Boost::Python

Bernhard Kaplan

Kirchhoff Institute for Physics, University of Heidelberg

What is boost?

What is boost?

- a collection of peer-reviewed, open source libraries that extend the functionality of C++

What is boost?

- a collection of peer-reviewed, open source libraries that extend the functionality of C++
- Boost::Python enables interoperability between C++ and Python

What is boost?

- a collection of peer-reviewed, open source libraries that extend the functionality of C++
- Boost::Python enables interoperability between C++ and Python
- Why would one have a C++ - Python Interface?

What is boost?

- a collection of peer-reviewed, open source libraries that extend the functionality of C++
- Boost::Python enables interoperability between C++ and Python
- Why would one have a C++ - Python Interface?

⇒ Use advantages from both languages:
Python's flexibility and efficiency of C++

Why Boost::Python?

- Comprehensive lifetime management facilities for low-level C++ pointers and references *(CallPolicies)*

Why Boost::Python?

- Comprehensive lifetime management facilities for low-level C++ pointers and references *(CallPolicies)*
- Support for C++ virtual functions that can be overridden in Python

Why Boost::Python?

- Comprehensive lifetime management facilities for low-level C++ pointers and references *(CallPolicies)*
- Support for C++ virtual functions that can be overridden in Python
- Wrapping of overloaded operators, STL container classes

Why Boost::Python?

- Comprehensive lifetime management facilities for low-level C++ pointers and references *(CallPolicies)*
- Support for C++ virtual functions that can be overridden in Python
- Wrapping of overloaded operators, STL container classes
- Support for organizing extensions as Python packages, with a central registry for inter-language type conversions

Why Boost::Python?

- Comprehensive lifetime management facilities for low-level C++ pointers and references *(CallPolicies)*
- Support for C++ virtual functions that can be overridden in Python
- Wrapping of overloaded operators, STL container classes
- Support for organizing extensions as Python packages, with a central registry for inter-language type conversions

⇒ Expose C++ classes and functions to Python without an additional wrapping language, simply use C++ compiler

Extending: example

```
#include <boost/python.hpp>
using namespace boost::python;
class A{    // simple example class
    public:
      A(int n) { value = n; }
      void set(int n) { value = n; }
      int get() { return value; }
    private:
      int value;
};
BOOST_PYTHON_MODULE(module_A){
    // Create the Python type object for our extension class and
    // define __init__ function.

    class_<A>("A", init<int>())
        .def("get", &A::get, "docstring here") //Add a regular member function
        .add_property("value", &A::get, &A::set)
    ;
}
```

Extending: example

Compile the C++ file:

```
g++ -I/usr/include/boost -I/usr/include/python2.5
    -l$(BOOSTLIBRARY) -fPIC -shared
    -o module_A.so class_A.cpp
```

Use the module in python:

```
In [1]: import module_A as m
In [2]: a = m.A(123)
In [3]: a.get()
Out[3]: 123
In [4]: a.value = 321
In [5]: a.value
Out[5]: 321
```

Wrapping STL containers

```
#include <boost/python.hpp>
#include <boost/python/suite/indexing/vector_indexing_suite.hpp>
using namespace boost::python;
BOOST_PYTHON_MODULE(vector_wrapper){
  using namespace boost::python;

  //! python access to stl integer vectors
  class_< std::vector<int> >("vectorInt")
      .def(vector_indexing_suite<std::vector<int> >())
  ;

  //! python access to stl vectors of integer vectors
  class_< std::vector< std::vector<int> > >("vectorVectorInt")
      .def(vector_indexing_suite<std::vector< std::vector<int> > >())
;}
```

In Python:

```
b = vector_wrapper.vectorInt()
b.append(123); b[0]; len(b)
```

Overloading

```
class X{
    bool f(int a){return true;}
    bool f(int a, double b){return true;}
    int  f(int a, int b, int c){return a+b+c;}
};
// write some "thin wrappers"
bool    (X::*fx1)(int)              = &X::f;
bool    (X::*fx2)(int, double)      = &X::f;
int     (X::*fx3)(int, int, int)    = &X::f;

.def("f", fx1)
.def("f", fx2)
.def("f", fx3)
```

Wrapping of functions with default arguments works very similar.

Call Policies

```
X& f(Y& y, Z* z){
    y.z = z;
    return y.x;
}
>>> x = f(y, z)      # x refers to some C++ class X
>>> del y            # x becomes a dangling ref.
>>> x.some_method() # BOOM!
```

Call Policies

```
X& f(Y& y, Z* z){
    y.z = z;
    return y.x;
}
>>> x = f(y, z)     # x refers to some C++ class X
>>> del y           # x becomes a dangling ref.
>>> x.some_method() # BOOM!
```

```
.def("f", f,
     return_internal_reference<1,
         with_custodian_and_ward<1, 2> >());
    // 1) Ties lifetime of one argument to that of result
    // 2) Lifetime of the argument the 2nd argument(Z* z, ward)
    //    is dependent on the lifetime of the 1st argument custodian
```

References

- www.boost.org/doc/libs/1_35_0/libs/python/doc/index.html
- wiki.python.org/moin/boost.python
- David Abrahams, Ralf W. Grosse-Kunstleve "Building Hybrid Systems with Boost.Python" www.boost-consulting.com/writing/bpl.html