Calling C/C++ from Python

Alexey Svyatkovskiy, based on the original slides by Robert Lupton

22 November 2016

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)
```

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.

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.

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

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

There's also PyPy (`http://pypy.org`) which replaces the traditional C implementation of python with a *J*ust *I*n *T*ime compiler (*JIT*) that provides significant speedups.

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

There's also PyPy (http://pypy.org) which replaces the traditional C implementation of python with a *J*ust *I*n *T*ime compiler (*JIT*) that provides significant speedups.

cython

cython lets you write python and speed up some critical part of the code.

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.

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(name):
    print("Hello %s" % name)
Hello Alexey
```

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(name):
    print("Hello %s" % name)
Hello Alexey
```

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

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(name):
    print("Hello %s" % name)
Hello Alexey
```

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

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

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(name):
    print("Hello %s" % name)
Hello Alexey
```

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

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

```
$ wc -l ~/.pyxbld/temp.linux-x86_64-2.7/pyrex/hello.c
1792 /home/alexeys/.pyxbld/temp.linux-x86_64-2.7/pyrex/hello.c
```

That innocent 2-line cython script generated 1792 lines of C.

cython website

http://cython.org

Building cython extensions

We just saw pyximport as a simple way to build cython extensions.

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"])]
)
```

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
```

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
```

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
```

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
```

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:

```
def 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
```

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:

```
def 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

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:

```
def 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.

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:

```
def 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.

ctypes

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

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

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.

ctypes

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 C standard library

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

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

Using ctypes to access C standard library

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)
```

Using ctypes to access C standard library

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:

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

Defining custom types with ctypes

The ctypes module of the Python standard library provides definitions of fundamental data types that can be passed to C programs. For example a C double type:

```
import ctypes as C
x = C.c_double(2.71828)
```

Defining custom types with ctypes

The ctypes module of the Python standard library provides definitions of fundamental data types that can be passed to C programs. For example a C double type:

```
import ctypes as C
x = C.c_double(2.71828)
```

And here is the pointer type to double:

```
xp = C.POINTER(C.c_double)();
xp.contents = x
```

Defining custom types with ctypes

The ctypes module of the Python standard library provides definitions of fundamental data types that can be passed to C programs. For example a C double type:

```
import ctypes as C
x = C.c_double(2.71828)
```

And here is the pointer type to double:

```
xp = C.POINTER(C.c_double)();
xp.contents = x
```

Which gives:

```
>>> print(xp)
>>> <__main__.LP_c_double object at 0x7fb63427c9e0>
>>> print(x)
>>> c_double(2.71828)
```

Defining custom types with ctypes

The ctypes module of the Python standard library provides definitions of fundamental data types that can be passed to C programs. For example a C double type:

```
import ctypes as C
x = C.c_double(2.71828)
```

And here is the pointer type to double:

```
xp = C.POINTER(C.c_double)();
xp.contents = x
```

Which gives:

```
>>> print(xp)
>>> <__main__.LP_c_double object at 0x7fb63427c9e0>
>>> print(x)
>>> c_double(2.71828)
```

Array types can be created by multiplying a ctype by a positive integer, e.g.:

```
ylist = [1.,2.3,4.,5.]
n = len(ylist)
y = (C.c_double*n)()
y[:] = ylist
#or simply
y = (C.c_double*n)(*ylist)
```

ctypes: a complete example I

Let us start by writing some C code. The dot product of two vectors for instance:

```
double dot_product(double v[], double u[], int n)
{
    double result = 0.0;
    for (int i = 0; i < n; i++)
        result += v[i]*u[i];
    return result;
}
```

ctypes: a complete example I

Let us start by writing some C code. The dot product of two vectors for instance:

```
double dot_product(double v[], double u[], int n)
{
    double result = 0.0;
    for (int i = 0; i < n; i++)
        result += v[i]*u[i];
    return result;
}
```

Next we compile it, and build a shared object:

```
gcc -c -Wall -Werror -fpic my_dot.c
gcc -shared -o my_dot.so my_dot.o
```

ctypes: a complete example II

We have already seen that the ctypes module has a utility subpackage to assist in locating a dynamically-loaded library:

```
import ctypes.util
C.util.find_library('my_dot')
myDL = C.CDLL('./my_dot.so')
```

ctypes: a complete example II

We have already seen that the ctypes module has a utility subpackage to assist in locating a dynamically-loaded library:

```
import ctypes.util
C.util.find_library('my_dot')
myDL = C.CDLL('./my_dot.so')
```

Here is a full example:

```
from ctypes import CDLL, c_int, c_double
mydot = CDLL('./my_dot.so').dot_product
def dot(vec1, vec2): # vec1, vec2 are Python lists
    n = len(vec1)
    mydot.restype = c_double
    return mydot((c_double*n)(*vec1), (c_double*n)(*vec2), c_int(n))

vec1 = [x for x in range(1000000)]
vec2 = [x for x in range(1000000)]
```

ctypes: a complete example II

We have already seen that the ctypes module has a utility subpackage to assist in locating a dynamically-loaded library:

```
import ctypes.util
C.util.find_library('my_dot')
myDL = C.CDLL('./my_dot.so')
```

Here is a full example:

```
from ctypes import CDLL, c_int, c_double
mydot = CDLL('./my_dot.so').dot_product
def dot(vec1, vec2): # vec1, vec2 are Python lists
    n = len(vec1)
    mydot.restype = c_double
    return mydot((c_double*n)(*vec1), (c_double*n)(*vec2), c_int(n))

vec1 = [x for x in range(1000000)]
vec2 = [x for x in range(1000000)]
```

Warning: if you use the extension .so for the name of a file, do not make its stem the same as a .py file in the same directory, e.g., do not have both a funcs.py and a funcs.so.

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)),

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.

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).

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
```

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)
```

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
    void c_vector_add "vector_add" (double *x, double *y, double *z, lon
#
# 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
    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
```

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
vec_ops.c: In function '__Pyx_GetBuffer':
vec_ops.c:4649: warning: unused variable 'getbuffer_cobj'
```

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
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);
```

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
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"))) {
...
```

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"))) {
...
```

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

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.

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).

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"
```

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);
}
```

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, *e.g.* on Della cluster:

```
$ module load boost/1.55.0
$ g++ -o speak.os -c -fPIC -g -Wall \
        -I/usr/include/boost -I/usr/include/python2.7 speak.cc
$ g++ -o speak.so -shared speak.os -lpython2.7 -lboost_python
```

Building with boost::python

Building boost::python extensions is easy, *e.g.* on Della cluster:

```
$ module load boost/1.55.0
$ g++ -o speak.os -c -fPIC -g -Wall \
        -I/usr/include/boost -I/usr/include/python2.7 speak.cc
$ g++ -o speak.so -shared speak.os -lpython2.7 -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.

Building with boost::python

Building boost::python extensions is easy, *e.g.* on Della cluster:

```
$ module load boost/1.55.0
$ g++ -o speak.os -c -fPIC -g -Wall \
        -I/usr/include/boost -I/usr/include/python2.7 speak.cc
$ g++ -o speak.so -shared speak.os -lpython2.7 -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()
```

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());
}
```

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

td::ostringstream ss; ss « "Hello " « i; return ss.str(); stlistinganguage=C,label= ,caption= ,numbers=nonestlistingtd::ostringstream ss; ss « "Hello"; for (int i = 0; i != nrepeat; ++i) ss « " " « str; return ss.str(); stlistinganguage=Python,label= ,caption= ,numbers=nonestlisting
BOOST$_PYTHON_FUNCTION_OVERLOADS(greet_{so}verloads, greet, 0, 2)BOOST_PYTHON_MO$

classes

Consider

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

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);
}
```

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("Alexey")
>>> print x.greet()
hello Alexey
```

Data members in classes

Consider

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

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);
}
```

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*

STL is well support in boost::python as well:

```
#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> >());
}
```

boost::python and the *STL*

STL is well support in boost::python as well:

```
#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;
}
```

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>);
```

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
```

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 2016-06-12 is 3.0.10

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 << "Alexey 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
```

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.

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

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")
Alexey 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. . .

_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().

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").

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 << "Alexey 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.

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.

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;
    }
};
%}
```

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...
>>>
```

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():

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_;
};
```

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"
```

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)
```

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())
   %}
}
```

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"
```

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>;
```

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.

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
```

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 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

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'
```

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
```

swig *typemaps*

Much of swig is built around *typemaps*.

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.

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 *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;
}
```

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.");
    }
}
```

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).

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?

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.

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?

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.

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.

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

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

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.

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 prefer cython.

Using the C API, *CPython*

Python's C API

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

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