Calling C/C++ from Python

Robert Lupton

25 November 2014

Motivation

You'd like to be able to write your C++ and then say things like:

```
data = Image(filename)
print data.getWidth()

if data.get(0, 0) < 10:
    debias(data)
```

Some things are so much easier in python, and you can reload files while preserving all your data — rather like using a debugger instead of `printf`; python makes a nice high-level interactive debugger for algorithms and data.

The authors of debuggers have realised this themselves, and you can now extend `gdb` and `lldb` in python — but that's a different story.

Motivation: SExtractor lite

Here's a partial emulation of a famous astronomical code:

- Read data:

```
exposure = ExposureF(fileName)
```

- Subtract background:

```
im -= makeBackground(im, bctrl).getImageF()
```

- Find detection threshold:

```
stats = makeStatistics(im, MEANCLIP | STDEVCLIP)
threshold = stats.getValue(MEANCLIP) + nsigma*stats.getValue(STDEVCLIP)
```

- Smooth image with filter:

```
convolve(smoothedIm, im, kernel)
```

- Detect sources:

```
fs = makeFootprintSet(smoothedIm, threshold, "", npixMin)
fs = makeFootprintSet(fs, grow, isotropic)
fs.setMask(exposure.getMaskedImage().getMask(), "DETECTED")
```

- Measure sources:

```
sources = fs.getFootprints()
measureSources = makeMeasureSources(exposure)
for i in range(len(objects)):
    source = Source()
       measureSources.apply(source, objects[i])
```

N.b. C++ is in red; the python is blue.

Tools to bind C/C++/Fortran to python

There is a large variety of solutions available to the problem of binding C/C++ to python; the common ones are:

- cython
- hand-crafted code using the python C API, *CPython*
- ctypes
- boost::python
- swig

There's also PyPy (http://pypy.org) which replaces the traditional C implementation of python with a *J*ust *I*n *C*ompiler (a *JIT*) that provides factors of a few speedups, and implements python 2.7.2. Unfortunately, it doesn't support numpy.

cython

cython lets you write python and speed up some critical part of the code. cython can also be used to bind C/C++ to python — a topic I'll return to later.

For example, I have a file **hello.pyx**:

```
def speak(string):
    print("Hello %s" % string)
```

```
>>> import pyximport; pyximport.install()
>>> import hello
>>> hello.speak("Robert")
Hello Robert
```

Trivia: it's "pyx" is because cython is a fork of an old project called pyrex.

```
$ wc -l ~/.pyxbld/temp.macosx-10.6-universal-2.7/pyrex/hello.c
  1269 /Users/rhl/.pyxbld/temp.macosx-10.6-universal-2.7/pyrex/hello.c
```

That innocent 2-line cython script generated 1269 lines of C.

cython website

http://cython.org

Building cython extensions

We just saw `pyximport` as a simple way to build cython extensions. An alternative is to use python's `distutils`; this requires a **setup.py** file along these lines:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("hello", ["hello.pyx"])]
)
```

which can be invoked as

```
$ python setup.py build_ext --inplace
```

cython timings

Stealing an example from the cython documentation (http://docs.cython.org/src/quickstart/cythonize.html),

```
def f(x):
    return x**2 - x

def integrate_f(a, b, N):
    s = 0
    dx = (b - a)/N
    for i in range(N):
        s += f(a + i*dx)
    return s*dx
```

We can time this with:

```
>>> import timeit; import hello
>>> t = timeit.Timer("import hello; hello.integrate_f(0, 10.0, 10**7)")
>>> print "%.2fs" % t.timeit(1)
5.46s
```

(I wrote `10**7` not `1e7` as I wanted an `int`)

If I use `pyximport` to import (*i.e.* convert to C, compile, and dynamically load) that file, the same test takes 3.85s

Using cython's cdef on variables

The first thing to do is to declare some variable's types with cdef:

```
def f1(double x):
    return x**2 - x

def integrate_f1(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b - a)/N
    for i in range(N):
        s += f1(a + i*dx)
    return s*dx
```

This runs in 1.07s

- Since the iterator variable i is typed with C semantics, the for-loop will be compiled to pure C code.
- Typing a, s and dx is important as they are involved in arithmetic within the for-loop
- Typing b and N makes less of a difference

Using cython's cdef on functions

Next we can change f to use the C calling sequence, *i.e.* passing C variables (in registers?) rather than converting to and from python objects:

```
cdef double f2(double x) except? -2:
    return x**2 - x

def integrate_f2(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b - a)/N
    for i in range(N):
        s += f2(a + i*dx)
    return s*dx
```

This runs in 0.05s

N.b. This version of f is of course only callable from cython code, not vanilla python.

The except? −2 bit says "If you return -2 check if an error occurred, and maybe throw an exception".

Summary of timings

- python

`f`	5.46s

- cython

`f`	3.85s
`f1`	1.07s
`f2`	0.05s

Clearly using `cython` can provide dramatic speedups for critical sections of code.

Using the C API, *CPython*

Python's C API

http://docs.python.org/extending/index.html#extending-index

http://docs.python.org/c-api

A simple example

Here's an example taken from `http://docs.python.org/extending/extending.html#a-simple-example`; it makes unix's `system` call available from python (for those who don't know about `os.system`).

A simple example

The first thing you need is the python header file:

```
#include <Python.h>
```

Next the wrapper for `system`:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    if (!PyArg_ParseTuple(args, "s", &command)) return NULL;

    const int sts = system(command);
    return Py_BuildValue("i", sts);
}
```

Now we need to tell python about that wrapper:

```
static PyMethodDef SpamMethods[] = {
    {"system",  spam_system, METH_VARARGS, "Execute a shell command."},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};

PyMODINIT_FUNC
initspam(void)
{
   (void)Py_InitModule("spam", SpamMethods);
}
```

We can use it as:

```
>>> import spam; spam.system("echo hello world")
```

Building C extensions

On my laptop, I type:

```
$ make spam.so
cc -o spam.os -c -fPIC -g -Wall   --std=c99 -I/usr/include/python2.7 spam.c
c++ -o spam.so -bundle -flat_namespace spam.os \
        -L/Library/Python/2.7/site-packages -lpython
```

Actually, I type `make spam` as my Makefile includes

```
.PHONY : spam
spam : spam.so
```

You can also use python's `distutils`; create a file **setup.py**:

```
from distutils.core import setup, Extension

module1 = Extension('spam',
                    sources = ['spam.c'])

setup (name = 'Spam',
       version = '1.0',
       description = 'This is a pedagogical wrapper for system',
       ext_modules = [module1])
```

and chant

```
$ python setup.py build_ext --inplace
```

(there is more to `distutils`; e.g. you can say `python setup.py build` if you don't want to build in your current directory).

Inventing your own python type

Here's an example from http://docs.python.org/extending/newtypes.html#the-basics:

I have a type Noddy

```
typedef struct {
    char *first;  /* first name */
    char *last;   /* last name */
    int number;
    char *name();
} Noddy;
```

After some magic (to be revealed), I can say:

```
>>> import noddy; rhl = noddy.Noddy("Robert", "Lupton", 323)
>>> rhl.first, rhl.last, rhl.number
('Robert', 'Lupton', 323)
>>> rhl.name()
'Robert Lupton'
```

Both C++ and python are object oriented languages, so this is just what the doctor ordered. But it isn't pretty. . . And it's more of a re-implementation than a wrapping.

Creating the Noddy type I

```
#include <Python.h>
#include <structmember.h>

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
          {
            Py_DECREF(self);
```

Creating the Noddy type II

```
            return NULL;
          }

        self->last = PyString_FromString("");
        if (self->last == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                      &first, &last,
                                      &self->number))
        return -1;

    if (first) {
```

Creating the Noddy type III

```
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }

    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }

    return 0;
}


static PyMemberDef Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
```

Creating the Noddy type IV

```
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}
```

Creating the Noddy type V

```
static PyMethodDef Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL}  /* Sentinel */
};

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /*ob_size*/
    "noddy.Noddy",             /*tp_name*/
    sizeof(Noddy),             /*tp_basicsize*/
    0,                         /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0,                         /*tp_print*/
    0,                         /*tp_getattr*/
    0,                         /*tp_setattr*/
    0,                         /*tp_compare*/
    0,                         /*tp_repr*/
    0,                         /*tp_as_number*/
    0,                         /*tp_as_sequence*/
    0,                         /*tp_as_mapping*/
    0,                         /*tp_hash */
    0,                         /*tp_call*/
    0,                         /*tp_str*/
    0,                         /*tp_getattro*/
    0,                         /*tp_setattro*/
    0,                         /*tp_as_buffer*/
```

Creating the Noddy type VI

```
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
    "Noddy objects",           /* tp_doc */
    0,                               /* tp_traverse */
    0,                               /* tp_clear */
    0,                               /* tp_richcompare */
    0,                               /* tp_weaklistoffset */
    0,                               /* tp_iter */
    0,                               /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
```

Creating the Noddy type VII

```
initnoddy(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
      return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
```

That’s even worse than mex (but it is doing quite a lot more).

Generating complete interfaces

Writing an interface for a given function isn't very hard, but it's a **lot** of work to support in the general (*i.e.* realistic) case. To generate an interface, you have to:

- Expose (`public`) member functions and data
- Transform between python and C++ data types
- Handle exceptions

ctypes

The python docs that I quoted in the *CPython* section say:

if your use case is calling C library functions or system calls, you should consider using the `ctypes` module rather than writing custom C code. Not only does `ctypes` let you write Python code to interface with C code, but it is more portable between implementations of Python than writing and compiling an extension module

`ctypes` is a builtin part of python (as of python 2.5)
`http://docs.python.org/library/ctypes.html#module-ctypes`
`ctypes` provides a way to access functions (and other symbols) in sharable object libraries. Despite this official status, my reading is that `cython` is slowly taking over from `ctypes` for wrapping scientific code.

Using `ctypes` to access *libc*

Loading a sharable library (in this case `libc`) is easy:

```
import ctypes
libc = ctypes.CDLL("libc.dylib")
```

In general you may not know the file to load, so you can use something like

```
import ctypes.util
libc = ctypes.util.find_library("c")
libc = ctypes.CDLL(libc)
```

It's easy to use too; our adventure with *CPython* can be written as:

```
libc.system("echo hello world")
```

ctypes' limitations

The `ctypes` is interface is **very** low level. For example,

```
strchr = libc.strchr
print strchr("abcdef", ord("d"))
```

prints `8059980` (the address of the string `"def"`)

The problem is that `ctypes` doesn't know `strchr`'s return type, but we can tell it:

```
strchr.restype = ctypes.c_char_p
print strchr("abcdef", ord("d"))
```

prints `'def'`.

Why should I have to type `ord("d")`, which converts `d` to an `char`? I don't:

```
strchr.argtypes = [ctypes.c_char_p, ctypes.c_char]
print strchr("abcdef", "d")
```

prints `'def'` and

```
try:
    strchr("abcdef", "def")
except Exception, e:
    print e
```

raises a `ctypes.ArgumentError` exception

ctypes' limitations

Without argtypes,

```
        print libc.strlen(1)
```

crashes python. Fortunately,

```
    libc.strlen.argtypes = [ctypes.c_char_p]
    try:
        print libc.strlen(1)
    except Exception, e:
        print e
```

raises the expected ctypes.ArgumentError exception.

Unfortunately,

```
    print libc.strlen("abc", "def")
```

quietly returns 3

Returning a `struct`

Returning to an earlier example, we had

```
struct {
    char *first;  /* first name */
    char *last;   /* last name */
    int number;
} Noddy;
```

To return one of these via `ctypes` I need to define (on the python side)

```
import ctypes

class Noddy(ctypes.Structure):
    _fields_ = [
        ('first', ctypes.c_char_p),
        ('last', ctypes.c_char_p),
        ('number', ctypes.c_int),
        ]

function.restype = ctypes.POINTER(Noddy)
```

I obviously need to keep this is sync with the C++ version.

I don't know how to return a `class` with virtual functions (and thus a `vtbl`); the `c` in `ctypes` really does seem to stand for C, not C++.

ctypes and numpy

Here's an example taken from the ctypes manual

```
import numpy
import ctypes

# Extract desired information from libfoo.so [or libfoo.dylib]
_foo = numpy.ctypeslib.load_library('libfoo', '/my/working/directory')
_foo.bar.restype = ctypes.c_int
_foo.bar.argtypes = [ctypes.POINTER(ctypes.c_double), ctypes.c_int]

def bar(x):
    """Wrapper to call C function 'bar' nicely from python"""
    return _foo.bar(x.ctypes.data_as(ctypes.POINTER(ctypes.c_double)), len(x))

x = numpy.random.randn(10)
n = bar(x)
```

Note that `numpy` arrays provide `.ctypes` to extract the information that `ctypes` needs; there's also *e.g.* `x.ctypes.shape[:3]`

cython and C/C++

cython can also be used to bind C/CPP to python.

For example,

```
// Return the greatest common divisor of a and b
int gcd(int a, int b);
```

(the details are left to your imagination). To use this from python using cython, we need an interface file simple.pyx:

```
cdef extern from "gcd.h":
    int c_gcd "gcd" (int a, int b)

def gcd(int a, int b):
    return c_gcd(a, b)
```

cython and C/C++

We next need to build the glue layer; the Makefile looks like:

```
simple.so: simple.pyx gcd.c gcd.h
        python setup.py build_ext --inplace
```

with

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

source_files = ["simple.pyx", "gcd.c"]

ext_modules = [Extension(
    name="simple",
    sources=source_files,
    # extra_objects=["fc.o"],  # if you compile fc.cpp separately
    extra_compile_args = "-std=c99".split(),
    # extra_link_args = "...".split()
    )]

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)
```

cython and C/C++

We can then triumphantly type

```
import simple

x = 52
y = 65
z = simple.gcd(x,y)
print "The gcd of %d and %d is %d" % (x,y,z)
```

and discover that

```
The gcd of 52 and 65 is 13
```

cython and numpy

```
# -*- python -*-
#
# Import the bits we need from C
#
cdef extern from "vector_ops.h":
    void c_scalar_multiply "scalar_multiply" (double alpha,
                                              double *x, double *z, long n)
    void c_vector_add "vector_add" (double *x, double *y, double *z, long n)
#
# Define wrapper functions to be used from python
#
import numpy as np
cimport numpy as np

def scalar_multiply(double alpha, np.ndarray[np.double_t,ndim=1] x):
    cdef long n = x.shape[0]
    cdef np.ndarray z = np.empty(n, dtype=np.double)
    c_scalar_multiply(alpha, <double *> x.data, <double *> z.data, n)
    return z

def vector_add(np.ndarray[np.double_t,ndim=1] x,
               np.ndarray[np.double_t,ndim=1] y):
    cdef long n = x.shape[0]
    cdef np.ndarray z = np.empty(n, dtype=np.double)
    c_vector_add(<double *> x.data, <double *> y.data, <double *> z.data, n)
    return z
```

cython and numpy

Building is very similar to the previous example; here's the diff for **setup.py**:

```
$ diff numpy/setup.py simple/setup.py
1d0
< import numpy
6c5
< source_files = ["vec_ops.pyx", "vector_ops.c"]
---
> source_files = ["simple.pyx", "gcd.c"]
9c8
<     name="vec_ops",
---
>     name="simple",
12d10
<     include_dirs = [numpy.get_include()],
```

cython and numpy

Using our new extension is as easy as:

```
import vec_ops, numpy as np

alpha, x = 2.1, np.array([1.,2.,3.])
y = vec_ops.scalar_multiply(alpha, x)
print alpha, "times", x, "is", y

z = vec_ops.vector_add(x, y)
print x, "plus", y, "is", z
```

which prints

```
2.1 times [ 1.  2.  3.] is [ 2.1  4.2  6.3]
[ 1.  2.  3.] plus [ 2.1  4.2  6.3] is [ 3.1  6.2  9.3]
```

Digression: compiler warnings

there are some compiler warnings from the machine-generated code; *e.g.*

```
vec_ops.c: In function '__pyx_pf_7vec_ops_2vector_add':
vec_ops.c:1363: warning: implicit conversion shortens 64-bit value into a 32-bit valu
vec_ops.c: In function '__Pyx_GetBuffer':
vec_ops.c:4649: warning: unused variable 'getbuffer_cobj'
```

Some of these are avoidable: in **vector_ops.h** we see:

```
void scalar_multiply(double alpha, const double *x, double *z, int n);
```

(should be long)

Digression: compiler warnings

Some can’t be avoided without delving into cython internals:

```
static int __Pyx_GetBuffer(PyObject *obj, Py_buffer *view, int flags)
{
    PyObject *getbuffer_cobj;

    if (PyObject_TypeCheck(obj, __pyx_ptype_5numpy_ndarray))
       return __pyx_pw_5numpy_7ndarray_1__getbuffer__(obj, view, flags);

#if PY_VERSION_HEX < 0x02060000
    if (obj->ob_type->tp_dict &&
        (getbuffer_cobj = PyMapping_GetItemString(obj->ob_type->tp_dict,
                                                  "__pyx_getbuffer"))) {
...
```

That would have been better written as:

```
static int __Pyx_GetBuffer(PyObject *obj, Py_buffer *view, int flags)
{
    if (PyObject_TypeCheck(obj, __pyx_ptype_5numpy_ndarray))
       return __pyx_pw_5numpy_7ndarray_1__getbuffer__(obj, view, flags);

#if PY_VERSION_HEX < 0x02060000
    PyObject *getbuffer_cobj;
    if (obj->ob_type->tp_dict &&
        (getbuffer_cobj = PyMapping_GetItemString(obj->ob_type->tp_dict,
                                                  "__pyx_getbuffer"))) {
...
```

Auto-generating the interface

You might be thinking, "Why do I have to do this? I could write a little python script to parse my .h files and write that `cython` file. . . "
"Hmm, that'd make a nice APC524 project. . . "
Don't even think about it. A C++ parser is a non-trivial task. There's a project derived from `g++` called `gccxml` but it's not been updated since `g++ 4.2`. The obvious solution is to use `clang++`, and solutions are beginning to appear. E.g. XDress *Caveat*: I haven't tried this one.

Generating complete interfaces

We've seen how to generate interfaces, but it's quite a lot of work (although less than was involved with `CPython`)

- Expose (`public`) member functions and data
- Transform between python and C++ data types
- Handle exceptions

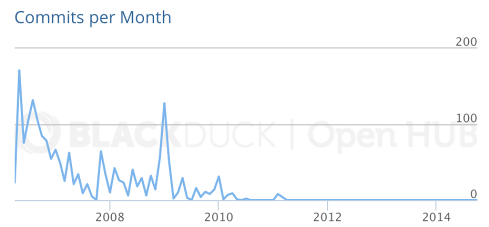
If you want to generate the interface automatically, you also need to

- Parse C++ header files
- Maintain the object schema (*i.e.* the information needed to regenerate the types; *cf.* gdb's `ptype`)
- Understand template instantiation
- ...

boost::python

`boost::python` provides a complete solution to the first set of problems, but some manual labour is involved. There's a project, `Py++`, that machine-generates at least part of the `boost::python` interface using `gccxml`, but I haven't tried it.

py++



. . . and it looks like I never will.

Actually that's unfair — they gave up on `svn` and moved to `git`. There's also `pyste` but it's very old.

boost::python

boost::python

http://www.boost.org/doc/libs/1_57_0/libs/python/doc/index.html

http://www.boost.org/doc/libs/1_57_0/libs/python/doc/tutorial/doc/html/index.html

There is also http://wiki.python.org/moin/boost.python

greet in boost::python

(This is based on http://www.boost.org/doc/libs/1_57_0/libs/python/doc/tutorial/doc/html/index.html)

Let's return to an old friend, `greet`. We first need `greet` itself:

```
std::string greet(const std::string &str="world") {
    return "hello " + str;
}
```

(we could `#include "greet.h"` instead). To bind this using `boost::python`, we need to include the proper header:

```
#include "boost/python.hpp"
```

and then

```
BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    def("greet", greet);

}
```

After building our extension `speak`, we can say

```
>>> import speak
>>> print speak.greet("class")
hello class
```

Building with boost::python

Building boost::python extensions is easy, if you happen to have boost installed; *e.g.* on the hats cluster:

```
$ g++ -o speak.os -c -fPIC -g -Wall \
        -I/usr/include/boost -I/usr/include/python2.4 speak.cc
$ g++ -o speak.so -shared speak.os -lpython2.4 -lboost_python
```

If you need to install boost on your own machine look at the *getting started* section of http://www.boost.org or go a-googling.

N.b. the boost documentation recommends building **speak.cc** with bjam; ignore this advice. With modernish versions of boost there's no need to lie on that bed of nails.

Overloaded functions

In

```
std::string greet(const std::string &str="world") {
    return "hello " + str;
}
```

we provided a default value for `str`, so this should work:

```
>>> import speak
>>> print speak.greet()
```

but in reality:

```
>>> print speak.greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Boost.Python.ArgumentError: Python argument types in
    speak.greet()
did not match C++ signature:
    greet(std::string)
```

(but at least that's a good error message)

Overloaded functions

`boost::python` can handle this, but it takes a bit more work:

```
BOOST_PYTHON_FUNCTION_OVERLOADS(greet_s_overloads, greet, 0, 1)

BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    def("greet", greet, greet_s_overloads());

}
```

The `BOOST_PYTHON_FUNCTION_OVERLOADS` says that `greet` takes from `0` to `1` arguments, and `boost::python` does the rest, generating the function `greet_s_overloads`.

Overloaded functions

If we also add

```
std::string greet(int i) {
    std::ostringstream ss;
    ss << "Hello " << i;
    return ss.str();
}
```

and

```
std::string greet(const std::string &str, int nrepeat) {
    std::ostringstream ss;
    ss << "Hello";
    for (int i = 0; i != nrepeat; ++i) ss << " " << str;
    return ss.str();
}
```

We need to tell `boost::python`:

```
std::string (*greet_i)(int) = &greet;
std::string (*greet_s)(const std::string &, int i) = &greet;

BOOST_PYTHON_FUNCTION_OVERLOADS(greet_s_overloads, greet, 0, 2)

BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    def("greet", greet_i);
    def("greet", greet_s, greet_s_overloads());

}
```

classes

Consider

```
class X {
public:
    void set(const std::string& msg) { msg_ = msg; }
    std::string greet() { return "hello " + msg_; }
private:
    std::string msg_;
};
```

The `boost::python` incantation is:

```
BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    class_<X>("X")
        .def("greet", &X::greet)
        .def("set",   &X::set)
    ;

}
```

after which:

```
>>> import speak
>>> X = speak.X()
>>> x.set("Clancy")
>>> print x.greet()
hello Clancy
```

Data members in `classes`

Consider

```
struct Y {
    int i;
};
```

We need to say what sort of access we need to `i`, *e.g.*

```
BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    class_<Y>("Y")
        .def_readwrite("i",   &Y::i)
    ;

}
```

and:

```
>>> import speak
>>> y = speak.Y()
>>> y.i
0
>>> y.i = 10
>>> y.i
10
```

boost::python and the *STL*

It's not easy to find in the documentation, but the *STL* is well supported nowadays:

```
#include "boost/python.hpp"

#include "boost/python/suite/indexing/vector_indexing_suite.hpp"

BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    class_<std::vector<double> >("vectorDouble")
        .def(vector_indexing_suite<std::vector<double> >())
        ;

}
```

```
>>> import speak
>>> v = speak.vectorDouble()
>>>  v.append(10); v.append(100)
>>> print len(v), [x for x in v]
2 [10.0, 100.0]
```

Functions that take `std::vector<double>`

If we define

```
#include <iostream>
#include <iterator>

...

template<typename T>
void print_vector(const std::vector<T>& v) {
    std::copy(v.begin(), v.end(),
              std::ostream_iterator<T>(std::cout, " "));
    std::cout << std::endl;
}
```

and add

```
    def("print_vector", print_vector<double>);
```

we can say:

```
>>> speak.print_vector(v)
10 100
```

boost::python and numpy

There was some work done long ago to integrate `boost::python` with `Numeric`, but that's ancient history now.
There is a third-party package for using `numpy` with `boost::python` in the `Boost.Sandbox`
https://svn.boost.org/svn/boost/sandbox/numpy/

My post-doc Jim Bosch writes:

It's a pain to install (grrr. . . bjam), but once you have done that it's very nice. Full disclosure: I wrote most of it.

Smart Pointers

Consider the C++ function:

```
std::vector<boost::shared_ptr<X> > makeVectorX()
{
    const int n = 2;
    std::vector<boost::shared_ptr<X> > v(n);

    v[0] = boost::shared_ptr<X>(new X); v[0]->set("Robert");
    v[1] = boost::shared_ptr<X>(new X); v[1]->set("Lupton");

    return v;
}
```

what does it take to bind this using `boost::python`?

boost::python and boost::shared_ptr

First we tell X about boost::shared_ptr<X>:

```
class_<X, boost::shared_ptr<X> >("X")
    .def("greet", &X::greet)
    .def("set",   &X::set)
;
```

Next, we need the std::vector:

```
class_<std::vector<boost::shared_ptr<X> > >("vectorX")
    .def(vector_indexing_suite<std::vector<boost::shared_ptr<X> >,
                               true>())
    ;
```

(note the magic true; it's the value of NoProxy) The actual declaration is simple:

```
def("makeVectorX", makeVectorX);
```

allowing me to say:

```
>>> import speak
>>> xx = speak.makeVectorX()
>>> print xx[0].greet()
hello Robert
>>> print xx[1].greet()
hello Lupton
```

Warning: this requires boost::shared_ptr and doesn't work with std::shared_ptr unless you have boost $\sim$ 1.52 (but boost's up to 1.57 so you're probably fine).

More complicated functions

If we have a function like

```
X& goo(Z &z, Y *y) {
    z.y = y;
    return z.x;
}
```

where

```
struct Z {
    X& getX() { return x; }
    X x;
    Y *y;
};
```

we have serious problems awaiting us if we're careless.

For example, after

```
>>> import speak
>>> y = speak.Y(); z = speak.Z(); x = speak.goo(z, y)
>>> del z
>>> x.set("rhl")
```

Does `x` still exist, or will the `x.set` call crash python? Alternatively, if I say

```
>>> y = speak.Y(); z = speak.Z(); x = speak.goo(z, y)
>>> del y
>>> z.y.i
```

is `z.y` accessing a dangling pointer?

Binding complicated functions

Fortunately, `boost::python` is picky about such things. To bind `goo`, I have to say

```
BOOST_PYTHON_MODULE(speak)
{
    using namespace boost::python;

    def("goo", goo,
        return_internal_reference<1, with_custodian_and_ward<1, 2> >());

}
```

I.e.

- `return_internal_reference<1>`: Tie lifetime of first argument to that of the result
- `with_custodian_and_ward<1, 2>`: Tie lifetimes of the first argument to the second

And, seeing as you were going to ask about `z.y`:

```
class_<Z>("Z")
  .def("getX",       &Z::getX, return_internal_reference<1>())
  .add_property("y",
   make_getter(&Z::y, return_value_policy<reference_existing_object>()),
   make_setter(&Z::y, return_value_policy<reference_existing_object>()))
  ;
```

What is `swig`?

The *S*implified *W*rapper and *I*nterface *G*enerator, (`swig`; `http://www.swig.org`) is a way of **automatically** generating code that interfaces C or C++ to { C#, Guile, Java, Lua, Modula 3, Mzscheme, Ocaml, Octave, Perl, PHP4, PHP5, Pike, Python, R (aka GNU S), Ruby, Lisp S-Expressions, Tcl, Common Lisp / UFFI, XML } that handles all of these concerns, and more.

The `swig` online documentation

`http://www.swig.org/Doc3.0/index.html`

The latest version as of 2014-06-04 is 3.0.2

N.b. there are still web pages that claim that an ancient release, 1.1 is the current version of `swig`; there are therefore pages that claim that `swig` cannot handle C++, is unsupported, etc. etc.

Hello World

Consider a couple of source files, **hello.h**:

```
#if !defined(HELLO_H)
#define HELLO_H 1

#include <string>

void speak(std::string const& str);
#endif
```

and **hello.c**:

```
#include <iostream>
#include "hello.h"

void speak(std::string const& str)
{
    std::cout << "Robert says: " << str << std::endl;
}
```

Hello World

With this `swig` interface file, **hello.i**:

```
%module hello

%{
#include "hello.h"
%}

%include "hello.h"
```

Hello World

and after running make:

```
$ swig -o hello_wrap.cc -c++ -python hello.i
$ g++ -o hello.os -c -fPIC -I/usr/include/python2.7 hello.cc
$ g++ -o hello_wrap.os -c -fPIC -I/usr/include/python2.7 hello_wrap.cc
$ g++ -o _hello.so -bundle -flat_namespace hello_wrap.os hello.os \
        -L/Library/Python/2.7/site-packages -lpython
```

i.e.

- Run `swig` to generate **hello_wrap.cc** (and also, as we shall see, **hello.py**)
- Compile **hello.cc** and **hello_wrap.cc** to create object files ***.os** with the usual compile-python boilerplate.
- Build a loadable module **_hello.so** using the usual os/x dylib boilerplate.

I can start python and import my new module

```
>>> import hello
>>> hello.speak("hello world")
Robert says: hello world
```

How does swig earn its keep?

What did swig actually do? It wrote two files, **hello_wrap.cc** (which we just compiled) and **hello.py**. When we started python there were two possible files we could import: **_hello.so** and **hello.py**. The former is created from **hello_wrap.cc** — a file that you really don't want to examine.

_wrap.cc files

You may not want to, but... A small and simplified part of **hello_wrap.cc**'s 4k lines reads:

```
SWIGINTERN PyObject *_wrap_speak(PyObject *SWIGUNUSEDPARM(self),
                                 PyObject *args) {
  std::string *arg1 = 0 ;
  int res1 = SWIG_OLDOBJ ;
  PyObject * obj0 = 0 ;

  if (!PyArg_ParseTuple(args,(char *)"O:speak",&obj0)) SWIG_fail;

  std::string *ptr = (std::string *)0;
  res1 = SWIG_AsPtr_std_string(obj0, &ptr);
  if (!SWIG_IsOK(res1)) {
    SWIG_exception_fail(SWIG_ArgError(res1), "in method 'speak' "
                        "argument 1 of type 'std::string const &'");
  }
  arg1 = ptr;

  speak((std::string const &)*arg1);

  if (SWIG_IsNewObj(res1)) delete arg1;
  return SWIG_Py_Void();
fail:
  if (SWIG_IsNewObj(res1)) delete arg1;
  return NULL;
}
```

Explanations

What's going on? This is within an `extern "C"` block, so it defines a callable function `_wrap_speak()` that checks the argument type and calls `speak()`.

Let's read some more of **hello_wrap.cc**:

```
static PyMethodDef SwigMethods[] = {
 { (char *)"SWIG_PyInstanceMethod_New",
          (PyCFunction)SWIG_PyInstanceMethod_New, METH_O, NULL},
  (char *)"speak", _wrap_speak, METH_VARARGS, NULL,
 { NULL, NULL, 0, NULL }
};
...
void
SWIG_init(void) {
  PyObject *m, *d;

  m = Py_InitModule((char *) SWIG_name, SwigMethods);
  d = PyModule_GetDict(m);

  SWIG_InitializeModule(0);
  SWIG_InstallConstants(d,swig_const_table);
}
```

I.e. we define a module `_hello` that knows the command `speak`.

Syntactic sugar

However, in python we said `hello.speak("hello world")`, not `_hello.speak("hello world")`.

Enter **hello.py**. In this case it's more-or-less trivial:

```
import _hello
...
speak = _hello.speak
```

swig's %inline directive

I could have avoided the extra files with:

```
// -*- c++ -*-
%module hello_inline
%include "std_string.i"

%{
#include <iostream>
%}

%inline %{
void speak(std::string const& str)
{
    std::cout << "Robert says: " << str << std::endl;
}
%}
```

C and swig

OK; so that was C++ but I could have just as well have used C and printf. A prime motivation for pairing C++ and python is that both are OO languages.

For example, using C I have to tell swig things like:

```
%newobject create_foo;
%delobject destroy_foo;

Foo *create_foo();
void destroy_foo(Foo *foo);
```

to handle creation/deletion of objects. As C++ possesses gnostic wisdom of life and death (*i.e.* about con- and de-structors) swig can and does handle such matters for you.

C++ and swig

```
%module classes
%include "std_string.i"

%{
#include <iostream>
#include <string>
%}

%inline %{
class Foo {
public:
    Foo() { std::cout << "Hail, fair morning" << std::endl; }
    ~Foo() {
       std::cout << "It is a far, far better thing that I do now..."
                 << std::endl;
    }
};
%}
```

```
>>> import classes
>>> x = classes.Foo()
Hail, fair morning
>>> del x
It is a far, far better thing that I do now...
>>> def tmp(): x = classes.Foo()
>>> tmp()
Hail, fair morning
It is a far, far better thing that I do now...
>>>
```

(note that the destructor fired when x went out of scope)

Proxy classes

In this case the swig-generated **classes.py** is more complicated. It defines a python “proxy” class Foo. For example, __init__ calls new_Foo in **_classes.so**’s defined python interface; which calls _wrap_new_Foo; which calls the constructor Foo():

```
class Foo(_object):
    __swig_setmethods__ = {}
    __setattr__ = \
       lambda self, name, value: _swig_setattr(self, Foo, name, value)
    __swig_getmethods__ = {}
    __getattr__ = lambda self, name: _swig_getattr(self, Foo, name)
    __repr__ = _swig_repr
    def __init__(self):
        this = _classes.new_Foo()
        try: self.this.append(this)
        except: self.this = this
    __swig_destroy__ = _classes.delete_Foo
    __del__ = lambda self : None;
Foo_swigregister = _classes.Foo_swigregister
Foo_swigregister(Foo)
```

More complicated classes

That was fun. Now for a slightly more interesting class:

```
class Goo {
public:
    Goo() : i_(0), s_("") {}
    Goo(int i) : i_(i), s_("") {} Goo(std::string const& s) : i_(0), s_(s) {}
    int getI() const { return i_; }
    std::string getS() const { return s_; }
private:
    int i_;
    std::string s_;
};
```

```
>>> import classes
>>> g = classes.Goo()
>>> print "%d \"%s\"" % (g.getI(), g.getS())
0 ""
>>> g = classes.Goo(12);  print "%d \"%s\"" % (g.getI(), g.getS())
12 ""
>>> g = classes.Goo("rhl");  print "%d \"%s\"" % (g.getI(), g.getS())
0 "rhl"
```

In this case, `Goo`'s proxy class has entries:

```
    def getI(self): return _classes.Goo_getI(self)
    def getS(self): return _classes.Goo_getS(self)
```

What if I don't like accessor functions?

What about `struct`-style coding?

```
class Goo {
public:
    Goo() : i_(0), s_(""), x(-1) {}
    Goo(int i) : i_(i), s_(""), x(-1) {}
    Goo(std::string const& s) : i_(0), s_(s), x(-1) {}
...
    double x;
};
```

```
>>> g.x
-1.0
>>> g.x = 10
>>> g.x
10.0
```

good

```
>>> g.y = 1000
>>> g.y
1000
```

Not so good (`y` isn't a member of `Goo`).

Preventing swig from extending classes

The problem is that Foo is a python proxy class, and python permits anyone to add members such as y to a class.
Fortunately, there's a solution:

```
%module goo
%{
#include <iostream>
#include "Goo.h"

%}

%pythonnondynamic;

%include "Goo.h"
```

and now:

```
>>> g = Goo(1)
>>> g.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "goo.py", line 356, in <lambda>
    __getattr__ = lambda self, name: _swig_getattr(self, Goo, name)
  File "goo.py", line 54, in _swig_getattr
    raise AttributeError(name)
AttributeError: y
```

that's better.

Extending the interface in python

You can also add code to the python interface (it's also possible to add to the C++ interface in python using `swig` *directors*, but I've never tried). It's not surprising that you can extend the python layer when you recall the existence of proxy classes.

Extending the interface in python

```
// -*- c++ -*-
%module goo

%{
#include <iostream>
#include "Goo.h"
%}
%pythonnondynamic;
%include "std_iostream.i"

%include "Goo.h"

%extend Goo {
   void printMe(std::ostream& os) {
      os << $self->getI() << " \"" << $self->getS() << "\"" << std::endl;
   }
   %pythoncode %{
   def __str__(self):
      return "%d \"%s\"" % (self.getI(), self.getS())
   %}
}

>>> goo.Goo(12).printMe(goo.cout)
12 ""
>>> print goo.Goo('xxx')
0 "xxx"
```

You can easily imagine what the proxy class looks like.

swig v. the *STL*

One reason to use C++ is the Standard Template Library.

```
// -*- c++ -*-
%module vector
%include "std_vector.i"

%{
#include <iostream>
#include <string>
#include <vector>

#include "Goo.h"
%}

%include "Goo.h"

%template(vectorGoo) std::vector<Goo>;
```

```
>>> import vector
>>> v = vector.vectorGoo()
>>> v.push_back(vector.Goo(0))
>>> v.append(vector.Goo(1))
>>> len(v)
2
>>> v[1].getI()
1
```

Why do I care?

So what? python already has vector-like lists. Consider:

```
// -*- c++ -*-
%module vector2
%include "std_vector.i"

%{
#include <iostream>
#include <string>
#include <vector>
#include "Goo.h"
%}

%import "goo.i"

%template(vectorGoo) std::vector<Goo>;

%inline %{
   std::vector<Goo> *makeVectorGoo() {
      return new std::vector<Goo>;
   }

    void printGV(std::vector<Goo> const& gv) {
        for (auto ptr = gv.begin(); ptr != gv.end(); ++ptr) {
            std::cout << ptr->getI() << std::endl;
        }
    }
%}
```

Using python's `list` with `std::vector` classes

```
>>> import vector2 as vector; import goo
>>> v = vector.makeVectorGoo()          # equivalent to vector.VectorGoo()
>>> v.push_back(goo.Goo(0)); v.push_back(goo.Goo(1))
>>> vector.printGV(v)
0
1
>>> vv = [goo.Goo(10), goo.Goo(20)]
>>> vector.printGV(vv)
10
20
```

That's pretty nice; we passed a python list to a C++ function expecting a `std::vector<>`.

swig woes

If you put the %template line *after* printGV, you'd get:

```
Traceback (most recent call last):
  File "./vector2_demo.py", line 13, in <module>
    main()
  File "./vector2_demo.py", line 10, in main
    vector.printGV(vv)
TypeError: in method 'printGV', argument 1 of type
                     'std::vector< Goo,std::allocator< Goo > > const &'
```

swig needed to be told how to handle std::vector<Goo> before it saw the declaration

If you omit the %template line and try the push_back you get:

```
Traceback (most recent call last):
  File "./vector2_demo.py", line 13, in <module>
    main()
  File "./vector2_demo.py", line 6, in main
    v.push_back(goo.Goo(0))
AttributeError: 'SwigPyObject' object has no attribute 'push_back'
```

whereas v.append(goo.Goo(666)) produces:

```
Traceback (most recent call last):
  File "./vector2_demo.py", line 15, in <module>
    main()
  File "./vector2_demo.py", line 7, in main
    v.append(goo.Goo(666))
SystemError: error return without exception set
```

Two ways of saying, "I don't know about std::vector<Goo>"

swig *typemaps*

Much of swig is built around *typemaps*. For example, when swig is wrapping python the following maps are active

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}

/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

These are essentially macros that generate C++ in the **_wrap.cc** file.

To generate perl bindings, we use a different typemap:

```
/* Convert from Perl --> C */
%typemap(in) int {
   $1 = SvIV($input);
}
```

swig *typemaps*

For example, given

```
int foo(int x, int y);
```

swig writes something like:

```
PyObject *wrap_foo(PyObject *self, PyObject *args) {
   int arg1, arg2;
   int result;
   PyObject *obj1, *obj2;
   PyObject *resultobj;

   if (!PyArg_ParseTuple("OO:foo", &obj1, &obj2)) return NULL;

   arg1 = PyInt_AsLong(obj1);
   arg2 = PyInt_AsLong(obj2);

   result = foo(arg1);

   resultobj = PyInt_FromLong(result);

   return resultobj;
}
```

You can write your own typemaps if you have special needs, but generally speaking the casual swig user doesn't need to learn these arcana.

applying typemaps

I can specify that a typemap only be applied to an argument with a particular name:

```
%typemap(in) int positive {
    $1 = PyInt_AsLong($input);
    if ($1 <= 0) {
       SWIG_exception(SWIG_ValueError,"Expected positive value.");
    }
}
```

(It would be better to use a `%typemap(check)` typemap).
Unfortunately, my routine looks like:

```
int *newArray(int n);
```

How do I check that `n` is positive? The solution is to ask `swig` to `%apply` my `int positive` map to `n`:

```
%apply int positive { int n };
```

and now `newArray`'s argument is checked.

swig and numpy

swig isn't omniscient. If you write a function with prototype

```
double rms(double *x, int n);
```

it *probably* returns the root-mean-square of a vector x of length n. But it might be:

```
double rms(double *x, int n)
{
   *x = 2*n;

   return 666.0;
}
```

The numpy project supports a set of swig typemaps to help you write your **.i** files:

numpy.i

http://docs.scipy.org/doc/numpy/reference/swig.interface-file.html

whence I stole these examples of numpy bindings

numpy.i

We can use **numpy.i** to solve our `rms` dilemma:

```
%{
#define SWIG_FILE_WITH_INIT
%}

%include "numpy.i"

%init %{
import_array();
%}

%apply (double* IN_ARRAY1, int DIM1) {(double *x, int n)};
%inline %{
    double rms(double *x, int n);
%}
```

(yes, typemaps can handle sets of arguments) You should

```
%clear (double *x, int n);
```

immediately after the `rms` prototype to avoid accidently applying it to some other function.

N.b. Within a compiled Python module, `import_array()` must be get called exactly once; read `http://docs.scipy.org/doc/numpy/reference/swig.interface-file.html#using-numpy-i` carefully if you want to get it right. Which you do.

numpy.i typedefs

There are lots of **numpy.i** typedefs. *E.g.* for input arrays

```
1D:
        * ( DATA_TYPE IN_ARRAY1[ANY] )
        * ( DATA_TYPE* IN_ARRAY1, int DIM1 )
        * ( int DIM1, DATA_TYPE* IN_ARRAY1 )

2D:
        * ( DATA_TYPE IN_ARRAY2[ANY][ANY] )
        * ( DATA_TYPE* IN_ARRAY2, int DIM1, int DIM2 )
        * ( int DIM1, int DIM2, DATA_TYPE* IN_ARRAY2 )
        * ( DATA_TYPE* IN_FARRAY2, int DIM1, int DIM2 )
        * ( int DIM1, int DIM2, DATA_TYPE* IN_FARRAY2 )

3D:
        ...
```

where `DATA_TYPE` is one of `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, and `double`.

numpy.i typedefs

There are typedefs for a number of uses for arrays:

input `(DATA_TYPE IN_ARRAY1[ANY])`

in-place modification: `(DATA_TYPE INPLACE_ARRAY1[ANY])`

output: `(DATA_TYPE ARGOUT_ARRAY1[ANY])`

output for when the C++ owns the data: `(DATA_TYPE** ARGOUTVIEW_ARRAY1, DIM_TYPE* DIM1)`

For `IN_ARRAY` (where the data's not changed) the input may be any python sequence or a `numpy` array; in all other case in must be a `numpy` array.

Smart Pointers

What about smart pointers?

Chant some incantations to the `swig` input (".i") files:

```
%{
#include <memory>
%}
...
%include "std_shared_ptr.i"
...
%shared_ptr(Goo);                       // must come before definition of Goo

%include "Goo.h"

%template(vectorGoo) std::vector<std::shared_ptr<Goo> >;
```

Then in python

```
>>> import vector3 as vector; import goo; Goo=goo.Goo
>>> v = vector.vectorGoo(); v.push_back(Goo(0)); v.push_back(Goo(1))
>>> v[0]
<goo.Goo; proxy of <Swig Object of type std::shared_ptr< Goo > *' at 0x100491f90> >
>>> print v[0]
0 ""
```

Note that nothing's changed for the user.

N.b. I assumed a C++ 11 compiler. If you need to use boost's `shared_ptr`, replace `std_shared_ptr.i` by `boost_shared_ptr.i` and include `boost/shared_ptr.hpp` instead of `<memory>`.

`boost::python` *v.* `cython` *v.* `swig`

There appear to be three viable technologies to wrap C++ and python: `boost::python`, `cython`, and `swig`. Which should you use?

In simple case `swig` is less work; you tell it about your **.h** file and it generates the full interface. In many complicated cases this is still true, but when it fails, it can be very confusing.

With `cython` you have to say what you want to wrap, and `cython` generates the interface

You have to tell `boost::python` exactly what you want to expose, and how to do so — but `boost::python` then does exactly what you told it to do

Ultimately the decision is a matter of taste. I use `swig`. Clancy likes `cython`. My post-doc Jim swears by `boost::python`.