (Multi)DeviceTestContext devices. See [migration guide](#)
- *Experimental feature*: use Python type hints to declare Device more easily. Read more in the new section: [Use Python type hints when declaring a device](#).
- `set_write_value()` now supports IMAGE attributes.
- Forwarded attributes are *partially* supported in the (Multi)DeviceTestContext. We say *partially*, because a cppTango limitation (at least version 9.5.0) means root attributes on devices running in “nodb” mode (like those in launched by the TestContext) don’t work. However, it does work if the test device accesses a root attribute on a Tango device running with a Tango database.
- Support for [EncodedAttribute](#) in high-level API devices.
- Added `free_it` and `clean_db` arguments to `tango.server.Device.remove_attribute()` and `tango.LatestDeviceImpl.remove_attribute()` methods.
- Support Tango server debugging with PyCharm, PyDev and VS Code. Breakpoints now work for command and attribute handler methods, as well as other standard [Device](#) methods, when running through a debugger that is based on [pydevd](#). However, it doesn’t currently work with dynamic attributes and commands. If necessary, the feature can be disabled by setting the environment variable `PYTANGO_DISABLE_DEBUG_TRACE_PATCHING=1`.
- Added support for Python 3.12.

12.4.3 Fixed

- Fixed various issues with [DeviceProxy](#) with non-synchronous green mode devices launched with [DeviceTestContext](#) and [MultiDeviceTestContext](#). This also fixes support for tests decorated with `@pytest.mark.asyncio`.

12.4.4 Removed

- Breaking change to the API: the `DevInt` data type was removed, due to its removal from [cppTango](#). See the [migration guide](#).
- Deprecated signature, `get_write_value(self, lst)()`, was removed.

12.5 What's new in PyTango 9.4.2?

Date: 2023-07-27

Type: minor release

12.5.1 Changed

- New python and NumPy *version policy* is implemented.

12.5.2 Added

- Correct code coverage of server's methods can be obtained
- `server_init_hook` was added to high-level and low-level API
- macOS wheels now are provided

12.5.3 Fixed

- DevEncoded attributes and commands read methods are now segfault safe
- DevEncoded attributes and commands now decoded with utf-8
- DevEncoded attributes and commands can be extracted and written as str, bytes and bytearray
- If string encoding with Latin-1 fails, UnicodeError will be raised instead of segfaulting
- When user gives empty spectrum properties to the DeviceTestContext, they will be patched with one space symbol " " for each element
- In case patching failed or any other problems with FileDatabase, instead of crash PyTango will raise an exception and print out generated file
- Regression when applying additional decorators on attribute accessor functions. Method calls would have the wrong signature and fail.

12.5.4 Removed

- Support for Python < 3.9. See *version policy*

12.6 What's new in PyTango 9.4.1?

Date: 2023-03-15

Type: major release (breaking changes compared to 9.4.0)

12.6.1 Changed

- Removed additional function signatures for high-level attribute read/write/is_allowed methods that were added in 9.4.0 resulting in a regression. For example, the high-level write method API for dynamic attributes of the form `write_method(self, attr, value)` has been removed, leaving only `write_method(self, attr)`. Similarly, unbound functions that could be used without a reference to the device object, like `read_function()`, are no longer supported - only `read_function(device)`. See the [migration guide](#).
- The dependencies packaged with the binary PyPI wheels are as follows:
 - Linux:
 - * cpptango: 9.4.1
 - * omniorb: 4.2.5 (changed since PyTango 9.4.0)
 - * libzmq: v4.3.4
 - * cppzmq: v4.7.1
 - * libjpeg-turbo: 2.0.9
 - * tango-idl: 5.1.1
 - * boost: 1.80.0 (with patch for Python 3.11 support)
 - Windows:
 - * cpptango: 9.4.1
 - * omniorb: 4.2.5
 - * libzmq: v4.0.5-2
 - * cppzmq: v4.7.1
 - * libjpeg-turbo: 2.0.3
 - * tango-idl: 5.1.2
 - * boost: 1.73.0

12.6.2 Fixed

- Regression for undecorated read attribute accessor functions in derived device classes. E.g., if we have `class A(Device)` with attribute reading via method `A.read_my_attribute`, then reading `my_attribute` from `class B(A)` would fail. More generally, repeated wrapping of methods related to attributes, commands and standard methods (like `init_device`) is now avoided.
- Regression when applying additional decorators on attribute accessor functions. Method calls would have the wrong signature and fail.

12.7 What's new in PyTango 9.4.0?

Date: 2023-02-15

Type: major release

Warning: significant regressions - use newer release!

12.7.1 Changed

- PyTango requires at least [cppTango 9.4.1](#). See the *migration guide*.
- Breaking change to the API when using empty spectrum and image attributes. Clients reading an empty attribute will get an empty sequence (list/tuple/numpy array) instead of a `None` value. Similarly, devices that have an empty sequence written will receive that in the write method instead of a `None` value. See the migration guide on *empty attributes* and *extract as*.
- Python dependencies: [numpy](#) is no longer optional - it is required. Other new requirements are [packaging](#) and [psutil](#).
- Binary wheels for more platforms, including Linux, are available on [PyPI](#). Fast installation without compiling and figuring out all the dependencies!
- The dependencies packaged with the binary PyPI wheels are as follows:
 - Linux:
 - * `cpptango`: 9.4.1
 - * `omniorb`: 4.2.4
 - * `libzmq`: v4.3.4
 - * `cppzmq`: v4.7.1
 - * `libjpeg-turbo`: 2.0.9
 - * `tango-idl`: 5.1.1
 - * `boost`: 1.80.0 (with patch for Python 3.11 support)
 - Windows:
 - * `cpptango`: 9.4.1
 - * `omniorb`: 4.2.5
 - * `libzmq`: v4.0.5-2
 - * `cppzmq`: v4.7.1
 - * `libjpeg-turbo`: 2.0.3
 - * `tango-idl`: 5.1.2
 - * `boost`: 1.73.0
- When using the `--port` commandline option without `--host`, the ORBendpoint for `gio::tcp` passed to `cppTango` will use "0.0.0.0" as the host instead of an empty string. This is to workaround a [regression with cppTango 9.4.1](#). Note that if the `--ORBendPoint` commandline option is specified directly, it will not be modified. This will lead to a crash if an empty host is used, e.g., `--ORBendPoint giop:tcp::1234`.

12.7.2 Added

- User methods for attribute access (read/write/is allowed), and for commands (execute/is allowed) can be plain functions. They don't need to be methods on the device class anymore. There was some inconsistency with this previously, but now it is the same for static and dynamic attributes, and for commands. Static and dynamic commands can also take an `isallowed` keyword argument. See the *migration guide*.
- Device methods for reading and writing dynamic attributes can use the high-level API instead of getting and setting values inside `Attr` objects. See the *migration guide*.
- High-level API support for accessing and creating DevEnum spectrum and image attributes. See the *migration guide*.

- Developers can optionally allow Python attributes to be added to a `DeviceProxy` instance by calling `unfreeze_dynamic_interface()`. The default behaviour is still to raise an exception when accessing unknown attributes. See the [migration guide](#).
- Attribute decorators have additional methods: `getter()`, `read()` and `is_allowed()`. See the [migration guide](#).
- Python 3.11 support.
- MacOS support. This is easiest installing from [Conda-forge](#). Compiling locally is not recommended. See [Getting started](#).
- Integrated development environment (IDE) autocompletion for methods inherited from `tango.server.Device` and `tango.LatestDeviceImpl`. Attributes from the full class hierarchy are now more easily accessible directly in your editor.

12.7.3 Fixed

- Log stream calls that include literal `%` symbols but no args now work properly without raising an exception. E.g., `self.debug_stream("I want to log a %s symbol")`. See the [migration guide](#).
- Writing a `numpy.array` to a spectrum attribute of type `str` no longer crashes.
- Reading an enum attribute with `ATTR_INVALID` quality via the high-level API now returns `None` instead of crashing. This behaviour is consistent with the other data types.

12.7.4 Removed

- Support for Python 2.7 and Python 3.5.
- The option to install PyTango without `numpy`.

INDEXES

- `modindex`
- `genindex`

Last update: Mar 29, 2024

PYTHON MODULE INDEX

t

tango, [23](#)

tango.server, [113](#)

tango.test_context, [281](#)

A

AccessControlType (class in tango), 109
 add() (tango.Group method), 87
 add_attribute() (tango.LatestDeviceImpl method), 139
 add_attribute() (tango.server.Device method), 117
 add_class() (tango.Util method), 180
 add_command() (tango.LatestDeviceImpl method), 139
 add_command() (tango.server.Device method), 117
 add_Cpp_TgClass() (tango.Util method), 180
 add_device() (tango.Database method), 190
 add_logging_target() (tango.DeviceProxy method), 33
 add_server() (tango.Database method), 190
 add_TgClass() (tango.Util method), 180
 add_wiz_class_prop() (tango.DeviceClass method), 153
 add_wiz_dev_prop() (tango.DeviceClass method), 153
 adm_name() (tango.DeviceProxy method), 33
 alias() (tango.DeviceProxy method), 33
 always_executed_hook() (tango.LatestDeviceImpl method), 140
 always_executed_hook() (tango.server.Device method), 118
 ApiUtil (class in tango), 97
 append_db_file() (tango.test_context.DeviceTestContext method), 288
 append_db_file() (tango.test_context.MultiDeviceTestContext method), 291
 append_status() (tango.LatestDeviceImpl method), 140
 append_status() (tango.server.Device method), 118
 ArchiveEventInfo (class in tango), 106
 asyn_req_type (class in tango), 109
 AsyncCall, 241
 AsyncReplyNotArrived, 241
 Attr (class in tango), 161
 AttrConfEventData (class in tango), 106
 AttrDataFormat (class in tango), 110
 AttReqType (class in tango), 108

Attribute (class in tango), 165
 attribute (class in tango.server), 131
 attribute_history() (tango.DeviceProxy method), 33
 attribute_list_query() (tango.DeviceProxy method), 34
 attribute_list_query_ex() (tango.DeviceProxy method), 34
 attribute_query() (tango.DeviceProxy method), 34
 AttributeAlarmInfo (class in tango), 99
 AttributeDimension (class in tango), 99
 AttributeEventInfo (class in tango), 106
 AttributeInfo (class in tango), 99
 AttributeInfoEx (class in tango), 100
 AttributeProxy (class in tango), 71
 AttrQuality (class in tango), 109
 AttrReadEvent (class in tango), 105
 AttrWriteType (class in tango), 110
 AttrWrittenEvent (class in tango), 105

B

black_box() (tango.DeviceProxy method), 34
 build_connection() (tango.Database method), 191

C

cancel_all_polling_asynch_request() (tango.DeviceProxy method), 34
 cancel_asynch_request() (tango.DeviceProxy method), 35
 cb_sub_model (class in tango), 109
 ChangeEventInfo (class in tango), 106
 check_access_control() (tango.Database method), 191
 check_alarm() (tango.Attribute method), 165
 check_alarm() (tango.MultiAttribute method), 174
 check_command_exists() (tango.LatestDeviceImpl method), 140
 check_command_exists() (tango.server.Device method), 118
 check_tango_host() (tango.Database method), 191
 check_type() (tango.Attr method), 161
 class_property (class in tango.server), 136

`cleanup()` (*tango.ApiUtil method*), 97
`CmdArgType` (*class in tango*), 107
`CmdDoneEvent` (*class in tango*), 105
`command()` (*in module tango.server*), 133
`command_history()` (*tango.DeviceProxy method*), 35
`command_inout()` (*tango.DeviceProxy method*), 35
`command_inout()` (*tango.Group method*), 87
`command_inout_async()` (*tango.DeviceProxy method*), 36
`command_inout_async()` (*tango.Group method*), 87
`command_inout_raw()` (*tango.DeviceProxy method*), 37
`command_inout_reply()` (*tango.DeviceProxy method*), 37
`command_inout_reply()` (*tango.Group method*), 88
`command_inout_reply_raw()` (*tango.DeviceProxy method*), 38
`command_list_query()` (*tango.DeviceProxy method*), 38
`command_query()` (*tango.DeviceProxy method*), 38
`CommandInfo` (*class in tango*), 101
`CommunicationFailed`, 240
`connect()` (*tango.DeviceProxy method*), 39
`connect_db()` (*tango.Util method*), 180
`ConnectionFailed`, 239
`contains()` (*tango.Group method*), 88
`create_device()` (*tango.DeviceClass method*), 154
`create_device()` (*tango.Util method*), 180

D

`Database` (*class in tango*), 190
`DataReadyEventData` (*class in tango*), 106
`DbDatum` (*class in tango*), 221
`DbDevExportInfo` (*class in tango*), 222
`DbDevExportInfos` (*class in tango*), 222
`DbDevImportInfo` (*class in tango*), 222
`DbDevImportInfos` (*class in tango*), 222
`DbDevInfo` (*class in tango*), 222
`DbHistory` (*class in tango*), 222
`DbServerInfo` (*class in tango*), 223
`debug_stream()` (*tango.LatestDeviceImpl method*), 140
`debug_stream()` (*tango.server.Device method*), 118
`DebugIt` (*class in tango*), 159
`decode_gray16()` (*tango.EncodedAttribute method*), 224
`decode_gray8()` (*tango.EncodedAttribute method*), 224
`decode_rgb32()` (*tango.EncodedAttribute method*), 225
`delete_attribute_alias()` (*tango.Database method*), 192
`delete_class_attribute_property()` (*tango.Database method*), 192
`delete_class_pipe_property()` (*tango.Database method*), 192
`delete_class_property()` (*tango.Database method*), 193
`delete_db()` (*tango.test_context.DeviceTestContext method*), 289
`delete_db()` (*tango.test_context.MultiDeviceTestContext method*), 291
`delete_device()` (*tango.Database method*), 193
`delete_device()` (*tango.DeviceClass method*), 154
`delete_device()` (*tango.LatestDeviceImpl method*), 140
`delete_device()` (*tango.server.Device method*), 118
`delete_device()` (*tango.Util method*), 181
`delete_device_alias()` (*tango.Database method*), 194
`delete_device_attribute_property()` (*tango.Database method*), 194
`delete_device_pipe_property()` (*tango.Database method*), 194
`delete_device_property()` (*tango.Database method*), 194
`delete_property()` (*tango.AttributeProxy method*), 71
`delete_property()` (*tango.Database method*), 195
`delete_property()` (*tango.DeviceProxy method*), 39
`delete_server()` (*tango.Database method*), 195
`delete_server_info()` (*tango.Database method*), 196
`description()` (*tango.DeviceProxy method*), 40
`dev_name()` (*tango.DeviceProxy method*), 40
`dev_state()` (*tango.LatestDeviceImpl method*), 140
`dev_state()` (*tango.server.Device method*), 119
`dev_status()` (*tango.LatestDeviceImpl method*), 141
`dev_status()` (*tango.server.Device method*), 119
`DevCommandInfo` (*class in tango*), 101
`DevError` (*class in tango*), 239
`DevFailed`, 239
`Device` (*class in tango.server*), 117
`device_destroyer()` (*tango.DeviceClass method*), 155
`device_factory()` (*tango.DeviceClass method*), 155
`device_name_factory()` (*tango.DeviceClass method*), 155
`device_property` (*class in tango.server*), 136
`DeviceAttribute` (*class in tango*), 102
`DeviceAttribute.ExtractAs` (*class in tango*), 102
`DeviceAttributeConfig` (*class in tango*), 101
`DeviceAttributeHistory` (*class in tango*), 107

DeviceClass (class in tango), 153
 DeviceData (class in tango), 104
 DeviceDataHistory (class in tango), 107
 DeviceInfo (class in tango), 102
 DeviceProxy (class in tango), 32
 DeviceProxy() (in module tango.asyncio), 95
 DeviceProxy() (in module tango.futures), 96
 DeviceProxy() (in module tango.gevent), 96
 DeviceTestContext (class in tango.test_context), 288
 DeviceUnlocked, 242
 DevSource (class in tango), 110
 DevState (class in tango), 110
 disable() (tango.Group method), 89
 DispLevel (class in tango), 110
 dyn_attr() (tango.DeviceClass method), 155

E

enable() (tango.Group method), 89
 encode_gray16() (tango.EncodedAttribute method), 225
 encode_gray8() (tango.EncodedAttribute method), 226
 encode_jpeg_gray8() (tango.EncodedAttribute method), 227
 encode_jpeg_rgb24() (tango.EncodedAttribute method), 228
 encode_jpeg_rgb32() (tango.EncodedAttribute method), 229
 encode_rgb24() (tango.EncodedAttribute method), 229
 EncodedAttribute (class in tango), 224
 environment variable
 PYTANGO_GREEN_MODE, 13
 TANGO_HOST, 3, 4, 243
 error_stream() (tango.LatestDeviceImpl method), 141
 error_stream() (tango.server.Device method), 119
 ErrorIt (class in tango), 160
 ErrSeverity (class in tango), 110
 event_queue_size() (tango.AttributeProxy method), 72
 event_queue_size() (tango.DeviceProxy method), 40
 EventCallback (class in tango.utils), 230
 EventData (class in tango), 106
 EventSystemFailed, 242
 EventType (class in tango), 109
 Except (class in tango), 237
 export_device() (tango.Database method), 196
 export_device() (tango.DeviceClass method), 155
 export_event() (tango.Database method), 196
 export_server() (tango.Database method), 197
 extract() (tango.DeviceData method), 104

F

fatal_stream() (tango.LatestDeviceImpl method), 141
 fatal_stream() (tango.server.Device method), 119
 FatalIt (class in tango), 161
 freeze_dynamic_interface() (tango.DeviceProxy method), 40
 fromdatetime() (tango.TimeVal static method), 111
 fromtimestamp() (tango.TimeVal static method), 111

G

get_access_control() (tango.DeviceProxy method), 40
 get_access_except_errors() (tango.Database method), 197
 get_access_right() (tango.DeviceProxy method), 41
 get_alias() (tango.Database method), 197
 get_alias_from_attribute() (tango.Database method), 197
 get_alias_from_device() (tango.Database method), 197
 get_assoc() (tango.Attr method), 161
 get_assoc_ind() (tango.Attribute method), 166
 get_assoc_name() (tango.Attribute method), 166
 get_asynch_cb_sub_model() (tango.ApiUtil method), 97
 get_asynch_replies() (tango.ApiUtil method), 97
 get_asynch_replies() (tango.DeviceProxy method), 41
 get_attr_by_ind() (tango.MultiAttribute method), 174
 get_attr_by_name() (tango.MultiAttribute method), 174
 get_attr_ind_by_name() (tango.MultiAttribute method), 175
 get_attr_min_poll_period() (tango.LatestDeviceImpl method), 142
 get_attr_min_poll_period() (tango.server.Device method), 120
 get_attr_nb() (tango.MultiAttribute method), 175
 get_attr_poll_ring_depth() (tango.LatestDeviceImpl method), 142
 get_attr_poll_ring_depth() (tango.server.Device method), 120
 get_attr_serial_model() (tango.Attribute method), 166
 get_attribute_alias() (tango.Database method), 198
 get_attribute_alias_list() (tango.Database method), 198
 get_attribute_config() (tango.DeviceProxy method), 41

`get_attribute_config()`
 (*tango.LatestDeviceImpl* method), 142

`get_attribute_config()` (*tango.server.Device*
 method), 120

`get_attribute_config_2()`
 (*tango.LatestDeviceImpl* method), 142

`get_attribute_config_2()`
 (*tango.server.Device* method), 120

`get_attribute_config_3()`
 (*tango.LatestDeviceImpl* method), 142

`get_attribute_config_3()`
 (*tango.server.Device* method), 120

`get_attribute_config_ex()`
 (*tango.DeviceProxy* method), 42

`get_attribute_from_alias()`
 (*tango.Database* method), 198

`get_attribute_list()` (*tango.DeviceProxy*
 method), 42

`get_attribute_list()` (*tango.MultiAttribute*
 method), 175

`get_attribute_name()` (*tango.DbHistory*
 method), 222

`get_attribute_poll_period()`
 (*tango.DeviceProxy* method), 42

`get_attribute_poll_period()`
 (*tango.LatestDeviceImpl* method), 142

`get_attribute_poll_period()`
 (*tango.server.Device* method), 121

`get_cl_name()` (*tango.Attr* method), 161

`get_class_attr()` (*tango.DeviceClass* method),
 155

`get_class_attribute_list()`
 (*tango.Database* method), 199

`get_class_attribute_property()`
 (*tango.Database* method), 199

`get_class_attribute_property_history()`
 (*tango.Database* method), 199

`get_class_for_device()` (*tango.Database*
 method), 200

`get_class_inheritance_for_device()`
 (*tango.Database* method), 200

`get_class_list()` (*tango.Database* method), 200

`get_class_list()` (*tango.Util* method), 181

`get_class_pipe_list()` (*tango.Database*
 method), 201

`get_class_pipe_property()` (*tango.Database*
 method), 201

`get_class_pipe_property_history()`
 (*tango.Database* method), 201

`get_class_properties()` (*tango.Attr* method),
 161

`get_class_property()` (*tango.Database*
 method), 202

`get_class_property_history()`
 (*tango.Database* method), 202

`get_class_property_list()` (*tango.Database*
 method), 203

`get_cmd_by_name()` (*tango.DeviceClass* method),
 156

`get_cmd_min_poll_period()`
 (*tango.LatestDeviceImpl* method), 143

`get_cmd_min_poll_period()`
 (*tango.server.Device* method), 121

`get_cmd_poll_ring_depth()`
 (*tango.LatestDeviceImpl* method), 143

`get_cmd_poll_ring_depth()`
 (*tango.server.Device* method), 121

`get_command_config()` (*tango.DeviceProxy*
 method), 42

`get_command_list()` (*tango.DeviceClass*
 method), 156

`get_command_list()` (*tango.DeviceProxy*
 method), 43

`get_command_poll_period()`
 (*tango.DeviceProxy* method), 43

`get_command_poll_period()`
 (*tango.LatestDeviceImpl* method), 143

`get_command_poll_period()`
 (*tango.server.Device* method), 121

`get_config()` (*tango.AttributeProxy* method), 72

`get_cvs_location()` (*tango.DeviceClass*
 method), 156

`get_cvs_tag()` (*tango.DeviceClass* method), 156

`get_data()` (*tango.GroupAttrReply* method), 94

`get_data()` (*tango.GroupCmdReply* method), 94

`get_data_format()` (*tango.Attribute* method),
 166

`get_data_raw()` (*tango.GroupCmdReply* method),
 94

`get_data_size()` (*tango.Attribute* method), 166

`get_data_type()` (*tango.Attribute* method), 166

`get_database()` (*tango.Util* method), 182

`get_date()` (*tango.Attribute* method), 166

`get_date()` (*tango.DbHistory* method), 223

`get_date()` (*tango.DeviceAttribute* method), 103

`get_db_host()` (*tango.DeviceProxy* method), 43

`get_db_port()` (*tango.DeviceProxy* method), 43

`get_db_port_num()` (*tango.DeviceProxy* method),
 44

`get_dev_host()` (*tango.DeviceProxy* method), 44

`get_dev_idl_version()`
 (*tango.LatestDeviceImpl* method), 143

`get_dev_idl_version()` (*tango.server.Device*
 method), 121

`get_dev_port()` (*tango.DeviceProxy* method), 44

`get_device()` (*tango.test_context.DeviceTestContext*
 method), 289

`get_device()` (*tango.test_context.MultiDeviceTestContext*
 method), 291

`get_device_access()`
 (*tango.test_context.DeviceTestContext*
 method), 289

`get_device_access()`
 (*tango.test_context.MultiDeviceTestContext*
 method), 291

`get_device_alias()` (*tango.Database* method),

- 203
- `get_device_alias_list()` (*tango.Database method*), 203
- `get_device_attr()` (*tango.LatestDeviceImpl method*), 143
- `get_device_attr()` (*tango.server.Device method*), 121
- `get_device_attribute_list()` (*tango.Database method*), 203
- `get_device_attribute_property()` (*tango.Database method*), 204
- `get_device_attribute_property_history()` (*tango.Database method*), 204
- `get_device_by_name()` (*tango.Util method*), 182
- `get_device_class()` (*tango.LatestDeviceImpl method*), 143
- `get_device_class()` (*tango.server.Device method*), 122
- `get_device_class_list()` (*tango.Database method*), 205
- `get_device_db()` (*tango.DeviceProxy method*), 44
- `get_device_domain()` (*tango.Database method*), 205
- `get_device_exported()` (*tango.Database method*), 205
- `get_device_exported_for_class()` (*tango.Database method*), 206
- `get_device_family()` (*tango.Database method*), 206
- `get_device_from_alias()` (*tango.Database method*), 206
- `get_device_info()` (*tango.Database method*), 206
- `get_device_ior()` (*tango.Util method*), 182
- `get_device_list()` (*tango.DeviceClass method*), 156
- `get_device_list()` (*tango.Group method*), 89
- `get_device_list()` (*tango.Util method*), 182
- `get_device_list_by_class()` (*tango.Util method*), 182
- `get_device_member()` (*tango.Database method*), 207
- `get_device_name()` (*tango.Database method*), 207
- `get_device_pipe_list()` (*tango.Database method*), 207
- `get_device_pipe_property()` (*tango.Database method*), 207
- `get_device_pipe_property_history()` (*tango.Database method*), 208
- `get_device_properties()` (*tango.LatestDeviceImpl method*), 144
- `get_device_properties()` (*tango.server.Device method*), 122
- `get_device_property()` (*tango.Database method*), 208
- `get_device_property_history()` (*tango.Database method*), 209
- `get_device_property_list()` (*tango.Database method*), 209
- `get_device_proxy()` (*in module tango*), 70
- `get_device_proxy()` (*tango.AttributeProxy method*), 73
- `get_device_service_list()` (*tango.Database method*), 210
- `get_disp_level()` (*tango.Attr method*), 162
- `get_doc_url()` (*tango.DeviceClass method*), 156
- `get_ds_exec_name()` (*tango.Util method*), 183
- `get_ds_inst_name()` (*tango.Util method*), 183
- `get_ds_name()` (*tango.Util method*), 183
- `get_dserver_device()` (*tango.Util method*), 183
- `get_dserver_ior()` (*tango.Util method*), 183
- `get_enum_labels()` (*in module tango.utils*), 231
- `get_err_stack()` (*tango.DeviceAttribute method*), 103
- `get_events()` (*tango.AttributeProxy method*), 73
- `get_events()` (*tango.DeviceProxy method*), 44
- `get_events()` (*tango.utils.EventCallback method*), 231
- `get_exported_flag()` (*tango.LatestDeviceImpl method*), 144
- `get_exported_flag()` (*tango.server.Device method*), 122
- `get_file_name()` (*tango.Database method*), 210
- `get_format()` (*tango.Attr method*), 162
- `get_fqdn()` (*tango.DeviceProxy method*), 45
- `get_from_env_var()` (*tango.DeviceProxy method*), 45
- `get_fully_qualified_name()` (*tango.Group method*), 90
- `get_green_mode()` (*in module tango*), 95
- `get_green_mode()` (*tango.DeviceProxy method*), 45
- `get_home()` (*in module tango.utils*), 235
- `get_host_list()` (*tango.Database method*), 210
- `get_host_name()` (*tango.Util method*), 183
- `get_host_server_list()` (*tango.Database method*), 210
- `get_idl_version()` (*tango.DeviceProxy method*), 46
- `get_info()` (*tango.Database method*), 211
- `get_instance_name_list()` (*tango.Database method*), 211
- `get_label()` (*tango.Attribute method*), 166
- `get_last_event_date()` (*tango.AttributeProxy method*), 74
- `get_last_event_date()` (*tango.DeviceProxy method*), 46
- `get_locker()` (*tango.DeviceProxy method*), 46
- `get_logger()` (*tango.LatestDeviceImpl method*), 144
- `get_logger()` (*tango.server.Device method*), 122
- `get_logging_level()` (*tango.DeviceProxy method*), 46
- `get_logging_target()` (*tango.DeviceProxy method*), 47

`get_max_dim_x()` (*tango.Attribute method*), 167
`get_max_dim_y()` (*tango.Attribute method*), 167
`get_max_value()` (*tango.WAttribute method*), 172
`get_memorized()` (*tango.Attr method*), 162
`get_memorized_init()` (*tango.Attr method*), 162
`get_min_poll_period()` (*tango.LatestDeviceImpl method*), 144
`get_min_poll_period()` (*tango.server.Device method*), 122
`get_min_value()` (*tango.WAttribute method*), 172
`get_name()` (*tango.Attr method*), 162
`get_name()` (*tango.Attribute method*), 167
`get_name()` (*tango.DbHistory method*), 223
`get_name()` (*tango.DeviceClass method*), 157
`get_name()` (*tango.Group method*), 90
`get_name()` (*tango.LatestDeviceImpl method*), 144
`get_name()` (*tango.server.Device method*), 122
`get_non_auto_polled_attr()` (*tango.LatestDeviceImpl method*), 144
`get_non_auto_polled_attr()` (*tango.server.Device method*), 123
`get_non_auto_polled_cmd()` (*tango.LatestDeviceImpl method*), 145
`get_non_auto_polled_cmd()` (*tango.server.Device method*), 123
`get_object_list()` (*tango.Database method*), 211
`get_object_property_list()` (*tango.Database method*), 211
`get_pid()` (*tango.Util method*), 184
`get_pid_str()` (*tango.Util method*), 184
`get_pipe_by_name()` (*tango.DeviceClass method*), 157
`get_pipe_config()` (*tango.DeviceProxy method*), 47
`get_pipe_list()` (*tango.DeviceClass method*), 157
`get_poll_old_factor()` (*tango.LatestDeviceImpl method*), 145
`get_poll_old_factor()` (*tango.server.Device method*), 123
`get_poll_period()` (*tango.AttributeProxy method*), 74
`get_poll_ring_depth()` (*tango.LatestDeviceImpl method*), 145
`get_poll_ring_depth()` (*tango.server.Device method*), 123
`get_polled_attr()` (*tango.LatestDeviceImpl method*), 145
`get_polled_attr()` (*tango.server.Device method*), 123
`get_polled_cmd()` (*tango.LatestDeviceImpl method*), 145
`get_polled_cmd()` (*tango.server.Device method*), 123
`get_polling_period()` (*tango.Attr method*), 162
`get_polling_period()` (*tango.Attribute method*), 167
`get_polling_threads_pool_size()` (*tango.Util method*), 184
`get_prev_state()` (*tango.LatestDeviceImpl method*), 145
`get_prev_state()` (*tango.server.Device method*), 123
`get_properties()` (*tango.Attribute method*), 167
`get_property()` (*tango.AttributeProxy method*), 75
`get_property()` (*tango.Database method*), 212
`get_property()` (*tango.DeviceProxy method*), 48
`get_property_forced()` (*tango.Database method*), 212
`get_property_history()` (*tango.Database method*), 213
`get_property_list()` (*tango.DeviceProxy method*), 48
`get_quality()` (*tango.Attribute method*), 167
`get_serial_model()` (*tango.Util method*), 184
`get_server_access()` (*tango.test_context.DeviceTestContext method*), 289
`get_server_access()` (*tango.test_context.MultiDeviceTestContext method*), 291
`get_server_class_list()` (*tango.Database method*), 213
`get_server_info()` (*tango.Database method*), 213
`get_server_list()` (*tango.Database method*), 214
`get_server_name_list()` (*tango.Database method*), 214
`get_server_version()` (*tango.Util method*), 184
`get_services()` (*tango.Database method*), 214
`get_size()` (*tango.Group method*), 90
`get_source()` (*tango.DeviceProxy method*), 49
`get_state()` (*tango.LatestDeviceImpl method*), 145
`get_state()` (*tango.server.Device method*), 124
`get_status()` (*tango.LatestDeviceImpl method*), 146
`get_status()` (*tango.server.Device method*), 124
`get_sub_dev_diag()` (*tango.Util method*), 184
`get_tango_lib_release()` (*tango.Util method*), 185
`get_tango_lib_version()` (*tango.DeviceProxy method*), 49
`get_timeout_millis()` (*tango.DeviceProxy method*), 49
`get_trace_level()` (*tango.Util method*), 185
`get_transparency_reconnection()` (*tango.AttributeProxy method*), 75
`get_transparency_reconnection()` (*tango.DeviceProxy method*), 49
`get_type()` (*tango.Attr method*), 162
`get_type()` (*tango.DeviceClass method*), 157

[get_type\(\)](#) (*tango.DeviceData method*), 104
[get_user_default_properties\(\)](#)
 (*tango.Attr method*), 163
[get_value\(\)](#) (*tango.DbHistory method*), 223
[get_version_str\(\)](#) (*tango.Util method*), 185
[get_w_attr_by_ind\(\)](#) (*tango.MultiAttribute method*), 175
[get_w_attr_by_name\(\)](#) (*tango.MultiAttribute method*), 175
[get_writable\(\)](#) (*tango.Attr method*), 163
[get_writable\(\)](#) (*tango.Attribute method*), 168
[get_write_value\(\)](#) (*tango.WAttribute method*), 173
[get_write_value_length\(\)](#) (*tango.WAttribute method*), 173
[get_x\(\)](#) (*tango.Attribute method*), 168
[get_y\(\)](#) (*tango.Attribute method*), 168
[GreenMode](#) (*class in tango*), 110
[Group](#) (*class in tango*), 87
[GroupAttrReply](#) (*class in tango*), 94
[GroupCmdReply](#) (*class in tango*), 94
[GroupReply](#) (*class in tango*), 94

H

[history\(\)](#) (*tango.AttributeProxy method*), 75

I

[import_device\(\)](#) (*tango.Database method*), 214
[import_info\(\)](#) (*tango.DeviceProxy method*), 49
[info\(\)](#) (*tango.DeviceProxy method*), 50
[info_stream\(\)](#) (*tango.LatestDeviceImpl method*), 146
[info_stream\(\)](#) (*tango.server.Device method*), 124
[InfoIt](#) (*class in tango*), 160
[init\(\)](#) (*tango.Util method*), 185
[init_device\(\)](#) (*tango.LatestDeviceImpl method*), 146
[init_device\(\)](#) (*tango.server.Device method*), 124
[init_logger\(\)](#) (*tango.LatestDeviceImpl method*), 146
[init_logger\(\)](#) (*tango.server.Device method*), 124
[initialize_dynamic_attributes\(\)](#)
 (*tango.server.Device method*), 124
[insert\(\)](#) (*tango.DeviceData method*), 104
[instance\(\)](#) (*tango.ApiUtil method*), 98
[instance\(\)](#) (*tango.Util method*), 185
[is_allowed\(\)](#) (*tango.Attr method*), 163
[is_archive_event\(\)](#) (*tango.Attr method*), 163
[is_archive_event\(\)](#) (*tango.Attribute method*), 168
[is_array_type\(\)](#) (*in module tango.utils*), 232
[is_assoc\(\)](#) (*tango.Attr method*), 163
[is_attribute_polled\(\)](#) (*tango.DeviceProxy method*), 50
[is_attribute_polled\(\)](#)
 (*tango.LatestDeviceImpl method*), 146
[is_attribute_polled\(\)](#) (*tango.server.Device method*), 124
[is_binary_type\(\)](#) (*in module tango.utils*), 234
[is_bool\(\)](#) (*in module tango.utils*), 232
[is_bool_type\(\)](#) (*in module tango.utils*), 233
[is_change_event\(\)](#) (*tango.Attr method*), 163
[is_change_event\(\)](#) (*tango.Attribute method*), 168
[is_check_archive_criteria\(\)](#) (*tango.Attr method*), 163
[is_check_archive_criteria\(\)](#)
 (*tango.Attribute method*), 168
[is_check_change_criteria\(\)](#) (*tango.Attr method*), 164
[is_check_change_criteria\(\)](#)
 (*tango.Attribute method*), 168
[is_command_polled\(\)](#) (*tango.DeviceProxy method*), 50
[is_command_polled\(\)](#) (*tango.LatestDeviceImpl method*), 146
[is_command_polled\(\)](#) (*tango.server.Device method*), 124
[is_control_access_checked\(\)](#)
 (*tango.Database method*), 215
[is_data_ready_event\(\)](#) (*tango.Attr method*), 164
[is_data_ready_event\(\)](#) (*tango.Attribute method*), 169
[is_dbase_used\(\)](#) (*tango.DeviceProxy method*), 50
[is_deleted\(\)](#) (*tango.DbHistory method*), 223
[is_device_locked\(\)](#) (*tango.LatestDeviceImpl method*), 146
[is_device_locked\(\)](#) (*tango.server.Device method*), 125
[is_device_restarting\(\)](#) (*tango.Util method*), 186
[is_dynamic_interface_frozen\(\)](#)
 (*tango.DeviceProxy method*), 51
[is_empty\(\)](#) (*tango.DbDatum method*), 221
[is_empty\(\)](#) (*tango.DeviceData method*), 104
[is_enabled\(\)](#) (*tango.Group method*), 90
[is_event_queue_empty\(\)](#)
 (*tango.AttributeProxy method*), 76
[is_event_queue_empty\(\)](#) (*tango.DeviceProxy method*), 51
[is_float_type\(\)](#) (*in module tango.utils*), 233
[is_int_type\(\)](#) (*in module tango.utils*), 233
[is_integer\(\)](#) (*in module tango.utils*), 232
[is_locked\(\)](#) (*tango.DeviceProxy method*), 51
[is_locked_by_me\(\)](#) (*tango.DeviceProxy method*), 51
[is_max_alarm\(\)](#) (*tango.Attribute method*), 169
[is_max_value\(\)](#) (*tango.WAttribute method*), 173
[is_max_warning\(\)](#) (*tango.Attribute method*), 169
[is_min_alarm\(\)](#) (*tango.Attribute method*), 169
[is_min_value\(\)](#) (*tango.WAttribute method*), 173
[is_min_warning\(\)](#) (*tango.Attribute method*), 169
[is_multi_tango_host\(\)](#) (*tango.Database method*), 215
[is_non_str_seq\(\)](#) (*in module tango.utils*), 231

`is_number()` (in module *tango.utils*), 232
`is_numerical_type()` (in module *tango.utils*), 233
`is_polled()` (*tango.Attribute* method), 169
`is_polled()` (*tango.AttributeProxy* method), 76
`is_polled()` (*tango.LatestDeviceImpl* method), 147
`is_polled()` (*tango.server.Device* method), 125
`is_pure_str()` (in module *tango.utils*), 231
`is_rds_alarm()` (*tango.Attribute* method), 169
`is_scalar_type()` (in module *tango.utils*), 232
`is_seq()` (in module *tango.utils*), 231
`is_str_type()` (in module *tango.utils*), 234
`is_svr_shutting_down()` (*tango.Util* method), 186
`is_svr_starting()` (*tango.Util* method), 186
`is_there_subscriber()` (*tango.LatestDeviceImpl* method), 147
`is_there_subscriber()` (*tango.server.Device* method), 125
`is_write_associated()` (*tango.Attribute* method), 170
`isoformat()` (*tango.TimeVal* method), 112

K

`KeepAliveCmdCode` (class in *tango*), 109

L

`LatestDeviceImpl` (class in *tango*), 139
`lock()` (*tango.DeviceProxy* method), 51
`LockCmdCode` (class in *tango*), 108
`LockerInfo` (class in *tango*), 102
`LockerLanguage` (class in *tango*), 107
`locking_status()` (*tango.DeviceProxy* method), 52
`LogIt` (class in *tango*), 159
`LogLevel` (class in *tango*), 109
`LogTarget` (class in *tango*), 109

M

`MessBoxType` (class in *tango*), 108
module
 tango, 23
 tango.server, 113
 tango.test_context, 281
`MultiAttribute` (class in *tango*), 174
`MultiDeviceTestContext` (class *tango.test_context*), 289

N

`name()` (*tango.AttributeProxy* method), 77
`name()` (*tango.DeviceProxy* method), 52
`name_equals()` (*tango.Group* method), 90
`name_matches()` (*tango.Group* method), 90
`NamedDevFailedList`, 242
`NonDbDevice`, 240
`NonSupportedFeature`, 241
`NotAllowed`, 242

`now()` (*tango.TimeVal* static method), 112

O

`obj_2_str()` (in module *tango.utils*), 234
`orb_run()` (*tango.Util* method), 186

P

`pending_asynch_call()` (*tango.ApiUtil* method), 98
`pending_asynch_call()` (*tango.DeviceProxy* method), 53
`PeriodicEventInfo` (class in *tango*), 106
`ping()` (*tango.AttributeProxy* method), 77
`ping()` (*tango.DeviceProxy* method), 53
`ping()` (*tango.Group* method), 90
`pipe` (class in *tango.server*), 134
`PipeWriteType` (class in *tango*), 110
`poll()` (*tango.AttributeProxy* method), 77
`poll_attribute()` (*tango.DeviceProxy* method), 53
`poll_attribute()` (*tango.LatestDeviceImpl* method), 147
`poll_attribute()` (*tango.server.Device* method), 125
`poll_command()` (*tango.DeviceProxy* method), 53
`poll_command()` (*tango.LatestDeviceImpl* method), 147
`poll_command()` (*tango.server.Device* method), 125
`PollCmdCode` (class in *tango*), 108
`PollDevice` (class in *tango*), 102
`polling_status()` (*tango.DeviceProxy* method), 53
`PollObjType` (class in *tango*), 108
`print_error_stack()` (*tango.Except* method), 237
`print_exception()` (*tango.Except* method), 237
`push_archive_event()` (*tango.LatestDeviceImpl* method), 148
`push_archive_event()` (*tango.server.Device* method), 126
`push_att_conf_event()` (*tango.LatestDeviceImpl* method), 148
`push_att_conf_event()` (*tango.server.Device* method), 126
`push_change_event()` (*tango.LatestDeviceImpl* method), 148
`push_change_event()` (*tango.server.Device* method), 126
`push_data_ready_event()` (*tango.LatestDeviceImpl* method), 149
`push_data_ready_event()` (*tango.server.Device* method), 127
`push_event()` (*tango.LatestDeviceImpl* method), 149
`push_event()` (*tango.server.Device* method), 127
`push_event()` (*tango.utils.EventCallback* method), 231

- [push_pipe_event\(\)](#) (*tango.LatestDeviceImpl method*), 149
[push_pipe_event\(\)](#) (*tango.server.Device method*), 128
[put_attribute_alias\(\)](#) (*tango.Database method*), 215
[put_class_attribute_property\(\)](#) (*tango.Database method*), 215
[put_class_pipe_property\(\)](#) (*tango.Database method*), 216
[put_class_property\(\)](#) (*tango.Database method*), 216
[put_device_alias\(\)](#) (*tango.Database method*), 217
[put_device_attribute_property\(\)](#) (*tango.Database method*), 217
[put_device_pipe_property\(\)](#) (*tango.Database method*), 217
[put_device_property\(\)](#) (*tango.Database method*), 218
[put_property\(\)](#) (*tango.AttributeProxy method*), 77
[put_property\(\)](#) (*tango.Database method*), 218
[put_property\(\)](#) (*tango.DeviceProxy method*), 54
[put_server_info\(\)](#) (*tango.Database method*), 219
[PYTANGO_GREEN_MODE](#), 13
[Python Enhancement Proposals](#)
[PEP 249](#), 315, 318
- ## R
- [re_throw_exception\(\)](#) (*tango.Except method*), 237
[read\(\)](#) (*tango.AttributeProxy method*), 78
[read_alarm\(\)](#) (*tango.MultiAttribute method*), 176
[read_async\(\)](#) (*tango.AttributeProxy method*), 79
[read_attr_hardware\(\)](#) (*tango.LatestDeviceImpl method*), 150
[read_attr_hardware\(\)](#) (*tango.server.Device method*), 128
[read_attribute\(\)](#) (*tango.DeviceProxy method*), 54
[read_attribute\(\)](#) (*tango.Group method*), 90
[read_attribute_async\(\)](#) (*tango.DeviceProxy method*), 55
[read_attribute_async\(\)](#) (*tango.Group method*), 91
[read_attribute_reply\(\)](#) (*tango.DeviceProxy method*), 55
[read_attribute_reply\(\)](#) (*tango.Group method*), 91
[read_attributes\(\)](#) (*tango.DeviceProxy method*), 55
[read_attributes\(\)](#) (*tango.Group method*), 91
[read_attributes_async\(\)](#) (*tango.DeviceProxy method*), 56
[read_attributes_async\(\)](#) (*tango.Group method*), 91
[read_attributes_reply\(\)](#) (*tango.DeviceProxy method*), 57
[read_attributes_reply\(\)](#) (*tango.Group method*), 92
[read_pipe\(\)](#) (*tango.DeviceProxy method*), 57
[read_reply\(\)](#) (*tango.AttributeProxy method*), 79
[reconnect\(\)](#) (*tango.DeviceProxy method*), 58
[register_service\(\)](#) (*tango.Database method*), 219
[register_signal\(\)](#) (*tango.DeviceClass method*), 157
[register_signal\(\)](#) (*tango.LatestDeviceImpl method*), 150
[register_signal\(\)](#) (*tango.server.Device method*), 128
[Release](#) (*class in tango*), 111
[remove_all\(\)](#) (*tango.Group method*), 92
[remove_attribute\(\)](#) (*tango.LatestDeviceImpl method*), 150
[remove_attribute\(\)](#) (*tango.server.Device method*), 128
[remove_command\(\)](#) (*tango.LatestDeviceImpl method*), 150
[remove_command\(\)](#) (*tango.server.Device method*), 128
[remove_configuration\(\)](#) (*tango.Attribute method*), 170
[remove_logging_target\(\)](#) (*tango.DeviceProxy method*), 58
[rename_server\(\)](#) (*tango.Database method*), 219
[requires_pyrtango\(\)](#) (*in module tango.utils*), 235
[requires_tango\(\)](#) (*in module tango.utils*), 235
[reread_filedatabase\(\)](#) (*tango.Database method*), 220
[reset_filedatabase\(\)](#) (*tango.Util method*), 186
[run\(\)](#) (*in module tango.server*), 137
[run_server\(\)](#) (*tango.server.Device class method*), 129
- ## S
- [scalar_to_array_type\(\)](#) (*in module tango.utils*), 234
[seqStr_2_obj\(\)](#) (*in module tango.utils*), 234
[SerialModel](#) (*class in tango*), 108
[server_cleanup\(\)](#) (*tango.Util method*), 187
[server_init\(\)](#) (*tango.Util method*), 187
[server_init_hook\(\)](#) (*tango.LatestDeviceImpl method*), 150
[server_init_hook\(\)](#) (*tango.server.Device method*), 129
[server_run\(\)](#) (*in module tango.server*), 138
[server_run\(\)](#) (*tango.Util method*), 187
[server_set_event_loop\(\)](#) (*tango.Util method*), 187
[set_abs_change\(\)](#) (*tango.UserDefaultAttrProp method*), 176
[set_access_checked\(\)](#) (*tango.Database method*), 220

<code>set_access_control()</code> (<i>tango.DeviceProxy method</i>), 59	<code>set_default_properties()</code> (<i>tango.Attr method</i>), 165
<code>set_archive_abs_change()</code> (<i>tango.UserDefaultAttrProp method</i>), 176	<code>set_delta_t()</code> (<i>tango.UserDefaultAttrProp method</i>), 177
<code>set_archive_event()</code> (<i>tango.Attr method</i>), 164	<code>set_delta_val()</code> (<i>tango.UserDefaultAttrProp method</i>), 177
<code>set_archive_event()</code> (<i>tango.Attribute method</i>), 170	<code>set_description()</code> (<i>tango.UserDefaultAttrProp method</i>), 177
<code>set_archive_event()</code> (<i>tango.LatestDeviceImpl method</i>), 151	<code>set_disp_level()</code> (<i>tango.Attr method</i>), 165
<code>set_archive_event()</code> (<i>tango.server.Device method</i>), 129	<code>set_display_unit()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_archive_event_abs_change()</code> (<i>tango.UserDefaultAttrProp method</i>), 176	<code>set_enum_labels()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_archive_event_period()</code> (<i>tango.UserDefaultAttrProp method</i>), 177	<code>set_event_abs_change()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_archive_event_rel_change()</code> (<i>tango.UserDefaultAttrProp method</i>), 177	<code>set_event_period()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_archive_period()</code> (<i>tango.UserDefaultAttrProp method</i>), 177	<code>set_event_rel_change()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_archive_rel_change()</code> (<i>tango.UserDefaultAttrProp method</i>), 177	<code>set_format()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_assoc_ind()</code> (<i>tango.Attribute method</i>), 170	<code>set_green_mode()</code> (<i>in module tango</i>), 95
<code>set_asynch_cb_sub_model()</code> (<i>tango.ApiUtil method</i>), 99	<code>set_green_mode()</code> (<i>tango.DeviceProxy method</i>), 60
<code>set_attr_serial_model()</code> (<i>tango.Attribute method</i>), 170	<code>set_label()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_attribute_config()</code> (<i>tango.DeviceProxy method</i>), 59	<code>set_logging_level()</code> (<i>tango.DeviceProxy method</i>), 60
<code>set_attribute_config_3()</code> (<i>tango.LatestDeviceImpl method</i>), 151	<code>set_max_alarm()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_attribute_config_3()</code> (<i>tango.server.Device method</i>), 129	<code>set_max_value()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_change_event()</code> (<i>tango.Attr method</i>), 164	<code>set_max_value()</code> (<i>tango.WAttribute method</i>), 173
<code>set_change_event()</code> (<i>tango.Attribute method</i>), 170	<code>set_max_warning()</code> (<i>tango.UserDefaultAttrProp method</i>), 178
<code>set_change_event()</code> (<i>tango.LatestDeviceImpl method</i>), 151	<code>set_memorized()</code> (<i>tango.Attr method</i>), 165
<code>set_change_event()</code> (<i>tango.server.Device method</i>), 129	<code>set_memorized_init()</code> (<i>tango.Attr method</i>), 165
<code>set_cl_name()</code> (<i>tango.Attr method</i>), 164	<code>set_min_alarm()</code> (<i>tango.UserDefaultAttrProp method</i>), 179
<code>set_class_properties()</code> (<i>tango.Attr method</i>), 165	<code>set_min_value()</code> (<i>tango.UserDefaultAttrProp method</i>), 179
<code>set_config()</code> (<i>tango.AttributeProxy method</i>), 79	<code>set_min_value()</code> (<i>tango.WAttribute method</i>), 173
<code>set_data_ready_event()</code> (<i>tango.Attr method</i>), 165	<code>set_min_warning()</code> (<i>tango.UserDefaultAttrProp method</i>), 179
<code>set_data_ready_event()</code> (<i>tango.Attribute method</i>), 171	<code>set_period()</code> (<i>tango.UserDefaultAttrProp method</i>), 179
<code>set_data_ready_event()</code> (<i>tango.LatestDeviceImpl method</i>), 151	<code>set_pipe_config()</code> (<i>tango.DeviceProxy method</i>), 60
<code>set_data_ready_event()</code> (<i>tango.server.Device method</i>), 130	<code>set_polling_period()</code> (<i>tango.Attr method</i>), 165
<code>set_date()</code> (<i>tango.Attribute method</i>), 171	<code>set_polling_threads_pool_size()</code> (<i>tango.Util method</i>), 188
	<code>set_properties()</code> (<i>tango.Attribute method</i>), 171

- set_quality() (*tango.Attribute* method), 171
 set_rel_change() (*tango.UserDefaultAttrProp* method), 179
 set_serial_model() (*tango.Util* method), 188
 set_server_version() (*tango.Util* method), 188
 set_source() (*tango.DeviceProxy* method), 61
 set_standard_unit() (*tango.UserDefaultAttrProp* method), 179
 set_state() (*tango.LatestDeviceImpl* method), 151
 set_state() (*tango.server.Device* method), 130
 set_status() (*tango.LatestDeviceImpl* method), 152
 set_status() (*tango.server.Device* method), 130
 set_timeout_millis() (*tango.DeviceProxy* method), 61
 set_timeout_millis() (*tango.Group* method), 92
 set_trace_level() (*tango.Util* method), 188
 set_transparency_reconnection() (*tango.AttributeProxy* method), 80
 set_transparency_reconnection() (*tango.DeviceProxy* method), 61
 set_type() (*tango.DeviceClass* method), 158
 set_unit() (*tango.UserDefaultAttrProp* method), 179
 set_value() (*tango.Attribute* method), 171
 set_value_date_quality() (*tango.Attribute* method), 172
 set_w_dim_x() (*tango.DeviceAttribute* method), 103
 set_w_dim_y() (*tango.DeviceAttribute* method), 103
 set_write_value() (*tango.WAttribute* method), 173
 signal_handler() (*tango.DeviceClass* method), 158
 signal_handler() (*tango.LatestDeviceImpl* method), 152
 signal_handler() (*tango.server.Device* method), 130
 size() (*tango.DbDatum* method), 222
 start() (*tango.test_context.DeviceTestContext* method), 289
 start() (*tango.test_context.MultiDeviceTestContext* method), 291
 start_logging() (*tango.LatestDeviceImpl* method), 152
 start_logging() (*tango.server.Device* method), 130
 state() (*tango.AttributeProxy* method), 81
 state() (*tango.DeviceProxy* method), 61
 status() (*tango.AttributeProxy* method), 81
 status() (*tango.DeviceProxy* method), 62
 stop() (*tango.test_context.DeviceTestContext* method), 289
 stop() (*tango.test_context.MultiDeviceTestContext* method), 291
 stop_logging() (*tango.LatestDeviceImpl* method), 152
 stop_logging() (*tango.server.Device* method), 130
 stop_poll() (*tango.AttributeProxy* method), 81
 stop_poll_attribute() (*tango.DeviceProxy* method), 62
 stop_poll_attribute() (*tango.LatestDeviceImpl* method), 152
 stop_poll_attribute() (*tango.server.Device* method), 130
 stop_poll_command() (*tango.DeviceProxy* method), 62
 stop_poll_command() (*tango.LatestDeviceImpl* method), 152
 stop_poll_command() (*tango.server.Device* method), 130
 stop_polling() (*tango.LatestDeviceImpl* method), 152
 stop_polling() (*tango.server.Device* method), 131
 strftime() (*tango.TimeVal* method), 112
 subscribe_event() (*tango.AttributeProxy* method), 82
 subscribe_event() (*tango.DeviceProxy* method), 62
- ## T
- tango module, 23
 tango.server module, 113
 tango.test_context module, 281
 TANGO_HOST, 3, 4, 243
 throw_exception() (*tango.Except* method), 238
 throw_python_exception() (*tango.Except* method), 238
 TimeVal (class in *tango*), 111
 to_dev_failed() (*tango.Except* static method), 239
 todatetime() (*tango.TimeVal* method), 112
 totime() (*tango.TimeVal* method), 113
 trigger_attr_polling() (*tango.Util* method), 189
 trigger_cmd_polling() (*tango.Util* method), 189
- ## U
- unexport_device() (*tango.Database* method), 220
 unexport_event() (*tango.Database* method), 220
 unexport_server() (*tango.Database* method), 221
 unfreeze_dynamic_interface() (*tango.DeviceProxy* method), 64
 unlock() (*tango.DeviceProxy* method), 64
 unregister_server() (*tango.Util* method), 189

`unregister_service()` (*tango.Database method*), 221
`unregister_signal()` (*tango.DeviceClass method*), 158
`unregister_signal()` (*tango.LatestDeviceImpl method*), 152
`unregister_signal()` (*tango.server.Device method*), 131
`unsubscribe_event()` (*tango.AttributeProxy method*), 84
`unsubscribe_event()` (*tango.DeviceProxy method*), 65
`UserDefaultAttrProp` (*class in tango*), 176
`Util` (*class in tango*), 180

W

`warn_stream()` (*tango.LatestDeviceImpl method*), 153
`warn_stream()` (*tango.server.Device method*), 131
`WarnIt` (*class in tango*), 160
`WAttribute` (*class in tango*), 172
`write()` (*tango.AttributeProxy method*), 84
`write_async()` (*tango.AttributeProxy method*), 85
`write_attr_hardware()` (*tango.LatestDeviceImpl method*), 153
`write_attr_hardware()` (*tango.server.Device method*), 131
`write_attribute()` (*tango.DeviceProxy method*), 65
`write_attribute()` (*tango.Group method*), 92
`write_attribute_async()` (*tango.DeviceProxy method*), 66
`write_attribute_async()` (*tango.Group method*), 92
`write_attribute_reply()` (*tango.DeviceProxy method*), 66
`write_attribute_reply()` (*tango.Group method*), 93
`write_attributes()` (*tango.DeviceProxy method*), 66
`write_attributes_async()` (*tango.DeviceProxy method*), 67
`write_attributes_reply()` (*tango.DeviceProxy method*), 68
`write_filedatabase()` (*tango.Database method*), 221
`write_pipe()` (*tango.DeviceProxy method*), 68
`write_read()` (*tango.AttributeProxy method*), 85
`write_read_attribute()` (*tango.DeviceProxy method*), 69
`write_read_attributes()` (*tango.DeviceProxy method*), 69
`write_reply()` (*tango.AttributeProxy method*), 86
`WrongData`, 241
`WrongNameSyntax`, 240