Boost - Python second take

Table of Contents

simple example

```
3
4  #include <boost/python/module.hpp>
5  #include <boost/python/def.hpp>
6  #include <cmath>
7  //#include <stdio.h>
8
9
10 std::string greet(const char * name)
11 {
12   printf("\n%s\n", name);
13   std::string a(name);
14   return a;
15 }
16
17 int square(int value)
18 {
19   return value*value;
20 }
21
22 double myerf(double value)
23 {
24   return erf(value);
25 }
26
27 namespace python = boost::python;
28
29 BOOST_PYTHON_MODULE(hello)
30 {
31   python::def("greet",greet);
32   python::def("square",square);
33   python::def("erf",myerf);
34 }
35
```

```
#/usr/bin/env python
import hello
import random

print dir(hello)

a = hello.greet('Peter')

sum = 0.0
for i in xrange(100):
    sum += hello.erf(random.uniform(0,1))
print sum
print hello.square(int(sum))

```

Classes

Consider a C++ class/struct that we want to expose to Python:

```
struct World
{
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
    std::string msg;
};
```

We can expose this to Python by writing a corresponding Boost.Python C++ Wrapper:

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
    class_<World>("World")
        .def("greet", &World::greet)
        .def("set", &World::set)
    ;
}
```

Here, we wrote a C++ class wrapper that exposes the member functions `greet` and `set`. Now, after building our module as a shared library, we may use our class `World` in Python. Here's a sample Python session:

```
>>> import hello
>>> planet = hello.World()
>>> planet.set('howdy')
>>> planet.greet()
'howdy'
```

Custom Constructor

```cpp
#include <string>
#include <sstream>

struct Ctor
{
    Ctor(std::string msg) : mMsg(msg) {}
    Ctor(double x,double y)
        {
            std::stringstream os;
            os << x << ":" << y << std::ends;
            set(os.str());
    }
    void set(std::string msg) { mMsg = msg; }
    std::string greet() { return mMsg; }
    std::string mMsg;
};

#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(ctor)
{
    class_<Ctor>("Ctor", init<std::string>())
        .def(init<double, double>())
        .def("greet", &Ctor::greet)
        .def("set", &Ctor::set)
    ;

}
```

fine-tuning access

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


struct Var
{
    Var(std::string name) : name(name), value() {}
    std::string const name;
    float value;
};

using namespace boost::python;

BOOST_PYTHON_MODULE(var)
{
    class_<Var>("Var", init<std::string>())
    .def_readonly("name", &Var::name)
    .def_readwrite("value", &Var::value);
}
```

Then, in Python, assuming we have placed our Var class inside the namespace hello as we did before:

```
>>> x = hello.Var('pi')
>>> x.value = 3.14
>>> print x.name, 'is around', x.value
pi is around 3.14
```

Note that `name` is exposed as **read-only** while `value` is exposed as **read-write**.

```
>>> x.name = 'e' # can't change name
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: can't set attribute
```

Class Properties

In C++, classes with public data members are usually frowned upon. Well designed classes that take advantage of encapsulation hide the class' data members. The only way to access the class' data is through access (getter/setter) functions. Access functions expose class properties. Here's an example:

```
struct Num
{
    Num();
    float get() const;
    void set(float value);
    ...
};
```

However, in Python attribute access is fine; it doesn't neccessarily break encapsulation to let users handle attributes directly, because the attributes can just be a different syntax for a method call. Wrapping our `Num` class using Boost.Python:

```
class_<Num>("Num")
    .add_property("rovalue", &Num::get)
    .add_property("value", &Num::get, &Num::set);
```

And at last, in Python:

```
>>> x = Num()
>>> x.value = 3.14
>>> x.value, x.rovalue
(3.14, 3.14)
>>> x.rovalue = 2.17 # error!
```

Take note that the class property `rovalue` is exposed as **read-only** since the `rovalue` setter member function is not passed in:

```
.add_property("rovalue", &Num::get)
```

Object Interface

Python is dynamically typed, unlike C++ which is statically typed. Python variables may hold an integer, a float, list, dict, tuple, str, long etc., among other things. In the viewpoint of Boost.Python and C++, these Pythonic variables are just instances of class `object`. We will see in this chapter how to deal with Python objects.

As mentioned, one of the goals of Boost.Python is to provide a bidirectional mapping between C++ and Python while maintaining the Python feel. Boost.Python C++ `objects` are as close as possible to Python. This should minimize the learning curve significantly.

To illustrate, this Python code snippet:

```
def f(x, y):
     if (y == 'foo'):
          x[3:7] = 'bar'
     else:
          x.items += y(3, x)
     return x

def getfunc():
   return f;
```

Can be rewritten in C++ using Boost.Python facilities this way:

```
object f(object x, object y) {
     if (y == "foo")
          x.slice(3,7) = "bar";
     else
          x.attr("items") += y(3, x);
     return x;
}
object getfunc() {
    return object(f);
}
```

Derived Object types

Boost.Python comes with a set of derived `object` types corresponding to that of Python's:

- list
- dict
- tuple
- str
- long_
- enum

These derived `object` types act like real Python types. For instance:

```
str(1) ==> "1"
```

Wherever appropriate, a particular derived `object` has corresponding Python type's methods. For instance, `dict` has a `keys()` method:

```
d.keys()
```

`make_tuple` is provided for declaring *tuple literals*. Example:

```
make_tuple(123, 'D', "Hello, World", 0.0);
```

```
#include <boost/python.hpp>
#include <string>
#include <stdio.h>

namespace py = boost::python;

// dummy class
class drow{
public:
  int size;
  std::string word;
  drow(py::list words);
  py::list get_chars();
};
```

```
int g(py::object x)
{
  py::extract<int> get_int(x);
  if (get_int.check())
    return get_int();
  else
    {
py::extract<float> get_float(x);
      if (get_float.check())
        return get_float();
    }
    return -1;
}
```

```
// example of passing python list
//as argument (to constructor)
drow::drow(py::list l){

    std::string w;
    std::string token;
    size = len(l);
    printf("size=%i\n",size );
    for (int i = 0; i < size ; i++){
      token = std::to_string(g(l[i]));
        printf("token=%s\n",token.c_str());
        w += token;
    }
    this -> word = w;
}
```

```
// example of returning a python list
py::list drow::get_chars(){
    py::list char_vec;
    for (int c=0; c < size; c++){
        char_vec.append(word[c]);
    }
    return char_vec;
}

// binding with python
BOOST_PYTHON_MODULE(drow){
    py::class_<drow>("drow", py::init<py::list>()
        .def("get_chars", &drow::get_chars);
}
```

```python
#!/usr/bin/env python
import drow

print dir(drow)

a = drow.drow([1,'aaaa',6,'d'])
print a.get_chars()
```

```
nagal:L20>python t.py
['__doc__', '__file__', '__name__', '__package__', 'drow']
size=4
token=1
token=-1
token=6
token=-1
['1', '-', '1', '6']
```