Tango /Python Binding

User's & Reference Guide

Version courante du document : 3.0

Dernière modification le 21/10/2003

Historique des modifications

Date	Revision	Description	Author
18/07/03	1.0	Initial Version	M. Ounsy
6/10/03	2.0	Extension of the "Getting Started " paragraph	A. Buteau/M. Ounsy
14/10/03	3.0	Added Exception Handling paragraph	M. Ounsy

TABLE OF CONTENTS

1 Introduction

The python Tango binding is made accessible using the module : PyTango. The PyTango python module implements the Python Tango Device and Database API mapping. It then allows access from Python environment to the Tango high level C++ classes and structures (see “The TANGO Control system manual” for complete reference) .

These Tango high level C++ classes and structures are exported to Python using the Boost.python library (see http://boost.sourceforge.net). Details on how to install the boost library and generate the python PyTango module from source code will be in given in Appendix at the end of this document.

2 Getting started

2.1 Installation of binding

- To use the Python Binding, two dlls ***PyTango.pyd*** and ***boost_python.dll*** (on Windows) or PyTango.so and ***libboost_python.so*** (on Linux) have to be installed in a directory referenced in the **PYTHONPATH** search path.
- To access Tango devices in your python script, the following line must be in the header of your script file (it’s purpose is to import the Tango binding in the interpreter)

  ```
  from PyTango import *
  ```

- You are now ready to execute your first PyTango script !

2.2 A quick tour of binding through real examples

2.2.1 Example1 : Test the connection to the Device and get it’s current state. See test_ping_state.py

```
from PyTango import *
import sys
import time

#  Protect the script from Exceptions raised by the Tango or python itself
try :
        # Get proxy on the tangotest1 device
        print "Getting DeviceProxy "
        tangotest = DeviceProxy("tango/tangotest/1")

        # First use the classical command_inout way to execute the DevString command
        # (DevString in this case is a command of the TangoTest device)

        result= tangotest.command_inout("DevString", "First hello to device")
        print "Result of execution of DevString command=", result

        # the same with a Device specific command
        result= tangotest.DevString("Second Hello to device")
        print "Result of execution of DevString command=", result
```

```
        # Please note that argin argument type is automagically managed by python
        result= tangotest.DevULong(12456)
        print "Result of execution of Status command=", result

#       Catch Tango and Systems  Exceptions
except :
        print "Failed with exception !"
        print sys.exc_info()[0]
```

2.2.2 Example 2 : Execute commands with scalar arguments on a Device. See test_simple_commands.py

As you can see in the following example, when scalar types are used, the Tango binding automagically manages the data types, and writing scripts is quite easy !!

```
        tangotest = DeviceProxy("tango/tangotest/1")

        # First use the classical command_inout way to execute the DevString command
        # (DevString in this case is a command of the TangoTest device)

        result= tangotest.command_inout("DevString", "First hello to device")
        print "Result of execution of DevString command=", result

        # the same with a Device specific command
        result= tangotest.DevString("Second Hello to device")
        print "Result of execution of DevString command=", result

        # Please note that argin argument type is automagically managed by python
        result= tangotest.DevULong(12456)
        print "Result of execution of Status command=", result
```

2.2.3 Example 3: Execute commands with more complex types.See test_complex_commands.py

In this case you have to use put your arguments data in the correct python structures

```
        print "Getting DeviceProxy "
        tango_test = DeviceProxy("tango/tangotest/1")
        # The input argument is a DevVarLongStringArray
        # so create the argin variable containing
        # an array of longs and an array of strings
        argin = ([1,2,3], ["Hello", "TangoTest device"])

        result= tango_test.DevVarLongArray(argin)
        print "Result of execution of DevVarLongArray command=", result
```

2.2.4 Example 4: Reading and writing attributes : test_read_write_attributes.py

```
        #Read a scalar attribute
        scalar=tangotest.read_attribute("long_scalar")
```

```
print "attribute value", scalar.value
#Read a spectrum attribute
spectrum=tangotest.read_attribute("double_spectrum")
print "attribute value", spectrum.value


# Write a scalar attribute so use the scalar structure
print "Writing attributes"


# Write a scalar attribute so use the scalar structure
scalar.value = 18
print "attribute scalar  ", scalar
print "Writing scalar attributes"
tangotest.write_attribute(scalar)


# Write a scalar attribute so use the scalar structure
spectrum.value = [1.2,3.2,12.3]
print "attribute spectrum ", spectrum
print "Writing spectrum attributes"
tangotest.write_attribute(spectrum)
```

2.2.5 Example 5: Defining devices in the Tango DataBase.See test_device_creation.py

```
#  A reference on the DataBase
db = Database()

# The 3 devices name we want to create
# Note : these 3 devices will be served by the same DServer
new_device_name1="px1/tdl/mouse1"
new_device_name2="px1/tdl/mouse2"
new_device_name3="px1/tdl/mouse3"

# Define the Tango Class served by this  DServer
new_device_info_mouse = DbDevInfo()
new_device_info_mouse._class = "Mouse"
new_device_info_mouse.server = "ds_Mouse/server_mouse"

# add the first device
print "Creation Device :" , new_device_name1
new_device_info_mouse.name = new_device_name1
db.add_device(new_device_info_mouse)

# add the next device
print "Creation Device :" , new_device_name2
new_device_info_mouse.name = new_device_name2
db.add_device(new_device_info_mouse)
# add the third device
print "Creation Device :" , new_device_name3
new_device_info_mouse.name = new_device_name3
db.add_device(new_device_info_mouse)
```

2.2.6 Example 6: Setting up Device properties. See test_device_properties.py

2.2.7 Example 7: A more complex example using python subtilities

The following python script example (containing some functions and instructions manipulating a Galil motor axis device server) gives an idea of how the Tango API should be accessed from Python

```
# connecting to the motor axis device
axis1 = DeviceProxy ("microxas/motorisation/galilbox")
# Getting Device Properties
property_names = ["AxisBoxAttachement",
                  "AxisEncoderType",
                  "AxisNumber",
                  "CurrentAcceleration",
                  "CurrentAccuracy",
                  "CurrentBacklash",
                  "CurrentDeceleration",
                  "CurrentDirection",
                  "CurrentMotionAccuracy",
                  "CurrentOvershoot",
                  "CurrentRetry",
                  "CurrentScale",
                  "CurrentSpeed",
                  "CurrentVelocity",
                  "EncoderMotorRatio",
                  "logging_level",
                  "logging_target",
                  "UserEncoderRatio",
                  "UserOffset"]
axis_properties = axis1.get_property(property_names)
for prop in axis_properties.keys():
print "%s : %s" % (prop,axis_properties[prop][0])
# Changing Properties
axis_properties["AxisBoxAttachement"] = ["microxas/motorisation/galilbox"]
axis_properties["AxisEncoderType"] = ["1"]
axis_properties["AxisNumber"] = ["6"]
axis1.put_property(axis_properties)
# Reading attributes and storing them in a python dictionary
att_dict = {}
att_list = axis.get_attribute_list()
for att in att_list:
   att_val = axis.read_attribute(att)
    print "%s : %s" % (att,att_val.value)
   att_dict[att] = att_val
# Changing some  attribute values
attributes["AxisBackslash"].value = 0.5
axis1.write_attribute(attributes["AxisBackslash"])
attributes["AxisDirection"].value = 1.0
axis1.write_attribute(attributes["AxisDirection"])
attributes["AxisVelocity"].value = 1000.0
axis1.write_attribute(attributes["AxisVelocity"])
attributes["AxisOvershoot"].value = 500.0
```

```
axis1.write_attribute(attributes["AxisOvershoot"])
# Testing some device commands
pos1=axis1.read_attribute("AxisCurrentPosition")
axis1.command_inout("AxisBackward")
while pos1.value > 1000.0:
    pos1=axis1.read_attribute("AxisCurrentPosition")
    print "position axis 1 = ",pos1.value
axis1.command_inout("AxisStop")
```

3 The Tango Device Python API

The PyTango allows access from Python environment to the following Tango high level C++ Device classes and structures :

- DeviceInfo
- DbDevImportInfo
- CommandInfo
- TimeVal
- DeviceDataHistory
- AttributeInfo
- AttributeValue
- DeviceAttributeHistory
- DbDevInfo
- DbDevExportInfo
- Database

Additionally, Tango enumerated types are mapped to named python constants as follows,

Tango DevState enumeration values are mapped to the following integer constants :

- DevState.ON
- DevState.OFF
- DevState.CLOSE
- DevState.OPEN
- DevState.INSERT
- DevState.EXTRACT
- DevState.MOVING
- DevState.STANDBY
- DevState.FAULT
- DevState.RUNNING
- DevState.ALARM

- DevState.DISABLE
- DevState.UNKNOWN

Tango DevSource enumeration values are mapped to the following integer constants :

- DevSource.DEV
- DevSource.CACHE
- DevSource.CACHE_DEV

Tango DispLevel enumeration values are mapped to the following integer constants :

- DispLevel.OPERATOR
- DevSource.EXPERT

Tango AttrWriteType enumeration values are mapped to the following integer constants :

- AttrWriteType.READ
- AttrWriteType.READ_WITH_WRITE
- AttrWriteType.WRITE
- AttrWriteType.READ_WRITE

Tango AttrDataFormat enumeration values are mapped to the following integer constants :

- AttrDataFormat.SCALAR
- AttrDataFormat.SPECTRUM
- AttrDataFormat.IMAGE

Tango AttrQuality enumeration values are mapped to the following integer constants :

- AttrQuality.ATTR_VALID
- AttrQuality.ATTR_INVALID
- AttrQuality.ATTR_ALARM

3.1 DeviceInfo

DeviceInfo is a Python object containing available information for a device in the following field members

- dev_class : string
- server_id : string
- server_host : string
- server_version : integer

- doc_url : string

3.2 DbDevImportInfo

DbDevImportInfo is a Python object containing information that can be imported from the configuration database for a device in the following field members

- name : device name
- exported : 1 if device is running, 0 else
- ior : CORBA reference of the device
- version : string

3.3 CommandInfo

A device command info with the following members

- cmd_name : command name as ascci string
- cmd_tag : command as binary value (for TACO)
- in_type : input type as binary value (integer)
- out_type : output type as binary value (integer)
- in_type_desc : description of input type (optional)
- out_type_desc : description of output type (optional)
- disp_level : command display level (DispLevel type)

3.4 DevError

Python structure discribing any error resulting from a command execution or an attribute query, with following members

- reason : string
- severity : one of ErrSeverity.WARN, ERRSeverity.ERR
  or ErrSeverity.PANIC constants
- desc : error description (string)
- out_type : output type as binary value (integer)
- origin : Tango server method in which the error happened

3.5 TimeVal

Time value structure with three field members

- tv_sec : seconds
- tv_usec : microseconds
- ty_nsec : nanoseconds

3.6 DeviceDataHistory

A python object obtained as a result of query of a command history with field members

- time : time of command execution (see TimeVal type)
- cmd_failed : true if attribute command execution failed
- value : returned value as a python object,
  valid if cmd_failed is false,
- errors : list of errors that occurred (see DevError type)
  empty if cmd_failed is false

3.7 AttributeInfo

A structure containing available information for an attribute with

the following members

- name : attribute name
- writable : one of AttrWriteType constant values
  AttrWriteType.READ, AttrWriteType.READ_WITH_WRITE
  AttrWriteType.WRITE or AttrWriteType.READ_WRITE
- data_format : one of AttrDataFormat constant values
  AttrWriteType.SCALAR, AttrWriteType.SPECTRUM
  or AttrWriteType.IMAGE
- data_type : integer value indicating attribute type (float, string,..)
- max_dim_x : first dimension of attribute (spectrum or image attributes)
- max_dim_y : second dimension of attribute(image attribute)
- description : string describing the attribute
- label : attribute label (Voltage, time, ...)
- unit : attribute unit (V, ms, ...)
- standard_unit : string
- display_unit : string
- format : string
- min_value : string
- max_value : string
- min_alarm : string
- max_alarm : string
- writable_attr_name : string
- extensions : list of strings

- disp_level : one of DispLevel constants

DispLevel.OPERATOR or DispLevel.EXPERT

3.8 AttributeValue

A structure encapsulating the attribute value with additional information in the following members

- value : python object with effectuve value
- quality : one of AttrQuality constant values

AttrQuality.VALID, AttrQuality.INVALID

or AttrQuality.ALARM\n"

- time : time of value read (see TimeVal type)
- name : attribute name
- dim_x : effective first dimension of attribute (spectrum or image attributes)
- dim_y : effective second dimension of attribute(image attribute)

3.9 DeviceAttributeHistory

A python object obtained as a result of an attribute read history query with field members

- attr_failed : true if attribute read operation failed
- value : attribute value as an AttributeValue type

valid if attr_failed is false

- errors : list of errors that occurred (see DevError type)

empty if attr_failed is false

- name : attribute name
- dim_x : effective first dimension of attribute (spectrum or image attributes)
- dim_y : effective second dimension of attribute(image attribute)

3.10 DeviceProxy

DeviceProxy is the high level Tango object which provides the client with an easy-to-use interface to TANGO devices. DeviceProxy provides interfaces to all TANGO Device interfaces. The DeviceProxy manages timouts, 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 = DeviceProxy("tango/tangotest/1")
```

3.10.1 state()

A method which returns the state of the device.

Parameters : None

Return : DevState constant

```
Example :
    dev = DeviceProxy("tango/tangotest/1")
    if dev.state() == DevState.ON : ...
```

3.10.2 status()

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

Parameters : None

Return : string

3.10.3 ping()

A method which sends a ping to the device

Parameters : None

Return : time elapsed in milliseconds

3.10.4 set_timeout_millis()

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)
```

3.10.5 get_timeout_millis()

```
Get the client side timeout in milliseconds.
Parameters : None
```

3.10.6 get_idl_version()

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

3.10.7 set_source()

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)
```

3.10.8 get_source()

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

Parameters : None

Return : DevSource constant

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

3.10.9 black_box()

Get the last commands executed on the device server.

Parameters :

- n : n number of commands to get\n"

Return : list of strings containing the date, time, command and from which client computer the command was executed.

```
Example :
    print black_box(4)
```

3.10.10 name()

Return the device name from the device itself.

3.10.11 adm_name()

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.

3.10.12 dev_name()

Return the device name as it is stored locally.

3.10.13 info()

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.

3.10.14 import_info()

Query the device for import info from the database.

Parameters : None

Return : DbDevImportInfo object

```
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.

3.10.15 description()

Get device description.

Parameters : None

Return : string describing the device

3.10.16 command_query()

Query the device for information about a single command.

Parameters :

- command : command name

Return : CommandInfo object

```
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
```

3.10.17 command_list_query()

Query the device for information on all commands.

Parameters : None

Return : list of CommandInfo objects

3.10.18 command_inout()

Execute a command, on a device, which takes zero or one input argument.

Parameters :

- name : command name
- argin : if needed, python object containing the input argument

Return : result of command execution as a python object

Example :

```
str_res = dev.command_inout("DevString","Hello!")
print str_res
```

3.10.19 command_history()

Retrive command history from the command polling buffer.

Parameters :

- name : command name
- depth : integer representing the wanted history depth

Return : a list of DeviceDataHistory objects.

Example :

```
com_hist = dev.command_history("DevString",3)
for dev_hist in com_hist : print dev_hist
```

See DeviceDataHistory documentation form more detail.

3.10.20 attribute_query()

Query the device for information about a single attribute.

Parameters :

- attribute : attribute name

Return : AttributeInfo object

```
Example :
    attr_info = dev.attribute_query("short_scalar")
    print attr_info.name
    print attr_info.writable
    print attr_info.data_format
    print attr_info.data_type
    print attr_info.max_dim_x
    print attr_info.max_dim_y
    print attr_info.description
    print attr_info.label
    print attr_info.unit
    print attr_info.standard_unit
    print attr_info.display_unit
    print attr_info.format
    print attr_info.min_value
    print attr_info.max_value
    print attr_info.min_alarm
    print attr_info.max_alarm
    print attr_info.writable_attr_name
    print attr_info.extensions
    print attr_info.disp_level
```

See AttributeInfo documentation form more detail.

3.10.21 attribute_list_query()

Query the device for information on a list of attributes.

Parameters : None

Return : list of AttributeInfo objects.

3.10.22 get_attribute_list()

Return the names of all attributes implemented for this device.

Parameters : None

Return : list of strings

3.10.23 get_attribute_config()

Same effect as calling attribute_query() or attribute_list_query(), depending on the argument passed to the method : a string name or a list of names.

3.10.24 set_attribute_config()

Change the attribute configuration for the specified attributes.

Parameters : list of AttributeInfo types

Return : None

3.10.25 read_attribute()

Read a single attribute.

Parameters :

- name : attribute name

Return : AttributeValue object

```
Example :
    attr_val = dev.read_attribute("short_scalar")
    print attr_val.value
    print attr_val.time
    print attr_val.quality
    print attr_val.name
    print attr_val.dim_x
    print attr_val.dim_y
```

See AttributeValue documentation string form more detail.

3.10.26 read_attributes()

Read the list of specified attributes.

Parameters :

- name : list of attribute names

Return : list of AttributeValue types

3.10.27 write_attribute()

Write the specified attribute.

Parameters :

- attr_val : AttributeValue type

Return : None

```
Example :
    attr_val = dev.read_attribute("short_scalar")
    attr_val.value = 5
    dev.write_attribute(attr_val)
```

3.10.28 write_attributes()

Write the specified list of attributes.

Parameters :

- attr_list : list of AttributeValue objects

```
Return : None
```

3.10.29 attribute_history()

Retrive attribute history from the command polling buffer.

Parameters :

- name : attribute name
- depth : integer representing the wanted history depth

Return : a list of DeviceAttributeHistory types

Example :

```
for dev_hist in dev.attribute_history("short_scalar",3): print dev_hist
```

See DeviceAttributeHistory documentation string form more detail.

3.10.30 is_command_polled()

True if the command is polled.

Parameters :

- cmd_name : command name

Return : boolean value

3.10.31 is_attribute_polled()

True if the attribute is polled.

Parameters :

- attr_name : command name

Return : boolean value

3.10.32 get_command_poll_period()

Return the command polling period.

Parameters :

- cmd_name : command name

Return : polling period in milliseconds

3.10.33 get_attribute_poll_period()

Return the attribute polling period.

Parameters :

- attr_name : attribute name

Return : polling period in milliseconds

3.10.34 polling_status()

Return the device polling status.

Parameters :None

Return : list of strings, with one string for each
polled command/attribute

Each string is a 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

3.10.35 poll_command()

Add a command to the list of polled commands.

Parameters :

- cmd_name : command name
- period : polling period in milliseconds

Return : None

3.10.36 poll_attribute()

Add an attribute to the list of polled attributes.

Parameters :

- attr_name : attribute name
- period : polling period in milliseconds

Return : None

3.10.37 stop_poll_command()

Remove a command from the list of polled commands.

Parameters :

- cmd_name : command name

Return : None

3.10.38 stop_poll_attribute()

Remove an attribute from the list of polled attributes.

Parameters :

- attr_name : attribute name

Return : None

3.10.39 get_property()

Get a list of properties for a device.

Parameters :

- prop_list : list of property names

Return : a dictionary which keys are the property names the value associated with each key being a

list of strings representing the property value

3.10.40 put_property()

Put a list of properties for a device.

Parameters :

- props : a dictionary which keys are the property names the value associated with each key

being a list of strings representing the property value

Return : None

3.10.41 delete_property()

Delete a list of properties for a device.

Parameters :

- prop_list : list of property names

Return : None

4 The Tango Database Python API

The PyTango allows access from Python environment to the following Tango high level C++ Database classes and structures :

- DbDevInfo
- DbDevImportInfo
- DbDevExportInfo
- Database

4.1 DbDevInfo

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

- name : string
- class : string
- server : string

4.2 DbDevExportInfo

Export info for a device with the following members,

- name : device name
- ior : CORBA reference of the device
- host : name of the computer hosting the server
- version : string
- pid : process identifier

4.3 Database

Database is the high level Tango object which contains the link to the static database. Database provides methods for all database commands : get_device_property(), info(), etc.

To create a Database, use the default constructor :

```
db = Database()
```

The constructor uses the TANGO_HOST environment variable to determine which instance of the Database to connect to.

4.3.1 get_info()

Query the database for some general info about the tables.

Parameters : None

Return : a multi-line string

4.3.2 add_device()

Add a device to the database. The device name, server and class are specified in the DbDevInfo structure.

Parameters : DbDevInfo structure

Return : None

```
Example :
    dev_info = DbDevInfo()
    dev_info.name = "my/own/device"
    dev_info.class = "MyDevice"
    dev_info.server = "MyServer/test"
    db.add_device(dev_info)
```

4.3.3 delete_device()

```
Delete the device of the specified name from the database
Parameters : Device name
Return     : None
```

```
Example :
    db.delete_device("my/own/device")
```

4.3.4 import_device()

Query the databse for the import info of the specified device.

Parameters : Device name

Return : DbDevImportInfo object

```
Example :
    dev_imp_info = db.import_device("my/own/device")
    print dev_imp_info.name
    print dev_imp_info.exported
    print dev_imp_info.ior
    print dev_imp_info.version
```

4.3.5 export_device()

Update the export info for this device in the database.

Parameters : DbDevExportInfo structure

Return : None

```
Example :
    dev_export = DbDevExportInfo()
    dev_export.name = "my/own/device"
    dev_export.ior = "the real ior"
    dev_export.host = "the host"
    dev_export.version = "1.0"
    dev_export.pid = "...."
    db.export_device(dev_export)
```

4.3.6 unexport_device()

Mark the specified device as un-exported in the database.

Parameters : Device name

Return : None

```
Example :
    db.unexport_device("my/own/device")
```

4.3.7 add_server()

Add a group of devices to the database.

Parameters : - Server name

- List of DbDevInfo structures

Return : None

4.3.8 delete_server()

Delete the device server and its associated devices from database.

Parameters : Server name

Return : None

4.3.9 export_server()

Export a group of devices to the database.

Parameters : - Server name

- List of DbDevExportInfo structures

Return : None

4.3.10 unexport_server()

Mark all devices exported for this server as un-exported.

Parameters : Server name

Return : None

4.3.11 get_device_name()

Query the database for a list of devices served by a server for a given device class.

Parameters : - Server name

- Device class name

Return : List of device names

4.3.12 get_device_alias()

Query the database for a list of aliases for the specified device.

Parameters : Device name

Return : List of aliases

4.3.13 get_device_exported()

Query the database for a list of exported devices whose names satisfy the supplied filter (* is wildcard for any character(s)).

Parameters : string filter

Return : List of exported devices

4.3.14 get_device_domain()

Query the database for a list of of device domain names which match the wildcard provided (* is wildcard for any character(s)).

Domain names are case insensitive.

Parameters : string filter

Return : List of device domain names

4.3.15 get_device_family()

Query the database for a list of of device family names which match the wildcard provided (* is wildcard for any character(s)).

Family names are case insensitive.

Parameters : string filter

Return : List of device family names

4.3.16 get_device_member()

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 : string filter

Return : List of device member names

4.3.17 get_property()

Query the database for a list of object (i.e non-device) properties.

Parameters :

- string : object name
- prop_list : list of property names

Return : a dictionary which keys are the property names
the value associated with each key being a list
of strings representing the property value

4.3.18 put_property()

Insert or update a list of properties for the specified object.

Parameters :

- string : object name
- props : a dictionary which keys are the property names

the value associated with each key being a list
of strings representing the property value

Return : None

4.3.19 delete_property()

Delete a list of properties for the specified object.

Parameters :

- string : object name
- prop_list : list of property names

Return : None

4.3.20 get_device_property()

Query the database for a list of device properties.

Parameters :

- string : device name
- prop_list : list of property names

Return : a dictionary which keys are the property names
the value associated with each key being a list
of strings representing the property value

4.3.21 put_device_property()

Insert or update a list of properties for the specified device.

Parameters :

- string : device name
- props : a dictionary which keys are the property names
the value associated with each key being a list
of strings representing the property value

Return : None

```
Example :
properties_logging= {'logging_level':["DEBUG"],
                     'logging_target':["device::tmp/log/1"]}
db.put_device_property("px1/tdl/mouse",properties_logging)
```

4.3.22 delete_device_property()

Delete a list of properties for the specified device.

Parameters :

- string : device name
- prop_list : list of property names

Return : None

4.3.23 get_device_attribute_property()

Query the database for a list of device attribute properties.

Parameters :

- string : device name

- prop_list : list of attribute names

Return : a dictionary which keys are the attribute names, the value associated with each key being

another dictionary which keys are the attribute property names.

4.3.24 put_device_attribute_property()

Insert or update a list of properties for the specified device attribute.

Parameters :

- string : device name

- props : a dictionary which keys are the attribute names, the value associated with each key being

another dictionary which keys are the attribute property names.

Return : None

```
Example :
properties_positionX={"min_value" : ["20"],
                      "max_value" : ["1000"],
                      "min_alarm":["50"],
                     "max_alarm":["950"],
                     "label":["Pos en pixel"],
                     "format":["%3d"]}

properties_positionY={"min_value" : ["20"],
                      "max_value" : ["1000"],
                      "min_alarm":["50"],
                     "max_alarm":["950"],
                     "label":["Pos en pixel"],
                     "format":["%3d"]}

attr_props_mouse = { "positionX" : properties_positionX ,
                     "positionY" : properties_positionY }

db.put_device_attribute_property(mouse_name,attr_props_mouse)
```

4.3.25 delete_device_attribute_property()

Delete a list of properties for the specified device attribute.

Parameters :

- string : device name

- prop_list : list of attribute names

Return : None

4.3.26 get_class_property()

Query the database for a list of device class properties.

Parameters :

- string : class name

- prop_list : list of property names

Return : a dictionary which keys are the property names the value associated with each key being a

list of strings representing the property value

4.3.27 put_class_property()

Insert or update a list of properties for the specified device class.

Parameters :

- string : class name

- props : a dictionary which keys are the property names the value associated with each key being

a list of strings representing the property value

Return : None

4.3.28 delete_class_property()

Delete a list of properties for the specified device class.

Parameters :

- string : class name

- prop_list : list of property names

Return : None

4.3.29 get_class_attribute_property()

Query the database for a list of device class attribute properties.

Parameters :

- string : class name

- prop_list : list of property names

Return : a dictionary which keys are the attribute names, the value associated with each key being

another dictionary which keys are the attribute property names.

4.3.30 put_class_attribute_property()

Insert or update a list of properties for the specified class attribute.

Parameters :

- string : class attribute name

- props : a dictionary which keys are the attribute names, the value associated with each key being

  another dictionary which keys are the attribute property names.

Return : None

4.3.31 delete_class_attribute_property()

Delete a list of properties for the specified class attribute.

Parameters :

- string : class name

- prop_list : list of attribute names

Return : None

5 Exception Handling

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.

For a detailed meaning and description of the context in which they are thrown, please refer to the Tango control system documentation.

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.

Exemple

```
#  Protect the script from Exceptions raised by the Tango or python itself
try :
        # Get proxy on the tangotest1 device
        print "Getting DeviceProxy "
```

```
            tangotest = DeviceProxy("tango/tangotest/1")
#           Catch Tango and Systems  Exceptions
except DevFailed :
            exctype , value = sys.exc_info()[:2]
            print "Failed with exception ! “ , exctype
            for err in value :
                        print “ reason” , err[“reason”]
                        print “ description” , err[“desc”]
                        print “origin” , err[“origin”]
                        print “severity” , err[“severity”]
```