

Contents

- [Wrapping C/C++ for Python](#)
 - [Manual wrapping](#)
 - [Wrapping Python code with SWIG](#)
 - [Wrapping C code with pyrex](#)
 - [ctypes](#)
 - [SIP](#)
 - [Boost.Python](#)
 - [Recommendations](#)
 - [One or two more notes on wrapping](#)
- [Packages for Multiprocessing](#)
 - [threading](#)
 - [parallelpython](#)
 - [Rpyc](#)
 - [pyMPI](#)
 - [multitask](#)
- [Useful Packages](#)
 - [subprocess](#)
 - [rpy](#)
 - [matplotlib](#)
- [Idiomatic Python Take 3: new-style classes](#)
 - [Managed attributes](#)
 - [Descriptors](#)
- [GUI Gossip](#)
- [Python 3.0](#)

Wrapping C/C++ for Python

There are a number of options if you want to wrap existing C or C++ functionality in Python.

Manual wrapping

If you have a relatively small amount of C/C++ code to wrap, you can do it by hand. The [Extending and Embedding](#) section of the docs is a pretty good reference.

When I write wrappers for C and C++ code, I usually provide a procedural interface to the code and then use Python to construct an object-oriented interface. I do things this way for two reasons: first, exposing C++ objects to Python is a pain; and second, I prefer writing higher-level structures in Python to writing them in C++.

Let's take a look at a basic wrapper: we have a function 'hello' in a file 'hello.c'. 'hello' is defined like so:

```
char * hello(char * what)
```

To wrap this manually, we need to do the following.

First, write a Python-callable function that takes in a string and returns a string.

```
static PyObject * hello_wrapper(PyObject * self, PyObject * args)
```

```

{
  char * input;
  char * result;
  PyObject * ret;

  // parse arguments
  if (!PyArg_ParseTuple(args, "s", &input)) {
    return NULL;
  }

  // run the actual function
  result = hello(input);

  // build the resulting string into a Python object.
  ret = PyString_FromString(result);
  free(result);

  return ret;
}

```

Second, register this function within a module's symbol table (all Python functions live in a module, even if they're actually C functions!)

```

static PyMethodDef HelloMethods[] = {
  { "hello", hello_wrapper, METH_VARARGS, "Say hello" },
  { NULL, NULL, 0, NULL }
};

```

Third, write an init function for the module (all extension modules require an init function).

```

DL_EXPORT(void) inithello(void)
{
  Py_InitModule("hello", HelloMethods);
}

```

Fourth, write a setup.py script:

```

from distutils.core import setup, Extension

# the c++ extension module
extension_mod = Extension("hello", ["hellomodule.c", "hello.c"])

setup(name = "hello", ext_modules=[extension_mod])

```

There are two aspects of this code that are worth discussing, even at this simple level.

First, error handling: note the `PyArg_ParseTuple` call. That call is what tells Python that the 'hello' wrapper function takes precisely one argument, a string ("s" means "string"; "ss" would mean "two strings"; "si" would mean "string and integer"). The convention in the C API to Python is that a `NULL` return from a function that returns `PyObject*` indicates an error has occurred; in this case, the error information is set within `PyArg_ParseTuple` and we're just passing the error on up the stack by returning `NULL`.

Second, references. Python works on a system of reference counting: each time a function "takes ownership" of an object (by, for example, assigning it to a list, or a dictionary) it increments that object's reference count by one using `Py_INCREF`. When the object is removed from use in that particular place (e.g. removed from the list or dictionary), the reference count is decremented with `Py_DECREF`. When the reference count reaches 0, Python knows that this object is not being used by anything and can be freed (it may not be freed immediately, however).

Why does this matter? Well, we're creating a `PyObject` in this code, with `PyString_FromString`. Do we need to `INCRF` it? To find out, go take a look at the documentation for `PyString_FromString`:

<http://docs.python.org/api/stringObjects.html#12h-461>

See where it says "New reference"? That means it's handing back an object with a reference count of 1, and that's what we want. If it had said "Borrowed reference", then we would need to INCREMENT the object before returning it, to indicate that we were using it as a persistent object (in this case, our return value!)

If you wanted to return None, by the way, you can use Py_None. Remember to INCREMENT it!

So that's a brief introduction to how you wrap things by hand.

As you might guess, however, there are a number of projects devoted to automatically wrapping code. Here's a brief introduction to some of them. .. talk about testing c code with python

Wrapping Python code with SWIG

SWIG stands for "Simple Wrapper Interface Generator", and it is capable of wrapping C in a large variety of languages. To quote, "SWIG is used with different types of languages including common scripting languages such as Perl, PHP, Python, Tcl, Ruby and PHP. The list of supported languages also includes non-scripting languages such as C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), Java, Modula-3 and OCAML. Also several interpreted and compiled Scheme implementations (Guile, MzScheme, Chicken) are supported."

Whew.

But we only care about Python for now!

SWIG is essentially a macro language that groks C code and can spit out wrapper code for your language of choice.

You'll need three things for a SWIG wrapping of our 'hello' program. First, a Makefile:

```
all:
    swig -python -c++ -o _swigdemo_module.cc swigdemo.i
    python setup.py build_ext --inplace
```

This shows the steps we need to run: first, run SWIG to generate the C code extension; then run `setup.py build` to actually build it.

Second, we need a SWIG wrapper file, 'swigdemo.i'. In this case, it can be pretty simple:

```
%module swigdemo

%{
#include <stdlib.h>
#include "hello.h"
%}

#include "hello.h"
```

A few things to note: the `%module` specifies the name of the module to be generated from this wrapper file. The code between the `%{ %}` is placed, verbatim, in the C output file; in this case it just includes two header files. And, finally, the last line, `%include`, just says "build your interface against the declarations in this header file".

OK, and third, we will need a `setup.py`. This is virtually identical to the `setup.py` we wrote for the manual wrapping:

```

from distutils.core import setup, Extension

extension_mod = Extension("_swigdemo", ["_swigdemo_module.cc", "hello.c"])

setup(name = "swigdemo", ext_modules=[extension_mod])

```

Now, when we run 'make', swig will generate the `_swigdemo_module.cc` file, as well as a `'swigdemo.py'` file; then, `setup.py` will compile the two C files together into a single shared library, `'_swigdemo'`, which is imported by `swigdemo.py`; then the user can just `'import swigdemo'` and have direct access to everything in the wrapped module.

Note that swig can wrap most simple types "out of the box". It's only when you get into your own types that you will have to worry about providing what are called "typemaps"; I can show you some examples.

I've also heard (from someone in the class) that SWIG is essentially not supported any more, so buyer beware. (I will also say that SWIG is pretty crafty. When it works and does exactly what you want, your life is good. Fixing bugs in it is messy, though, as is adding new features, because it's a template language, and hence many of the constructs are ad hoc.)

Wrapping C code with pyrex

pyrex, as I discussed yesterday, is a weird hybrid of C and Python that's meant for generating fast Python-esque code. I'm not sure I'd call this "wrapping", but ... here goes.

First, write a `.pyx` file; in this case, I'm calling it `'hellomodule.pyx'`, instead of `'hello.pyx'`, so that I don't get confused with `'hello.c'`.

```

cdef extern from "hello.h":
    char * hello(char *s)

def hello_fn(s):
    return hello(s)

```

What the `'cdef'` says is, "grab the symbol `'hello'` from the file `'hello.h'`". Then you just go ahead and define your `'hello_fn'` as you would if it were Python.

and... that's it. You've still got to write a `setup.py`, of course:

```

from distutils.core import setup
from distutils.extension import Extension
from Pyrex.Distutils import build_ext

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

```

but then you can just run `'setup.py build_ext --inplace'` and you'll be able to `'import hellomodule; hellomodule.hello_fn'`.

ctypes

In Python 2.5, the `ctypes` module is included. This module lets you talk directly to shared libraries on both Windows and UNIX, which is pretty darned handy. But can it be used to call our C code directly?

The answer is yes, with a caveat or two.

First, you need to compile 'hello.c' into a shared library.

```
gcc -o hello.so -shared -fPIC hello.c
```

Then, you need to tell the system where to find the shared library.

```
export LD_LIBRARY_PATH=.
```

Now you can load the library with ctypes:

```
from ctypes import cdll
hello_lib = cdll.LoadLibrary("hello.so")
hello = hello_lib.hello
```

So far, so good -- now what happens if you run it?

```
>> print hello("world")
136040696
```

Whoops! You still need to tell Python/ctypes what kind of return value to expect! In this case, we're expecting a char pointer:

```
from ctypes import c_char_p
hello.restype = c_char_p
```

And now it will work:

```
>> print hello("world")
hello, world
```

Voila!

I should say that ctypes is not intended for this kind of wrapping, because of the whole LD_LIBRARY_PATH setting requirement. That is, it's really intended for accessing *system* libraries. But you can still use it for other stuff like this.

SIP

SIP is the tool used to generate Python bindings for Qt (PyQt), a graphics library. However, it can be used to wrap any C or C++ API.

As with SWIG, you have to start with a definition file. In this case, it's pretty easy: just put this in 'hello.sip':

```
%CModule hellomodule 0
char * hello(char *);
```

Now you need to write a 'configure' script:

```
import os
import sipconfig

# The name of the SIP build file generated by SIP and used by the build
# system.
build_file = "hello.sbf"
```

```

# Get the SIP configuration information.
config = sipconfig.Configuration()

# Run SIP to generate the code.
os.system(" ".join([config.sip_bin, "-c", ".", "-b", build_file, "hello.sip"]))

# Create the Makefile.
makefile = sipconfig.SIPModuleMakefile(config, build_file)

# Add the library we are wrapping. The name doesn't include any platform
# specific prefixes or extensions (e.g. the "lib" prefix on UNIX, or the
# ".dll" extension on Windows).
makefile.extra_libs = ["hello"]
makefile.extra_lib_dirs = ["."]

# Generate the Makefile itself.
makefile.generate()

```

Now, run 'configure.py', and then run 'make' on the generated Makefile, and your extension will be compiled.

(At this point I should say that I haven't really used SIP before, and I feel like it's much more powerful than this example would show you!)

Boost.Python

If you are an expert C++ programmer and want to wrap a lot of C++ code, I would recommend taking a look at the Boost.Python library, which lets you run C++ code from Python, and Python code from C++, seamlessly. I haven't used it at all, and it's too complicated to cover in a short period!

<http://www.boost-consulting.com/writing/bpl.html>

Recommendations

Based on my little survey above, I would suggest using SWIG to write wrappers for relatively small libraries, while SIP probably provides a more manageable infrastructure for wrapping large libraries (which I know I did not demonstrate!)

Pyrex is astonishingly easy to use, and it may be a good option if you have a small library to wrap. My guess is that you would spend a lot of time converting types back and forth from C/C++ to Python, but I could be wrong.

ctypes is excellent if you have a bunch of functions to run and you don't care about extracting complex data types from them: you just want to pass around the encapsulated data types between the functions in order to accomplish a goal.

One or two more notes on wrapping

As I said at the beginning, I tend to write procedural interfaces to my C++ code and then use Python to wrap them in an object-oriented interface. This lets me adjust the OO structure of my code more flexibly; on the flip side, I only use the code from Python, so I really don't care what the C++ code looks like as long as it runs fast ;). So, you might find it worthwhile to invest in figuring out how to wrap things in a more object-oriented manner.

Secondly, one of the biggest benefits I find from wrapping my C code in Python is that all of a sudden I can test it pretty easily. Testing is something you *do not* want to do in C, because you have to declare all

the variables and stuff that you use, and that just gets in the way of writing simple tests. I find that once I've wrapped something in Python, it becomes much more testable.

Packages for Multiprocessing

threading

Python has basic support for threading built in: for example, here's a program that runs two threads, each of which prints out messages after sleeping a particular amount of time:

```
from threading import Thread, local
import time

class MessageThread(Thread):
    def __init__(self, message, sleep):
        self.message = message
        self.sleep = sleep
        Thread.__init__(self)          # remember to run Thread init!

    def run(self):                      # automatically run by 'start'
        i = 0
        while i < 50:
            i += 1
            print i, self.message

            time.sleep(self.sleep)

t1 = MessageThread("thread - 1", 1)
t2 = MessageThread("thread - 2", 2)

t1.start()
t2.start()
```

However, due to the existence of the Global Interpreter Lock (GIL) (<http://docs.python.org/api/threads.html>), CPU-intensive code will not run faster on dual-core CPUs than it will on single-core CPUs.

Briefly, the idea is that the Python interpreter holds a global lock, and no Python code can be executed without holding that lock. (Code execution will still be interleaved, but no two Python instructions can execute at the same time.) Therefore, any Python code that you write (or GIL-naive C/C++ extension code) will not take advantage of multiple CPUs.

This is intentional:

<http://mail.python.org/pipermail/python-3000/2007-May/007414.html>

There is a long history of wrangling about the GIL, and there are a couple of good arguments for it. Briefly,

- it dramatically simplifies writing C extension code, because by default, C extension code does not need to know anything about threads.
- putting in locks appropriately to handle places where contention might occur is not only error-prone but makes the code quite slow; locks really affect performance.
- threaded code is difficult to debug, and most people don't need it, despite having been brainwashed to think that they do ;).

But we don't care about that: *we* do want our code to run on multiple CPUs. So first, let's dip back into C code: what do we have to do to make our C code release the GIL so that it can do a long computation?

Basically, just wrap I/O blocking code or CPU-intensive code in the following macros:

```
Py_BEGIN_ALLOW_THREADS
...Do some time-consuming operation...
Py_END_ALLOW_THREADS
```

This is actually pretty easy to do to your C code, and it does result in that code being run in parallel on multi-core CPUs. (note: example?)

The big problem with the GIL, however, is that it really means that you simply can't write parallel code in Python without jumping through some kind of hoop. One hoop is to write all your expensive code in C or C++ extension modules, and then use the BEGIN/END_ALLOW_THREADS construct above. However, there are some other options. Here are a few.

parallepython

parallepython is a system for controlling multiple Python processes on multiple machines. Here's an example program:

```
#!/usr/bin/python
def isprime(n):
    """Returns True if n is prime and False otherwise"""
    import math

    if n < 2:
        return False
    if n == 2:
        return True
    max = int(math.ceil(math.sqrt(n)))
    i = 2
    while i <= max:
        if n % i == 0:
            return False
        i += 1
    return True

def sum_primes(n):
    """Calculates sum of all primes below given integer n"""
    return sum([x for x in xrange(2, n) if isprime(x)])

####

import sys, time

import pp
# Creates jobserver with specified number of workers
job_server = pp.Server(ncpus=int(sys.argv[1]))

print "Starting pp with", job_server.get_ncpus(), "workers"

start_time = time.time()

# Submit a job of calculating sum_primes(100) for execution.
#
# * sum_primes - the function
# * (input,) - tuple with arguments for sum_primes
# * (isprime,) - tuple with functions on which sum_primes depends
#
# Execution starts as soon as one of the workers will become available

inputs = (100000, 100100, 100200, 100300, 100400, 100500, 100600, 100700)

jobs = []
for input in inputs:
```

```

    job = job_server.submit(sum_primes, (input,), (isprime,))
    jobs.append(job)

for job, input in zip(jobs, inputs):
    print "Sum of primes below", input, "is", job()

print "Time elapsed: ", time.time() - start_time, "s"
job_server.print_stats()

```

If you add "ppservers=('host1')" to to the line

```
pp.Server(...)
```

pp will check for parallepython servers running on those other hosts and send jobs to them as well.

The way parallepython works is it literally sends the Python code across the network & evaluates it there! It seems to work well.

Rpyc

[Rpyc](#) is a remote procedure call system built in (and tailored to) Python. It is basically a way to transparently control remote Python processes. For example, here's some code that will connect to an Rpyc server and ask the server to calculate the first 500 prime numbers:

```

from Rpyc import SocketConnection

# connect to the "remote" server c = SocketConnection("localhost")

# make sure it has the right code in its path c.modules.sys.path.append('/u/t/dev/misc/rpyc')

# tell it to execute 'primestuff.get_n_primes' primes =
c.modules.primestuff.get_n_primes(500) print primes[-20:]

```

Note that this is a synchronous connection, so the client waits for the result; you could also have it do the computation asynchronously, leaving the client free to request results from other servers.

In terms of parallel computing, the server has to be controlled directly, which makes it less than ideal. I think parallepython is a better choice for straightforward number crunching.

pyMPI

pyMPI is a nice Python implementation to the MPI (message-passing interface) library. MPI enables different processors to communicate with each other. I can't demo pyMPI, because I couldn't get it to work on my other machine, but here's some example code that computes pi to a precision of 1e-6 on however many machines you have running MPI.

```

import random
import mpi

def computePi(nsamples):
    rank, size = mpi.rank, mpi.size
    oldpi, pi, mypi = 0.0,0.0,0.0

    done = False
    while(not done):
        inside = 0
        for i in xrange(nsamples):
            x = random.random()

```

```

        y = random.random()
        if ((x*x)+(y*y)<1):
            inside+=1

    oldpi = pi
    mypi = (inside * 1.0)/nsamples
    pi = (4.0 / mpi.size) * mpi.allreduce(mypi, mpi.SUM)

    delta = abs(pi - oldpi)
    if(mpi.rank==0):
        print "pi:",pi," - delta:",delta
    if(delta < 0.00001):
        done = True
    return pi

if __name__ == "__main__":
    pi = computePi(10000)
    if(mpi.rank==0):
        print "Computed value of pi on",mpi.size,"processors is",pi

```

One big problem with MPI is that documentation is essentially absent, but I can still make a few points ;).

First, the "magic" happens in the 'allreduce' function up above, where it sums the results from all of the machines and then divides by the number of machines.

Second, pyMPI takes the unusual approach of actually building an MPI-aware Python interpreter, so instead of running your scripts in normal Python, you run them using 'pyMPI'.

multitask

multitask is not a multi-machine mechanism; it's a library that implements cooperative multitasking around I/O operations. Briefly, whenever you're going to do an I/O operation (like wait for more data from the network) you can tell multitask to yield to another thread of control. Here is a simple example where control is voluntarily yielded after a 'print':

```

import multitask

def printer(message):
    while True:
        print message
        yield

multitask.add(printer('hello'))
multitask.add(printer('goodbye'))
multitask.run()

```

Here's another example from the home page:

```

import multitask

def listener(sock):
    while True:
        conn, address = (yield multitask.accept(sock)) # WAIT
        multitask.add(client_handler(conn))

def client_handler(sock):
    while True:
        request = (yield multitask.recv(sock, 1024)) # WAIT
        if not request:
            break
        response = handle_request(request)
        yield multitask.send(sock, response) # WAIT

multitask.add(listener(sock))

```

```
multitask.run()
```

Useful Packages

subprocess

'subprocess' is a new addition (Python 2.4), and it provides a convenient and powerful way to run system commands. (...and you should use it instead of `os.system`, `commands.getstatusoutput`, or any of the Popen modules).

Unfortunately subprocess is a bit hard to use at the moment; I'm hoping to help fix that for Python 2.6, but in the meantime here are some basic commands.

Let's just try running a system command and retrieving the output:

```
>>> import subprocess
>>> p = subprocess.Popen(['/bin/echo', 'hello, world'], stdout=subprocess.PIPE)
>>> (stdout, stderr) = p.communicate()
>>> print stdout,
hello, world
```

What's going on is that we're starting a subprocess (running `'/bin/echo hello, world'`) and then asking for all of the output aggregated together.

We could, for short strings, read directly from `p.stdout` (which is a file handle):

```
>>> p = subprocess.Popen(['/bin/echo', 'hello, world'], stdout=subprocess.PIPE)
>>> print p.stdout.read(),
hello, world
```

but you could run into trouble here if the command returns a lot of data; you should use `communicate` to get the output instead.

Let's do something a bit more complicated, just to show you that it's possible: we're going to write to `'cat'` (which is basically an echo chamber):

```
>>> from subprocess import PIPE
>>> p = subprocess.Popen(["/bin/cat"], stdin=PIPE, stdout=PIPE)
>>> (stdout, stderr) = p.communicate('hello, world')
>>> print stdout,
hello, world
```

There are a number of more complicated things you can do with subprocess -- like interact with the `stdin` and `stdout` of other processes -- but they are fraught with peril.

rpy

[rpy](#) is an extension for R that lets R and Python talk naturally. For those of you that have never used R, it's a very nice package that's mainly used for statistics, and it has *tons* of libraries.

To use rpy, just

```
from rpy import *
```

The most important symbol that will be imported is `'r'`, which lets you run arbitrary R comments:

```
r("command")
```

For example, if you wanted to run a principle component analysis, you could do it like so:

```
from rpy import *
def plot_pca(filename):
    r("""data <- read.delim('%s', header=FALSE, sep=" ", nrows=5000)""" \
      % (filename,))
    r("""pca <- prcomp(data, scale=FALSE, center=FALSE)""")
    r("""pairs(pca$x[,1:3], pch=20)""")
plot_pca('vectors.txt')
```

Now, the problem with this code is that I'm really just using Python to drive R, which seems inefficient. You *can* go access the data directly if you want; I'm just using R's loading features directly because they're faster. For example,

```
x = r.pca['x']
```

is equivalent to 'x <- pca\$x'.

matplotlib

[matplotlib](#) is a plotting package that aims to make "simple things easy, and hard things possible". It's got a fair amount of matlab compatibility if you're into that.

Simple example:

```
x = [ i**2 for i in range(0, 500) ]
hist(x, 100)
```

Idiomatic Python Take 3: new-style classes

Someone (Lila) asked me a question about pickling and memory usage that led me on a chase through google, and along the way I was reminded that new-style classes do have one or two interesting points.

You may remember from the first day that there was a brief discussion of new-style classes. Basically, they're classes that inherit from 'object' explicitly:

```
>>> class N(object):
...     pass
```

and they have a bunch of neat features (covered [here](#) in detail). I'm going to talk about two of them: `__slots__` and descriptors.

`__slots__` are a memory optimization. As you know, you can assign any attribute you want to an object:

```
>>> n = N()
>>> n.test = 'some string'
>>> print n.test
some string
```

Now, the way this is implemented behind the scenes is that there's a dictionary hiding in 'n' (called 'n.__dict__') that holds all of the attributes. However, dictionaries consume a fair bit of memory above and beyond their contents, so it might be good to get rid of the dictionary in some circumstances and

specify precisely what attributes a class has.

You can do that by creating a `__slots__` entry:

```
>>> class M(object):
...     __slots__ = ['x', 'y', 'z']
```

Now objects of type 'M' will contain only enough space to hold those three attributes, and nothing else.

A side consequence of this is that you can no longer assign to arbitrary attributes, however!

```
>>> m = M()
>>> m.x = 5
>>> m.a = 10
Traceback (most recent call last):
...
AttributeError: 'M' object has no attribute 'a'
```

This will look strangely like some kind of type declaration to people familiar with B&D languages, but I assure you that it is not! You are supposed to use `__slots__` only for memory optimization...

Speaking of memory optimization (which is what got me onto this in the first place) apparently using new-style classes and `__slots__` both dramatically decrease memory consumption:

<http://mail.python.org/pipermail/python-list/2004-November/291840.html>

<http://mail.python.org/pipermail/python-list/2004-November/291986.html>

Managed attributes

Another nifty pair of features in new-style classes are managed attributes and descriptors.

You may know that in the olden days, you could intercept attribute access by overwriting `__getattr__`:

```
>>> class A:
...     def __getattr__(self, name):
...         if name == 'special':
...             return 5
...         return self.__dict__[name]
>>> a = A()
>>> a.special
5
```

This turns out to be kind of inefficient, because *every* attribute access now goes through `__getattr__`. Plus, it's a bit ugly and it can lead to buggy code.

Python 2.2 introduced "managed attributes". With managed attributes, you can *define* functions that handle the get, set, and del operations for an attribute:

```
>>> class B(object):
...     def _get_special(self):
...         return 5
...     special = property(_get_special)
>>> b = B()
>>> b.special
5
```

If you wanted to provide a `'_set_special'` function, you could do some really bizarre things:

```
>>> class B(object):
```

```

...     def _get_special(self):
...         return 5
...     def _set_special(self, value):
...         print 'ignoring', value
...     special = property(_get_special, _set_special)
>>> b = B()

```

Now, retrieving the value of the 'special' attribute will give you '5', no matter what you set it to:

```

>>> b.special
5
>>> b.special = 10
ignoring 10
>>> b.special
5

```

Ignoring the array of perverse uses you could apply, this is actually useful -- for one example, you can now do internal consistency checking on attributes, intercepting inconsistent values before they actually get set.

Descriptors

Descriptors are a related feature that let you implement attribute access functions in a different way. First, let's define a read-only descriptor:

```

>>> class D(object):
...     def __get__(self, obj, type=None):
...         print 'in get:', obj
...         return 6

```

Now attach it to a class:

```

>>> class A(object):
...     val = D()

>>> a = A()
>>> a.val                                     # doctest: +ELLIPSIS
in get: <A object at ...>
6

```

What happens is that 'a.val' is checked for the presence of a `__get__` function, and if such exists, it is called instead of returning 'val'. You can also do this with `__set__` and `__delete__`:

```

>>> class D(object):
...     def __get__(self, obj, type=None):
...         print 'in get'
...         return 6
...
...     def __set__(self, obj, value):
...         print 'in set:', value
...
...     def __delete__(self, obj):
...         print 'in del'

>>> class A(object):
...     val = D()
>>> a = A()
>>> a.val = 15
in set: 15
>>> del a.val
in del
>>> print a.val
in get
6

```

This can actually give you control over things like the *types* of objects that are assigned to particular classes: no mean thing.

GUI Gossip

Tkinter

- fairly primitive;
- but: comes with every Python install!
- still a bit immature (feb 2007) for Mac OS X native ("Aqua"); X11 version works fine on OS X.

PyQT (<http://www.riverbankcomputing.co.uk/pyqt/>)

- mature;
- cross platform;
- freely available for Open Source Software use;
- has a testing framework!

KWWidgets (<http://www.kwwidgets.org/>)

- immature; based on Tk, so Mac OS X native is still a bit weak;
- lightweight;
- attractive;
- has a testing framework!

pyFLTK (<http://sf.net/projects/pyfltk/>)

- cross platform;
- FLTK is mature, although primitive;
- not very pretty;
- very lightweight;

wxWindows (<http://www.wxwindows.org/>)

- cross platform;
- mature?; looks good.
- no personal or "friend" experience;
- try reading <http://www.ibm.com/developerworks/library/l-wxwin.html>

pyGTK (<http://www.pygtk.org/>)

- cross platform;
- mature; looks good.
- no personal or "friend" experience;
- UI designer;

Mild recommendation: start with Qt, which is apparently very mature and very powerful.

Python 3.0

What's coming in Python 3000 (a.k.a. Python 3.0)?

First of all, Python 3000 will be out sometime in 2008; large parts of it have already been implemented. It is simply an increment on the current code base.

The biggest point is that Python 3000 will break backwards compatibility, abruptly. This is very unusual for Python, but is necessary to get rid of some old cruft. In general, Python has been very good about updating slowly and incrementally without breaking backwards compatibility very much; this approach is being abandoned for the jump from 2.x to 3.x.

However, the actual impact of this is likely to be small. There will be a few expressions that no longer work -- for example, 'dict.has_key' is being removed, because you can just do 'if key in dict' -- but a lot of the changes are behind the scenes, e.g. functions that currently return lists will return iterators (dict.iterkeys -> dict.keys).

The biggest impact on this audience (scientists & numerical people) is probably that (in Python 3.0) 6 / 5 will no longer be 0, and <> is being removed.

Where lots of existing code must be made Python 3k compatible, you will be able to use an automated conversion tool. This should "just work" except for cases where there is ambiguity in intent.

The most depressing aspect of Py3k (for me) is that the stdlib is not being reorganized, but this does mean that most existing modules will still be in the same place!

See [David Mertz's blog](#) for his summary of the changes.