NOTE 259: libpyrap, casacore-python converters

Ger van Diepen, ASTRON Dwingeloo

November 10, 2006

Abstract

pyrap is a Python binding to casacore classes using Boost.Python. It consists of a set of standard converters and bindings to the classes. As much as possible the bindings are the same as in glish.

A pdf version of this note is available.

Contents

1 Introduction

Since long glish bindings to the Casacore system have been in place. Quite recently Python bindings have been created in the general casapy framework using tools like CCMTools, Xerces, Xalan, and IDL. Albeit very flexible, it is quite complicated and it is not straightforward to build on other systems than RedHat and OS-X.

Therefore an attempt has been made to make a simpler Python binding using Boost.Python. This proved to be very easy and succesful. The binding consists of two parts:

- Converters to translate objects between Python and C++.
- Class wrappers to map a C++ class and its functions to Python.

The Python numarray and numpy (version 1.0 or higher) packages are supported. At build time one can choose which ones should be used.

2 Converters

Boost.Python offers a nice way to convert objects to and from Python. Ralf W. Grosse-Kunstleve ¡rwgk@yahoo.com¿ of Lawrence Berkeley National Laboratory has built converters for standard STL containers. This has been extended to convert to/from other objects.
The following C++ objects are currently supported:

- scalars (bool, integer, real, complex)
- `std::string`
- `casa::String`
- `std::vector<T>`
- `casa::Vector<T>`
- `casa::IPosition`
- `casa::Record`
- `casa::ValueHolder`
- exceptions (`casa::IterError` and `std::exception`)

These C++ objects can usually be created from several types of Python objects and are converted to a specific Python object.

- A vector or IPosition object is converted to a Python list.
  It can be constructed from the following Python objects:
  - scalar

  - list or tuple
  - numarray scalar or 1-dim array
  - numpy scalar or 1-dim array

  Note that a list or tuple of arbitrary objects can be given. For example, it is possible to get a `Vector<TableProxy>` from Python.

- A casa::Record is mapped to a Python dict.
- Every C++ exception is mapped to a Python `RunTimeError` exception. However, `casa::IterError` is special and is mapped to an end-of-iteration exception (`StopIteration`) in Python.
- A casa::ValueHolder is a special Casacore object that can hold a record or a scalar value or n-dim array of many types (bool, numeric, string). It is meant to conceal the actual type which is useful in functions that can accept a variety of types (like `getcell` in the table binding).
  Converting a ValueHolder to Python creates the appropriate Python scalar, array, or dict object. When converting from Python to ValueHolder, the appropriate internal ValueHolder value is constructed; a list, tuple, and array object are converted to an Casacore array in the ValueHolder.

It means there is no direct Array conversion to/from Python. A ValueHolder object is always needed to do the conversion. Note that this is a cheap operation, as it uses Array reference semantics. ValueHolder has functions to convert between types, so one can get out an Array with the required type.

2.1 Array conversion to/from numpy and numarray

Casacore arrays are kept in Fortran-order, while Python arrays are kept in C-order. It was felt that the Python interface should be as pythonic as possible. Therefore it was decided that the array axes are reversed when converting to/from Python. The values in an IPosition object (describing shape or position) are also reversed when converting to/from Python.
Note that although numarray and numpy have Fortran-array provisions by setting the appropriate internal strides, they do not really support them. When adding, for instance, the scalar value 0 to a Fortran-array, the result is a transposed version of the original (which can be a quite expensive operation).

A function binding could be such that shape information is passed via, say, a `Record` and not via an `IPosition` object. In that case its values are not reversed automatically, so the programmer is responsible for doing it.

An Casacore array is returned to Python as an array object containing a copy of the Casacore array data. If pyrap has been built with support for only one Python array package (numpy or numarray), it is clear which array

type is returned. If support for both packages has been built in, by default an array of the imported package is returned. If both or no array packages have been imported, a numpy array is returned.
Note that there is no support for the old Numeric package.

An Casacore array constructed from a Python array is regarded as a temporary object. So if possible, the Casacore array refers to the Python array data to avoid a needless copy. This is not possible if the element size in Python differs from Casacore. It is also not possible if the Python array is not contiguous (or not aligned or byte swapped). In those cases a copy is made.

A few more numarray/numpy specific issues are dealt with:

- An empty N-dim Casacore array (i.e. an array containing no elements) is returned as an empty N-dim Python array. If the dimensionality is zero, it is returned as an empty 1-dim array, to prevent numarray/numpy from treating it as a scalar value.
- In numarray `array()` results in `Py_None`. This is accepted by the converters as an empty 1-dim array.
- Empty arrays can be constructed in Python using empty lists. For example, `array([[]])` results in an empty 2-dim array. The converters accept such empty N-dim Python arrays. The type of an empty array is set to Int by numarray and to Double by numpy.
- Because the type of an empty Python array cannot easily be set, the converters can convert an empty integer or real array to any type.
- The converters accept a numpy string array. However, it is returned to Python as the special `dict` object described above.

3 Class wrappers

Usually a binding to an existing Proxy class is made, for example `TableProxy`, which should be the same class used in the glish-binding. For a simple binding, only some simple C++ code has to be written in pyrap/apps/pyxx/pyxx.cc, where XX is the name of the class.

```
// Include files for converters being used.
#include <pyrap/Converters/PycExcp.h>
#include <pyrap/Converters/PycBasicData.h>
#include <pyrap/Converters/PycRecord.h>
// Include file for boost python.
#include <boost/python.hpp>

using namespace boost::python;
```

```
namespace casa { namespace pyrap {
  void wrap_xx()
  {
    // Define the class; "xx" is the class name in Python.
    class_<XX> ("xx")
      // Define the constructor.
      // Multiple constructors can be defined.
      // They have to have different number of arguments.
      .def (init<>())
      // Add a .def line for each function to be wrapped.
      // An arg line should be added for each argument giving
      // its name and possibly default value.
      .def ("func1", &XX::func1,
            (boost::python::arg("arg1"),
             boost::python::arg("arg2")=0))
    ;
  }
}}

BOOST_PYTHON_MODULE(_xx)
{
  // Register the conversion functions.
  casa::pyrap::register_convert_excp();
  casa::pyrap::register_convert_basicdata();
  casa::pyrap::register_convert_casa_record();
  // Initialize the wrapping.
  casa::pyrap::wrap_xx();
}
```

Python requires for each package a file `__init__.py`, so such an empty file should be created as well.

3.1 More complicated wrappers

Sometimes a C++ function cannot be wrapped directly, because the argument order needs to be changed or because some extra Python checks are necessary. In such a case the class needs to be implemented in Python itself. The C++ wrapped class name needs to get a different name, usually by preceeding it with an underscore like:

```
    class_<XX> ("_xx")
```

The Python class should be derived from it and implement the constructor by calling the constructor of _xx.

```
class xx(_xx):
    def __init__(self):
        _xx.__init__(self)
```

Now `xx` inherits all functions from `_xx`. The required function can be written in Python like

```
def func1 (self, arg1, arg2):
    return self._func1 (arg2, arg1);
```

Note that in the wrapper the function name also needs to be preceeded by an underscore to make it different.

3.2 Combining multiple classes

Sometimes one wants to combine multiple classes in a package. A example is package `pycasatable` which contains the classes `table`, `tablecolumn`, `tablerow`, `tableiter`, and `tableindex`. One is referred to the code of this package (in code/pyrap/apps/pycasatable) to see how to do it.

4 Python specifics

Besides an array being in C-order, there are a few more Python specific issues.

- Indexing starts at 0 (vs. 1 in glish).
- The end value in a range like `[10:20]` is exclusive (vs. inclusive in glish). Furthermore Python supports a step and reversed ranges.
- Where useful, the function `__str__` should be added giving the name of the object. This function is used when printing an object.
- Where useful, the functions `__len__`, `__setitem__(index, value)`, and `__getitem__(index)` should be added to make it possible that a user indexes an object directly like `tabcol[i]` or `tabcol[start:stop:step]`.
- When these functions are added, Python supports iteration in an object. Explicit iteration can also be done by adding the functions `__iter__` and `next`. At the end `next` should raise the Python `StopIteration` exception (or throw `casa::IterError` when implemented in C++) to stop the iteration.

5 Building pyrap

pyrap is part of the Casacore source tree and can be built as part of the Casacore build. Work is underway to build it separately as a Python egg. To build it in Casacore the following has to be added to the local makedefs:

- Definitions for PYTHON telling the Python locations.
- Definitions for BOOST telling the Boost.Python locations.

- `AUXILIARY += pyrap`

When doing it this way, pyrap is built with support for numarray. In order to build it with support for numpy as well, the following has to be added to the local makedefs:

```
PYTHONINCD += 'the path where numpy/arrayobject.h can be found'
PYTHONDEFS += -DAIPS_USENUMPY
```

If only numpy should be supported, the last line should be replaced by:

```
PYTHONDEFS := -DAIPS_USENUMPY
```

When building pyrap, it is by default built as the shared library `libpyrap.so` containing the various converters and associated Casacore code. It can be built as a static library by adding the appropriate `LIBpyrap` line to the local makedefs. However, it is strongly advised to build it as a shared library because:

- If multiple python packages are used, each of them registers the converters in Boost.Python. If the same converter is multiply registered, Boost.Python gives a warning. To avoid these warnings pyrap keeps a map of registered converters. It is clear that this map is only useful (i.e. shared between python packages) if pyrap is built as a shared library.

- Multiple python packages share the converter code, thus reducing the memory footprint.

A package `pyxx` is built as a shared library `_pyxx.so` residing in directory $AIPSPATH/python$PYTHONVERSION/pyxx. All .py files are also put in this directory and compiled to .pyo and .pyc files.

On Mac OS-X Python libraries have to be built as bundles. For this the local makedefs need to set some make variables:

```
PYLDSOPTS  := -bundle
PYSFXSHAR  := so
SFXSHAR    := dylib
LDSOPTS    := -dynamiclib -single_module
PYTHONLIBS := -framework Python
```