



USING SWIG TO BIND C++ TO PYTHON

By Teresa L. Cottom

AN INCREASINGLY POPULAR APPROACH TO SCIENTIFIC COMPUTING IS TO COMBINE PYTHON AND COMPILED MODULES. SUCH AN APPROACH MERGES THE HIGH PERFORMANCE

typically found in compiled routines with the interface of a flexible, scalable, and easy-to-learn interpreted language. Although using C to hand-code extensions to Python binds the latter to a given compiled asset in C++, programmers who used C++'s more advanced features (until recently) lacked the automated support available in Fortran and C.

One tool for creating Python bindings to C is David Beazley's Simplified Wrapper and Interface Generator, SWIG—an open-source application used by a large and ever-expanding community—began as an effort to expose physics packages in a large parallel simulation code to interpreted languages. SWIG preprocesses C and C++ code and generates library bindings in several interpreted languages including Python, Perl, Tcl, and Java.

Recent improvements to SWIG provide greater support for binding C++ code. SWIG now creates, for example, bindings for some of C++'s more advanced features such as templates and exceptions. This article explores how SWIG does this by examining a series of small C++ code examples. I hope this cookbook approach will help you produce your own bindings with ease.

Automated Bindings that Use Header Files

To create a binding to a C++ source, SWIG must have information on what routines and classes to bind and what types of arguments and return values those routines and classes should have. SWIG parses a C++ interface file that contains the assortment of rules and definitions needed to generate the Python module. An interface file can be either a C++ header file with condensed SWIG directives or a stand-alone file that indicates appropriate header files and definitions.

Using a header file is quicker and easier to maintain,

but specifying the information explicitly allows more control over the interface. As you will see, SWIG's facilities for controlling code generation offers the best of both worlds. This article presents the interface directives as a separate file.

To begin, let's create a simple binding using a header file. We'll wrap the C++ function `hello_world`:

```
#include <iostream>

void hello_world() {
    std::cout << "hello world" << std::endl;
}
```

The following interface file sufficiently establishes the rules and definitions for creating a module called `myexample`, which has a function `hello_world` available to Python:

```
%module myexample
%{
#include "myexample.h"
%}
#include "myexample.h"
```

Typically, the interface file begins by defining the Python module name as specified via the `%module` directive. For proper compilation, the `%{ }` block directive supports C++ code. The code contained in this block is copied verbatim to the auto-generated wrapper code. The C++ block commonly ensures that the wrapper code has all proper `#include(s)`. Next, the interface file establishes the rules to be applied to all subsequent Python binding definitions; a default set of rules exists and, in this instance, suffices. Lastly, the interface file specifies the Python binding definitions—prototypes of classes and functions to be wrapped. Prototypes either can be specified manually or automatically via the `%include` directive. (Figure 1 illustrates the full process.)

After parsing all the rules and definitions in the interface file, the SWIG executable generates a C++ source file and a Python source file. Compile the C++ source file with the same compiler used to build the underlying C++ object

code. Link the object file into a shared library following the naming convention of `<module name>`. The Python source file produced by SWIG, `myexample.py`, shadows the compiled and linked C++ module. This shadow provides Python with the ability to treat all wrapped C++ objects as true Python objects.

For a Linux system, I compiled and linked using these commands:

```
g++ -O2 -c myexample_wrap.
  cxx -DHAVE_CONFIG_H
  I/usr/local/include/
  python2.2 - I/usr/local/
  include/python2.2/config
g++ -shared -O2 myexample_wrap.o -o
  _myexample.so
```

Install the shared library module and the Python source module in an area accessible to the Python executable, or make the library location available via the environment. Now, the C++ function `hello_world` is available for use as a Python function:

```
>>> import myexample
>>> myexample.hello_world()
hello world
>>>
```

Function Overloading

C++ provides function-overloading support to loosely associate functions that accomplish similar tasks. Functions share (overload) the same name but operate on different arguments. The number of arguments and argument types determine which function invokes. SWIG provides easy, seamless support for C++ function overloading.

The following header file overloads `printit` with three C++ functions:

```
#include <iostream>
void printit(double a) {
    std::cout << "Double version:" << a <<
        std::endl;
}
void printit(int a) {
```

SWIG Interface File:
myexample.i

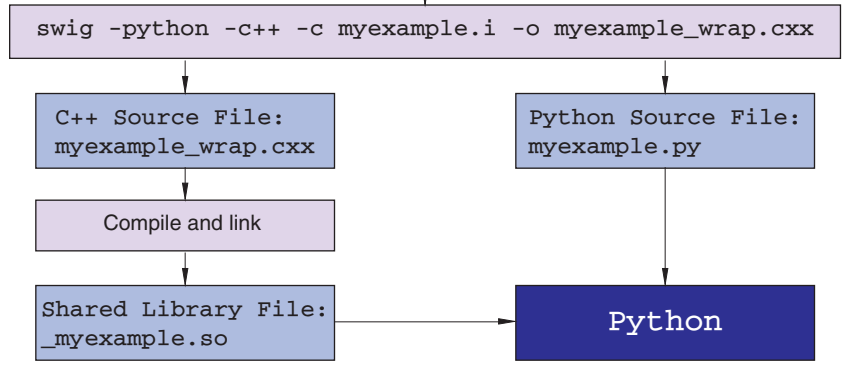


Figure 1. Using SWIG to build a Python module. SWIG, given an interface file, generates the necessary source code for accessing C++ assets in Python.

```
std::cout << "Int version:" << a <<
    std::endl;
}
void printit(char a) {
    std::cout << "Char version:" << a <<
        std::endl;
}
```

The Python module generated here overloads the function `printit` in Python (essentially, there is only one resulting `printit` function in Python). This function dispatches to one of the three possible C++ `printit` functions depending on the argument type:

```
%module overload
%{
#include "overload.h"
}%
#include "overload.h"
```

Let's look at the correct invocation of the underlying C++ functions based on argument type. Keep in mind that a double argument invokes the double parameter version, an integer argument invokes the integer parameter version, and a character argument invokes the character parameter version:

```
>>>from overload import *
>>>printit(7.0)
Double version: 7
>>>printit(3)
Int version: 3
>>>printit('a')
Char version: a
>>>
```

Python Bindings for Classes

SWIG simplifies the process of creating Python bindings for classes because public data members and their functions wrap automatically. The protected and private members conveniently remain protected and private because no bindings are generated.

The C++ class to be wrapped, `Simple`, contains public, private, and protected accessible data and member functions:

```
class Simple {
public:
    Simple():pubA(0),proA(1),priA(2) {}
    int pubF() { return pubA; }
    int pubA;
protected:
    int proF() { return proA; }
    int proA;
private:
    int priF() { return priA; }
    int priA;
};
```

Compare this with an interface file that generates the Python bindings for the C++ `Simple` class:

```
%module mysimple
%{
#include "simple.h"
%}
#include "simple.h"
```

Now, let's look at the members available from the `Simple` class in Python. Members `pubA` and `pubF` grant access to the C++ public members. No bindings exist for the protected and private members:

```
>>> from mysimple import *
>>> a = Simple()
>>> dir(Simple)
['__class__', '__del__', '__delattr__',
 '__dict__', '__doc__', '__getattr__',
 '__getattribute__', '__hash__', '__init__',
 '__module__', '__new__', '__reduce__',
 '__repr__', '__setattr__', '__str__',
 '__swig_getmethods__', '__swig_setmeth-
ods__', '__weakref__', 'pubA', 'pubF' ]
>>> print a.pubA, a.pubF(), a
0 0 <C Simple instance at e0490e400100000
```

```
_p_Simple>
>>>
```

SWIG provides the capability to extend a C++ class's functionality in Python. The `%extend` directive expands the exposed Python class with the requested additional functionality. Here's a more extensive interface file for creating bindings for the `Simple` class:

```
%module mysimpleextend
%{
#define protected public
#define private public
#include "simple.h"
%}
%feature("shadow") Simple::_priF %{
def __str__(self):
    return "< Simple "+str(self.pubA)+" "\
        +str(self._proF())+ " "\
        +str(apply(eval("_"+__name__
            .Simple__priF,[self]))+" >"
%}
%extend Simple {
    int _proF() { return self->proF(); }
    int _priF() { return self->priF(); }
};

#include "simple.h"
```

The interface file displays an example of extending the C++ `Simple` class for a `_proF` and a `_priF` member binding in Python. The `_proF` binding grants access to the C++ protected function `proF`, thus making the protected function available to any Python derived classes.

Exposure of `_priF` grants access to the C++ class private member `priF`. Note that access is granted only to the Python shadow class's extended functions. In this case, the exposure of `_priF` lets the Python shadow function `__str__` access the C++ private member and implement a complete printing of the instance. The `%feature("shadow") Simple::_priF` rule guarantees that the `_priF` will not be available from the Python class. The `%feature("shadow")` directive overrides the default shadow code generated for the specified function. In this case, the Python shadow code for `_priF` is replaced with a `__str__` implementation. As such, `_priF` is not accessible as a member function of `Simple` from Python:

```
>>> from mysimpleextend import *
```

Café Dubois

Pike and Xenofarm

Early adopters of new languages (the geek version of masochists) might be interested in looking at Pike (<http://pike.ida.liu.se>). A creative bunch of people seems to be involved. The tool they use for building has spun off as Xenofarm. The authors describe Xenofarm this way:

“Xenofarm is a distributed system for cross platform compilation and verification of software components. It is primarily used to help achieve and maintain portable source code across hardware architectures, operating system and libraries and to quickly detect and fix regression bugs in source code.”

Xenofarm is really different from anything I’ve seen before. Of course, people who try new build tools are the geek version of kamikazes, but if you survive, please let me know what you think of it. We all desperately need a tool for building third-party software reliably.

SCONS Grows

Speaking of kamikazes, the interest in SCONS continues to grow (<http://scons.sf.net>). Designed from the ground up for cross-platform builds—and known to work on Linux, Posix, Windows NT, Mac OS X, and OS/2—SCONS now includes built-in support for Fortran, C, C++, Yacc, and Lex and for building Tex and LaTeX documents.

Dive into Python

You’ve read a lot about Python in this department, but you’re too cheap to buy a book (heck, you even borrowed this copy of *CiSE*, right?). Your local Borders keeps throwing you out at closing time. You need a Python book you can read online or download. Proceed to <http://diveintopython.org> to read Mark Pilgrim’s *Dive Into Python*. Mark is an extremely talented and prolific writer. The chapter on Unit Testing is damn near perfect and has been a big influence on me. This book is published under the GNU Free Documentation License (GFDL), the book version of the GPL.

How to Teach the Kids

The modern high-school curriculum is so full, and in many places the number of class periods per day so limited, that a science-minded student essentially has no electives. It is hard for many of them to squeeze in a class on program-



ming at all, and you probably won’t be too happy with the result even in the best schools. If you think finding a good math teacher is hard, imagine how hard it is to find someone with any real software experience. You can’t teach your own kid anything at that age, and most books about programming languages assume you already know how to program. What to do?

One place that did manage to create a decent course is Yorktown High School in Arlington, Virginia. Out of that effort came a neat book (also GFDL licensed) *How to Think Like a Computer Scientist: Learning with Python*, by Allen Downey, Jeff Elkner, and Chris Meyers. The three authors are a college professor, a high-school teacher, and a professional programmer, respectively. You can download this book or read it online at <http://greenteapress.com/thinkpython.html>. I clicked from there over to Amazon and bought a printed copy for my kid. The unique thing about this book is that it is written for the true programming novice.

Since Arlington was near the site of the Python conference, I had a chance to meet some of the Yorktown students. They were a wonderful bunch of kids (we all enjoyed having them at the conference), and they showed a funny movie they had made about the virtues of Python.

Some kids don’t need much help. A group at our high school came in second to MIT in an autonomous submarine contest. The sub has an onboard Linux box running the sensors and engines. MIT, which is not run by dummies, tried to recruit the seniors. Ah, geeks! These are my people.

```
>>> a = Simple()
>>> print a
< Simple 0 1 2 >
>>>
```

Default Arguments

C++ provides for default argument support for function definitions. Because Python also provides this capability, we want the Python functions to have the same default argu-

ments as the C++ function it wraps.

SWIG automatically carries over C++ default arguments into the Python bindings. Being a preprocessor, SWIG can parse the default values as it processes the prototypes to bind. When it generates the bindings, it simply writes these default values to the auto-generated code. Because you only need to provide default values once, chances of error reduce—another great advantage of a preprocessing tool.

Let's look at two structures to wrap, a simple structure `A` and a class `Keeper`. The constructor for `Keeper` contains both a native and a user-defined type parameter. Both parameters come supplied with default arguments:

```
struct A{
    A(int i=7):a(i) {}
    A(const A& b) { a=b.a; }
    int a;
};
    A defaultA;

class Keeper {
public:
    Keeper(double a=16.3, A& b = defaultA):
        aa(a),bb(b) {}
    double aa;
    A bb;
};
```

The interface file carries over default arguments from C++ to Python:

```
%module defaultargs
%{
#include "defaultargs.h"
%}
#include "defaultargs.h"
```

SWIG correctly binds default arguments for both native and user-defined types:

```
>>> from defaultargs import *
>>> a = Keeper()
>>> print a.aa, a.bb.a
16.3 7
>>> a = Keeper(3.8,A(10))
>>> print a.aa, a.bb.a
3.8 10
>>>
```

Keyword Arguments

Python, being a bit more flexible than C++, provides keyword argument capability. SWIG provides a mechanism to automatically establish keyword arguments for Python bindings.

The `%feature("kwargs")` directive should appear before any prototypes requiring keyword arguments. The C++ function parameter names automatically become the argument name accessible in Python.

In SWIG, providing default and keyword arguments to the bindings created for Python is simple:

```
%module keywordargs
%{
#include "defaultargs.h"
%}
%feature("kwargs");
#include "defaultargs.h"
```

Keyword support is now available:

```
>>> from keywordargs import *
>>> a = Keeper()
>>> print a.aa, a.bb.a
16.3 7
>>> a = Keeper(b=A(10))
>>> print a.aa, a.bb.a
16.3 10
>>>
```

Operator Translation

Commonly, C++ class operators are overloaded to provide custom class behavior. In many cases, there is a direct correlation of a C++ operator to a Python operator. For example, the C++ operator `==` typically corresponds to the Python function `__eq__`, and `!=` corresponds to `__ne__`. The default SWIG conversion rules consist of these typical C++-to-Python correlations, and most operator translations happen automatically.

Let's look at another example. Here's the definition of a class, `Complex`, which overloads many of the standard operators:

```
class Complex{
private:
    double rpart, ipart;
public:
    Complex(double r=0,double
        i=0):rpart(r),ipart(i) {}
```

```

Complex(const Complex& c):
    rpart(c.rpart),ipart (c.ipart) {}
Complex operator+(const
Complex& c) const {
    return Complex(rpart+
        c.rpart, ipart+ c.ipart);
}
Complex operator-(const Complex& c)
    const {
    return Complex(rpart-
        c.rpart, ipart-c.ipart);
}
Complex operator*(const Complex& c)
    const {
    return Complex(rpart*
        c.rpart - ipart* c.ipart, rpart*
        c.ipart + c.rpart*ipart);
}
Complex operator-() const {
    return Complex(-rpart, -ipart);
}
int operator==(const Complex&
c) {
    if (rpart==c.rpart && ipart==c.ipart)
        return 1;
    return 0;
}
int operator!=(const
Complex& c) {
    if (rpart!=c.rpart
        || ipart!=c.ipart)
        return 1;
    return 0;
}
double re() const { return rpart; }
double im() const { return ipart; }
};

```

Example comes from SWIG installation
Examples/python/operator

The interface file that converts the C++ operators to Python operators is incredibly simple. It also extends the Python class for a `__str__` implementation:

```

%module mycomplex
%{
#include "mycomplex.h"

```

```

%}
%extend Complex {
    char* __str__() {
        static char id[100];
        sprintf(id,"%f, %f",
            self->re(),self->im());
        return id;
    }
};
#include "mycomplex.h"

```

Let's look at an interaction with the operator-overloaded Python class:

```

>>> from mycomplex import *
>>> a = Complex(1,2)
>>> b = Complex(3,0)
>>> print a+b
(4,2)
>>> print a*b
(3,6)
>>> print a==b
0
>>>

```

Note that the C++ operator `[]` and operator `=` do not get automatic support. The desired capability of these functions, however, sometimes warrants the excuse of using the `%extend` directive to define an appropriate Python `__getitem__` and `__setitem__`.

Python Bindings for Template Instantiations

For templates, SWIG provides a simple directive for generating Python bindings. The `%template` directive creates the binding of a specific instantiation of a C++ template to a Python class.

In the next example, three different `std::vector` instantiations are exposed as classes in Python:

```

%module myvectors
%{
#include <string>
#include <vector>
%}
#include stl.i
%template(IntVector) std::vector<int>;
%template(DoubleVector) std::vector<double>;
%template(StringVector) std::vector

```

```
<std::string>;
```

The SWIG file `stl.i` provides standard wrappings for several templates from the Standard Template Library (stl), including `string` and `vector`. The vector bindings include implicit conversions from Python lists or tuples to the corresponding vector parameter in function calls. The next example demonstrates this by passing a list into the vector copy constructor.

In Python, `IntVector` functions as the integer instantiation of `std::vector`, `DoubleVector` as the double instantiation of `std::vector`, and `StringVector` as the `std::string` instantiation of `std::vector`:

```
>>> from myvectors import *
>>> a = IntVector ([2,3])
>>> print a[0:len(a)]
(2, 3)
>>> a = DoubleVector(3)
>>> print len(a)
(0.0, 0.0, 0.0)
>>>
```

Similar to standard functions, template functions overload in Python as well. Let's define a `printVector` template function that accepts a container as a parameter:

```
#include <iostream>
template <typename Container>
void printVector (const Container& c) {
    std::cout << "container: ";
    for (typename Container::const_iterator
        a=c.begin(); a!=c.end ();++a) {
        std::cout << *a << " ";
    }
    std::cout << std::endl;
}
```

The next interface file demonstrates overloading as it creates three instances of the `printVector` template function. The `%import` directive used next grants the `printvector` module means to convert externally wrapped C++ types to the appropriate Python types and vice versa. This is the cross-module support directive for SWIG:

```
%module printvector
%{
#include "printvector.h"
```

```
#include <vector>
%}
%import "myvectors.i"
%template(printVector) printVector<std::
vector<int> >;
%template(printVector) printVector<std::
vector<double> >;
%template(printVector)printVector<std::vector
<std::string> >;
```

Generating the bindings in this way, the resulting Python `printVector` function agreeably accepts as input a list or tuple of integers, doubles, or strings as well as an `IntVector`, `DoubleVector`, or `StringVector` and dispatches to the appropriate C++ instantiation of `printVector`:

```
>>> from myvectors import *
>>> from printvector import *
>>> printVector([0,1,2])
container: 0 1 2
>>> printVector(IntVector ([7,8]))
container: 7 8
>>> printVector(("string","vector"))
container: string vector
>>> printVector([3.3, 7.4])
container: 3.3 7.4
>>>
```

Exception Translation

SWIG offers a convenient, simple, and automatic translation of C++ exceptions. The automatic conversion requires the prototype to supply the exception specification. The types of thrown exceptions translate into the Python `RuntimeError` in which the message string contains the type of C++ exception for complex types, the numeric value for numeric types, or the exact message string for `std::string` or character arrays.

Let's look at three functions, which throw a variety of types:

```
#include <string>
#include <exception>
class BadException:public std::exception {
public:
    BadException(const std::
        string& m="bad"):msg(m) {}
    ~BadException () throw () {} const char*
        what() const throw() {
        return msg.c_str();
    }
```

```

private:
    std::string msg;
};
void throwBad () throw
    (BadException) {
    throw BadException("custom
    message");
}
void throwMsg(const char* msg=
    "See ya") throw (const char*) {
    throw msg;
}
void throwInt(int i=0) throw (int) {
    throw i;
}

```

Here's the interface file for creating the Python bindings with default exception translation:

```

%module myexception
%{
#include "myexception.h"
%}
#include "myexception.h"

```

The Python functions correctly translate the C++ exceptions:

```

>>> from myexception import *
>>> try:
...     throwBad()
... except RuntimeError, e:
...     print e
...
BadException
>>> try:
...     throwInt(8)
... except RuntimeError, e:
...     print e
...
8
>>>

```

SWIG also provides a method for overriding default exception handling. The `%exception` directive supplies the exception handling code for the specified functions. This directive applies the exception-handling code to a single routine, a matching subset of routines, or to every routine that doesn't already match another exception rule:

```

%module mycustomexception
%{
#include "myexception.h"
%}
%exception throwBad {
    try { $action }
    catch (BadException& e) {
        PyErr_SetString
            (PyExc_RuntimeError, e.what());
        return NULL;
    }
}
%exception {
    try { $action }
    catch (int i) {
        switch(i) {
        case 1:
            PyErr_SetString (
                PyExc_AttributeError,
                "illegal value expected positive
                number");
            break;
        case 2:
            PyErr_SetString (
                PyExc_TypeError, "illegal type
                expected double");
            break;
        default:
            PyErr_SetString
                (PyExc_RuntimeError, "unknown error
                occurred");
            break;
        }
        return NULL;
    }
    catch (const char* msg) {
        PyErr_SetString
            (PyExc_RuntimeError, msg);
        return NULL;
    }
}
#include "myexception.h"

```

The directive followed by a name, in this case `throwBad`, applies the exception-handling code to all functions named `throwBad`. The `%exception` directive without naming applies the exception-handling code to all subsequent functions that do not match previous rules. In this case, the all-

encompassing rule applies to `throwInt` and `throwMsg`.

Let's look at how the Python functions handle the C++ exceptions thrown in the custom defined method:

```
>>> from mycustomexception
import *
>>> try:
...     throwBad()
... except RuntimeError, e:
...     print e
...
custom message
>>> try:
...     throwInt(1)
... except AttributeError, e:
...     print e
...
expected positive number
>>>
```

Extending SWIG

SWIG's functionality extends in many ways. First, we can create reusable macros for common groupings of SWIG rules. The ability to use macros for multiple bindings significantly simplifies interface files. The `%define` and `%enddef` directives encompass the macro's body.

This example uses the `property` concept introduced in Python 2.2; it won't work with earlier versions of Python. A `property` is a name in a class that appears to be an attribute but, in fact, a pair of functions handles accesses to read and write it.

Let's look at how to define macro `Property`, which, given the Python class name (`pyclass`), C++ class name (`cppclass`), Python property name (`prop`), C++ class get member function (`get`), and C++ class set member function (`set`), creates a property attribute on the resulting Python class. The specified `get` member function of the C++ class invokes for retrieval of the Python property. The `set` member function invokes for assignment to the Python property. The `cppclass` can be a simple class name or a template class name with template parameters:

```
%define Property(py, cpp, prop, get, set)
%feature("shadow") cpp::set %{ %}
%feature("shadow") cpp::get %{
__swig_setmethods__["prop"]=\
    eval("__"+__name__).##py##_##set
```

```
__swig_getmethods__["prop"]=\
    eval("__"+__name__).##py##_##get
if _newclass:prop = \
    property(\
        eval("__"+__name__).##py##_##get,\
        eval("__"+__name__).##py##_##set)
%}
%enddef
```

The C++ class `Circle` contains `get` and `set` member functions for accessing protected data members:

```
class Circle {
public:
    Circle (float x=0.0,float y=0.0,float
            r=1.0): _x(x),_y(y),_r(r) {}
    float getX () { return _x; }
    float getY () { return _y; }
    float getR () { return _r; }
    void setX (float x) { _x = x; }
    void setY (float y) { _y = y; }
    void setR (float r) throw (const char*) {
        if (r <= 0.0)
            throw "radius must be a positive number";
        _r = r;
    }
protected:
    float _x,_y,_r;
};
```

The interface file that follows generates bindings for the class `Circle` in which the Python class `Circle` contains properties for `x`, `y`, and `r`. We define `CircleProperty` to demonstrate how to build macros on top of the previously declared macros:

```
%module shapes
%{
#include "shapes.h"
%}
#include "Property.i"
#define CircleProperty(prop,get,set)
Property(Circle,Circle,prop,get,set)
%enddef

CircleProperty(x,getX,setX)
CircleProperty(y,getY,setY)
CircleProperty(r,getR,setR)
```

```
%include "shapes.h"
```

When accessed, the properties invoke the appropriate corresponding C++ member function. The interface specified also hides the C++ member functions `getX`, `getY`, `getR`, `setX`, `setY`, and `setR` from the class exposed to Python.

Let's look at the Python class `Circle`'s functionality. An attempt to assign the `r` property of a `Circle` instance to a nonpositive value results in the raising of an `AttributeError`. This provides a convenient way of validating attributes for both the C++ and Python class:

```
>>> from shapes import *
>>> a = Circle()
>>> print a.x, a.y, a.r
0.0 0.0 1.0
>>> a.x = -1.0
>>> a.y = 2.0
>>> a.r = 5.5
>>> print a.x, a.y, a.r
-1.0 2.0 5.5
>>> a.r = -1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "shapes.py", line 32, in <lambda>
    __setattr__ = lambda self, name, value:
      _swig_setattr(self, Circle, name,
        value)
  File "shapes.py", line 13, in _swig
    _setattr
```

```
    if method: return method (self,value)
RuntimeError: radius must be a positive
    number
>>>
```

SWIG's approach to providing function overloading, default argument, keyword argument, and exception translation support makes it a must-have tool. Besides providing automatic support, it can also custom control any Python bindings.

This article has only touched on SWIG's power. You can extend it, for example, to generate bindings for new languages. (For details on creating new language extensions and other features, please see the documentation provided with SWIG at <http://swig.org>; Python's home page is <http://python.org>.)

To use the features shown in this article, you must use the newest version of SWIG, which is SWIG 1.3.17. You can find tools for linking Python to Fortran and a fast numerical array facility for the former at the Numerical Python page (<http://numpy.sf.net>).

Teresa L. Cottom is a computer scientist at Lawrence Livermore National Laboratory. Her research focuses on Kull, a three-dimensional massively parallel simulation system that uses a Python interpreter over C++. In the past, she has worked on parallel computational fluid dynamics codes at Pratt & Whitney. She is a graduate of Mary Washington College in Fredericksburg, Virginia. Contact her at PO Box 808 L-312, Livermore, CA 94551; cottomt@llnl.gov.

Submissions: Send two copies, one word-processed file and one PostScript file, of articles and proposals to Francis Sullivan, Editor in Chief, *CISE*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314; cise@computer.org. Submissions should not exceed 6,000 words and 10 references. All submissions are subject to editing for clarity, style, and space.

Editorial: Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *CISE* does not necessarily constitute endorsement by the IEEE, the AIP, or the IEEE Computer Society.

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. Annual subscription rates for 2001: \$40 for Computer Society members (print only) and \$52 for AIP member society members (print plus online). For more information on other subscription prices, see <http://computer.org/subscribe> or <http://ojs.aip.org/cise/subscribe.html>. Back issues cost \$10 for members, \$20 for nonmembers. This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *Computing in Science & Engineering*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 0605298. Printed in the USA.

Copyright & reprint permission: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US copyright law for private use of patrons those articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Dr., Danvers, MA 01923. For other copying, reprint, or republication permission, write to Copyright and Permissions Dept., IEEE Publications Administration, 445 Hoes Ln., PO Box 1331, Piscataway, NJ 08855-1331. Copyright © 2002 by the Institute of Electrical and Electronics Engineers Inc. All rights reserved.