Please ask about problems and questions regarding this tutorial on answers.ros.org (http://answers.ros.org). Don't forget to include in your question the link to this page, the versions of your OS & ROS, and also add appropriate tags.

Using a C++ class in Python

Description: This tutorial illustrates a way to use a C++ class with ROS messages in Python.

Keywords: C++, Python, bindings

Tutorial Level: ADVANCED

Next Tutorial: Packaging your ROS project as a snap (/ROS/Tutorials/Packaging%20your%20ROS%20project%20as%20a%20snap)

catkin | rosbuild

Sommaire

This tutorial illustrates a way to use a C++ class with ROS messages in Python. The Boost Python library is used. The difficulty is to translate Python objects of ROS messages written in pure Python into equivalent C++ instances. This translation will be done through serialization/deserialization. The source files can be found at https://github.com/galou/python_bindings_tutorial (https://github.com/galou/python_bindings_tutorial).

Another solution is to use classical ROS services, the server written in C++ will be a wrapper around the C++ class and the client, C++ or Python, will call the service. The solution proposed here does not create a ROS node, provided the class to be wrapped does not make use of `ros::NodeHandle`.

Another solution is to write a wrapper for all needed ROS messages and their dependencies. Some apparently deprecated package proposed some solutions for the automation of this task: genpybindings (https://github.com/mkjaergaard/genpybindings.git) and boost_ python_ros (https://github.com/bhaskara/boost_python_ros.git).

1. Class without NodeHandle

Because roscpp is not initialized when calling `rospy.init_node`. `ros::NodeHandle` instances cannot be used in the C++ class without generating an error. If the C++ does not make use of `ros::NodeHandle`, this is no issue though.

1.1 Creating the package and writing the C++ class

Create a package and create the C++ class for which we will want to make a Python binding. This class uses ROS messages as arguments and return type.

Afficher/masquer les numéros de lignes

```
catkin_create_pkg python_bindings_tutorial rospy roscpp std_msgs
cd python_bindings_tutorial/include/python_bindings_tutorial
touch add_two_ints.h
rosed python_bindings_tutorial add_two_ints.h
```

The content of `include/python_bindings_tutorial/add_two_ints.h` will be:

Afficher/masquer les numéros de lignes

```
#ifndef PYTHON_BINDINGS_TUTORIAL_ADD_TWO_INTS_H
#define PYTHON_BINDINGS_TUTORIAL_ADD_TWO_INTS_H

#include <std_msgs/Int64.h>

namespace python_bindings_tutorial {

class AddTwoInts
{
  public:
    std_msgs::Int64 add(const std_msgs::Int64& a, const std_msgs::Int64& b);
};

} // namespace python_bindings_tutorial

#endif // PYTHON_BINDINGS_TUTORIAL_ADD_TWO_INTS_H

```

Write the class implementation into .

```
roscd python_bindings_tutorial/src
touch add_two_ints.cpp
rosed python_bindings_tutorial add_two_ints.cpp
```

The content of `src/add_two_ints.cpp` will be:

Afficher/masquer les numéros de lignes

```
#include <python_bindings_tutorial/add_two_ints.h>

using namespace python_bindings_tutorial;

std_msgs::Int64 AddTwoInts::add(const std_msgs::Int64& a, const std_msgs::Int64& b)
{
  std_msgs::Int64 sum;
  sum.data = a.data + b.data;
  return sum;
}
```

1.2 Binding, C++ part

The binding occurs through two wrapper classes, one in C++ and one in Python. The C++ wrapper translates input from serialized content to C++ message instances and output from C++ message instances into serialized content.

Afficher/masquer les numéros de lignes

```
roscd python_bindings_tutorial/src
touch add_two_ints_wrapper.cpp
rosed python_bindings_tutorial add_two_ints_wrapper.cpp
```

The content of `src/add_two_ints_wrapper.cpp` will be:

Afficher/masquer les numéros de lignes

```
#include <boost/python.hpp>

#include <string>

#include <ros/serialization.h>
#include <std_msgs/Int64.h>

#include <python_bindings_tutorial/add_two_ints.h>

/* Read a ROS message from a serialized string.
  */
template <typename M>
M from_python(const std::string str_msg)
{
  size_t serial_size = str_msg.size();
  boost::shared_array<uint8_t> buffer(new uint8_t[serial_size]);
  for (size_t i = 0; i < serial_size; ++i)
  {
    buffer[i] = str_msg[i];
  }
  ros::serialization::IStream stream(buffer.get(), serial_size);
  M msg;
  ros::serialization::Serializer<M>::read(stream, msg);
  return msg;
}

/* Write a ROS message into a serialized string.
*/
template <typename M>
std::string to_python(const M& msg)
{
  size_t serial_size = ros::serialization::serializationLength(msg);
  boost::shared_array<uint8_t> buffer(new uint8_t[serial_size]);
  ros::serialization::OStream stream(buffer.get(), serial_size);
  ros::serialization::serialize(stream, msg);
  std::string str_msg;
  str_msg.reserve(serial_size);
  for (size_t i = 0; i < serial_size; ++i)
  {
    str_msg.push_back(buffer[i]);
  }
  return str_msg;
}

class AddTwoIntsWrapper : public python_bindings_tutorial::AddTwoInts
{
  public:
    AddTwoIntsWrapper() : AddTwoInts() {}

    std::string add(const std::string& str_a, const std::string& str_b)
    {
      std_msgs::Int64 a = from_python<std_msgs::Int64>(str_a);
      std_msgs::Int64 b = from_python<std_msgs::Int64>(str_b);
      std_msgs::Int64 sum = AddTwoInts::add(a, b);

      return to_python(sum);
    }
```

```
  58 };
  59
  60 BOOST_PYTHON_MODULE(_add_two_ints_wrapper_cpp)
  61 {
  62   boost::python::class_<AddTwoIntsWrapper>("AddTwoIntsWrapper", boost::python::init
<>())
  63     .def("add", &AddTwoIntsWrapper::add)
  64     ;
  65 }
```

The line Error: No code_block found creates a Python module in the form of a dynamic library. The name of the module will have to be the name of the exported library in `CMakeLists.txt`.

1.3 Binding, Python part

The Python wrapper translates input from Python message instances into serialized content and output from serialized content to Python message instances. The translation from Python serialized content (`str`) into C++ serialized content (`std::string`) is built in the Boost Python library.

```
roscd python_bindings_tutorial/src
mkdir python_bindings_tutorial
roscd python_bindings_tutorial/src/python_bindings_tutorial
touch _add_two_ints_wrapper_py.py
rosed python_bindings_tutorial _add_two_ints_wrapper_py.py
```

The content of `src/python_bindings_tutorial/_add_two_ints_wrapper_py.py` will be

Afficher/masquer les numéros de lignes

```
 1 from StringIO import StringIO
 2
 3 import rospy
 4 from std_msgs.msg import Int64
 5
 6 from python_bindings_tutorial._add_two_ints_wrapper_cpp import AddTwoIntsWrapper
 7
 8
 9 class AddTwoInts(object):
10     def __init__(self):
11         self._add_two_ints = AddTwoIntsWrapper()
12
13     def _to_cpp(self, msg):
14         """Return a serialized string from a ROS message
15
16         Parameters
17         ----------
18         - msg: a ROS message instance.
19         """
20         buf = StringIO()
21         msg.serialize(buf)
22         return buf.getvalue()
23
24     def _from_cpp(self, str_msg, cls):
25         """Return a ROS message from a serialized string
26
27         Parameters
28         ----------
29         - str_msg: str, serialized message
30         - cls: ROS message class, e.g. sensor_msgs.msg.LaserScan.
31         """
32         msg = cls()
33         return msg.deserialize(str_msg)
34
35     def add(self, a, b):
36         """Add two std_mgs/Int64 messages
37
38         Return a std_msgs/Int64 instance.
39
40         Parameters
41         ----------
42         - a: a std_msgs/Int64 instance.
43         - b: a std_msgs/Int64 instance.
44         """
45         if not isinstance(a, Int64):
46             rospy.ROSException('Argument 1 is not a std_msgs/Int64')
47         if not isinstance(b, Int64):
48             rospy.ROSException('Argument 2 is not a std_msgs/Int64')
49         str_a = self._to_cpp(a)
50         str_b = self._to_cpp(b)
51         str_sum = self._add_two_ints.add(str_a, str_b)
52         return self._from_cpp(str_sum, Int64)
```

In order to be able to import the class as `python_bindings_tutorial.AddTwoInts`, we import the symbols in `__init__.py`. First, we create the file:

Afficher/masquer les numéros de lignes

```
1 roscd python_bindings_tutorial/src/python_bindings_tutorial
2 touch __init__.py
3 rosed python_bindings_tutorial __init__.py
```

The content of `src/python_bindings_tutorial/__init__.py` will be:

Afficher/masquer les numéros de lignes

```
1 from python_bindings_tutorial._add_two_ints_wrapper_py import AddTwoInts
```

1.4 Glueing everything together

Edit the `CMakeLists.txt` (`rosed python_bindings_tutorial CmakeLists.txt`) like this:

```
cmake_minimum_required(VERSION 2.8.3)
project(python_bindings_tutorial)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  roscpp_serialization
  std_msgs
)

## Both Boost.python and Python libs are required.
find_package(Boost REQUIRED COMPONENTS python)
find_package(PythonLibs 2.7 REQUIRED)


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
catkin_python_setup()

###################################
## catkin specific configuration ##
###################################
catkin_package(
        INCLUDE_DIRS include
        LIBRARIES add_two_ints _add_two_ints_wrapper_cpp
        CATKIN_DEPENDS roscpp
        #  DEPENDS system_lib
)

###########
## Build ##
###########

# include Boost and Python.
include_directories(
        include
        ${catkin_INCLUDE_DIRS}
        ${Boost_INCLUDE_DIRS}
        ${PYTHON_INCLUDE_DIRS}
        )

## Declare a cpp library
add_library(add_two_ints src/add_two_ints.cpp)
add_library(_add_two_ints_wrapper_cpp src/add_two_ints_wrapper.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(add_two_ints ${catkin_LIBRARIES})
target_link_libraries(_add_two_ints_wrapper_cpp add_two_ints ${catkin_LIBRARIES} ${Boost_LIBR
ARIES})

# Don't prepend wrapper library name with lib and add to Python libs.
set_target_properties(_add_two_ints_wrapper_cpp PROPERTIES
        PREFIX ""
        LIBRARY_OUTPUT_DIRECTORY ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_PYTHON_DESTINATION}
        )
```

The c++ wrapper library should be have the same name as the Python module. If the target name needs to be different for a reason, the library name can be specified with

`set_target_properties(_add_two_ints_wrapper_cpp PROPERTIES OUTPUT_NAME correct_library_name).`

The line

```
catkin_python_setup()
```

is used to export the Python module and is associated with the file `setup.py`

Afficher/masquer les numéros de lignes

```
1 roscd python_bindings_tutorial
2 touch setup.py
```

The content of `setup.py` will be:

Afficher/masquer les numéros de lignes

```
1 # ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD
2
3 from distutils.core import setup
4 from catkin_pkg.python_setup import generate_distutils_setup
5
6 # fetch values from package.xml
7 setup_args = generate_distutils_setup(
8     packages=['python_bindings_tutorial'],
9     package_dir={'': 'src'})
10
11 setup(**setup_args)
```

We then build the package with `catkin_make`.

1.5 Testing the binding

You can now use the binding in Python scripts or in a Python shell

Afficher/masquer les numéros de lignes

```
1 from std_msgs.msg import Int64
2 from python_bindings_tutorial import AddTwoInts
3 a = Int64(4)
4 b = Int64(2)
5 addtwoints = AddTwoInts()
6 sum = addtwoints.add(a, b)
7 sum
```

2. Class with NodeHandle

As stated, a Python call to `rospy.init_node` does not initialize roscpp. If roscpp is not initialized, instanciating `ros::NodeHandle` will lead to a fatal error. A solution for this is provided by the moveit_ros_planning_interface (http://moveit.ros.org). In any Python script that uses the wrapped class, two lines need to be added before instanciating `AddTwoInts`:

Afficher/masquer les numéros de lignes

```
1 from moveit_ros_planning_interface._moveit_roscpp_initializer import roscpp_init
2 roscpp_init('node_name', [])
```

This will create a ROS node. The advantage of this method over a classical ROS service server/client implementation is thus not as clear as in the case without the need of `ros::NodeHandle`.

3. Class with container of ROS messages

If the class uses containers of ROS messages, an extra conversion step must be added. This step is not specific to ROS but is part of the Boost Python library.

3.1 `std::vector<M>` as return type

In the file where the C++ wrapper class is defined, add these lines:

Afficher/masquer les numéros de lignes

```
 1 // Extracted from https://gist.github.com/avli/b0bf77449b090b768663.
 2 template<class T>
 3 struct vector_to_python
 4 {
 5   static PyObject* convert(const std::vector<T>& vec)
 6   {
 7     boost::python::list* l = new boost::python::list();
 8     for(std::size_t i = 0; i < vec.size(); i++)
 9       (*l).append(vec[i]);
10
11     return l->ptr();
12   }
13 };
14
15 class Wrapper : public WrappedClass
16 {
17 /*
18 ...
19 */
20     std::vector<std::string> wrapper_fun(const std::string str_msg)
21     {
22       /* ... */
23     }
24
25 };
26
27 BOOST_PYTHON_MODULE(module_wrapper_cpp)
28 {
29   boost::python::class_<Wrapper>("Wrapper", bp::init</* ... */>())
30     .def("fun", &Wrapper::wrapper_fun);
31
32   boost::python::to_python_converter<std::vector<std::string, std::allocator<std::stri
ng> >, vector_to_python<std::string> >();
33 }
```

3.2 `std::vector<M>` as argument type

Cf. Boost.Python documentation.

Wiki: ROS/Tutorials/Using a C++ class in Python (dernière édition le 2017-06-16 21:36:43 par kyrofa (/kyrofa))