Weave (`scipy.weave` (../weave.html#module-scipy.weave))

Outline

Contents

- Weave (scipy.weave (../weave.html#module-scipy.weave))
  - Outline
  - Introduction
  - Requirements
  - Installation
  - Testing
    - Testing Notes:
  - Benchmarks
  - Inline
    - More with printf
    - More examples
      - Binary search
      - Dictionary Sort
      - NumPy – cast/copy/transpose
      - wxPython
    - Keyword Option
    - Inline Arguments
    - Distutils keywords
      - Keyword Option Examples
      - Returning Values
        - The issue with `locals()`
      - A quick look at the code
    - Technical Details
    - Passing Variables in/out of the C/C++ code
    - Type Conversions
      - NumPy Argument Conversion
      - String, List, Tuple, and Dictionary Conversion
      - File Conversion
      - Callable, Instance, and Module Conversion
      - Customizing Conversions
    - The Catalog
      - Function Storage
      - Catalog search paths and the PYTHONCOMPILED variable
  - Blitz
    - Requirements
    - Limitations

        - NumPy efficiency issues: What compilation buys you
        - The Tools
            - Parser
            - Blitz and NumPy
        - Type definitions and coercion
        - Cataloging Compiled Functions
        - Checking Array Sizes
        - Creating the Extension Module
    - Extension Modules
        - A Simple Example
        - Fibonacci Example
    - Customizing Type Conversions – Type Factories
    - Things I wish **`weave`** did

Introduction

The scipy.weave (../weave.html#module-scipy.weave) (below just weave) package provides tools for including C/C++ code within in Python code. This offers both another level of optimization to those who need it, and an easy way to modify and extend any supported extension libraries such as wxPython and hopefully VTK soon. Inlining C/C++ code within Python generally results in speed ups of 1.5x to 30x speed-up over algorithms written in pure Python (However, it is also possible to slow things down...). Generally algorithms that require a large number of calls to the Python API don't benefit as much from the conversion to C/C++ as algorithms that have inner loops completely convertible to C.

There are three basic ways to use `weave`. The `weave.inline()` function executes C code directly within Python, and `weave.blitz()` translates Python NumPy expressions to C++ for fast execution. `blitz()` was the original reason `weave` was built. For those interested in building extension libraries, the `ext_tools` module provides classes for building extension modules within Python.

Most of `weave's` functionality should work on Windows and Unix, although some of its functionality requires `gcc` or a similarly modern C++ compiler that handles templates well. Up to now, most testing has been done on Windows 2000 with Microsoft's C++ compiler (MSVC) and with gcc (mingw32 2.95.2 and 2.95.3-6). All tests also pass on Linux (RH 7.1 with gcc 2.96), and I've had reports that it works on Debian also (thanks Pearu).

The `inline` and `blitz` provide new functionality to Python (although I've recently learned about the PyInline (http://pyinline.sourceforge.net/) project which may offer similar functionality to `inline`). On the other hand, tools for building Python extension modules already exists (SWIG, SIP, pycpp, CXX, and others). As of yet, I'm not sure where `weave` fits in this spectrum. It is closest in flavor to CXX in that it makes creating new C/C++ extension modules pretty easy. However, if you're wrapping a gaggle of legacy functions or classes, SWIG and friends are definitely the better choice. `weave` is set up so that you can customize how Python types are converted to C types in `weave`. This is great for `inline()`, but, for wrapping legacy code, it is more flexible to specify things the other way around – that is how C types map to Python types. This `weave` does not do. I guess it would be possible to build such a tool on top of `weave`, but with good tools like SWIG around, I'm not sure the effort produces any new capabilities. Things like function overloading are probably easily implemented in `weave` and it might be easier to mix Python/C code in function calls, but nothing beyond this comes to mind. So, if you're developing new extension modules or optimizing Python functions in C, `weave.ext_tools()` might be the tool for you. If you're wrapping legacy code, stick with SWIG.

The next several sections give the basics of how to use `weave`. We'll discuss what's happening under the covers in more detail later on. Serious users will need to at least look at the type conversion section to understand how Python variables map to C/C++ types and how to customize this behavior. One other note. If you don't know C or C++ then these docs are probably of very little help to you. Further, it'd be helpful if you know something about writing Python extensions. `weave` does quite a bit for you, but for anything complex, you'll need to do some conversions, reference counting, etc.

> **Note:**
> `weave` is actually part of the SciPy (https://www.scipy.org) package. However, it also works fine as a standalone package (you can install from scipy/weave with `python setup.py install`). The examples here are given as if it is used as a stand alone package. If you are using from within scipy, you can use `from scipy import weave` and the examples will work identically.

Requirements

- Python

  I use 2.1.1. Probably 2.0 or higher should work.

- C++ compiler

  `weave` uses `distutils` to actually build extension modules, so it uses whatever compiler was originally used to build Python. `weave` itself requires a C++ compiler. If you used a C++ compiler to build Python, your probably fine.

  On Unix gcc is the preferred choice because I've done a little testing with it. All testing has been done with gcc, but I expect the majority of compilers should work for `inline` and `ext_tools`. The one issue I'm not sure about is that I've hard coded things so that compilations are linked with the `stdc++` library. *Is this standard across Unix compilers, or is this a gcc-ism?*

  For `blitz()`, you'll need a reasonably recent version of gcc. 2.95.2 works on windows and 2.96 looks fine on Linux. Other versions are likely to work. Its likely that KAI's C++ compiler and maybe some others will work, but I haven't tried. My advice is to use gcc for now unless your willing to tinker with the code some.

  On Windows, either MSVC or gcc (mingw32 (http://www.mingw.org%3Ewww.mingw.org)) should work. Again, you'll need gcc for `blitz()` as the MSVC compiler doesn't handle templates well.

  I have not tried Cygwin, so please report success if it works for you.

- NumPy

  The python NumPy (http://numeric.scipy.org/) module is required for `blitz()` to work and for numpy.distutils which is used by weave.

Installation

There are currently two ways to get `weave`. First, `weave` is part of SciPy and installed automatically (as a sub- package) whenever SciPy is installed. Second, since `weave` is useful outside of the scientific community, it has been setup so that it can be used as a stand-alone module.

The stand-alone version can be downloaded from here (https://www.scipy.org/Weave). Instructions for installing should be found there as well. setup.py file to simplify installation.

Testing

Once `weave` is installed, fire up python and run its unit tests.

```
>>> import weave
>>> weave.test()
runs long time... spews tons of output and a few warnings
.
.
.
..............................................................
................................................................
..................................................
----------------------------------------------------------------------
Ran 184 tests in 158.418s
OK
>>>
```

This takes a while, usually several minutes. On Unix with remote file systems, I've had it take 15 or so minutes. In the end, it should run about 180 tests and spew some speed results along the way. If you get errors, they'll be reported at the end of the output. Please report errors that you find. Some tests are known to fail at this point.

If you only want to test a single module of the package, you can do this by running test() for that specific module.

```
>>> import weave.scalar_spec
>>> weave.scalar_spec.test()
.......
 ----------------------------------------------------------------------
Ran 7 tests in 23.284s
```

Testing Notes:

- Windows 1

  I've had some test fail on windows machines where I have msvc, gcc-2.95.2 (in c:gcc-2.95.2), and gcc-2.95.3-6 (in c:gcc) all installed. My environment has c:gcc in the path and does not have c:gcc-2.95.2 in the path. The test process runs very smoothly until the end where several test using gcc fail with cpp0 not found by g++. If I check os.system('gcc -v') before running tests, I get gcc-2.95.3-6. If I check after running tests (and after failure), I get gcc-2.95.2. ??huh??. The os.environ['PATH'] still has c:gcc first in it and is not corrupted (msvc/distutils messes with the environment variables, so we have to undo its work in some places). If anyone else sees this, let me know - - it may just be an quirk on my machine (unlikely). Testing with the gcc- 2.95.2 installation always works.

- Windows 2

  If you run the tests from PythonWin or some other GUI tool, you'll get a ton of DOS windows popping up periodically as `weave` spawns the compiler multiple times. Very annoying. Anyone know how to fix this?

- wxPython

  wxPython tests are not enabled by default because importing wxPython on a Unix machine without access to a X-term will cause the program to exit. Anyone know of a safe way to detect whether wxPython can be imported and whether a display exists on a machine?

Benchmarks

This section has not been updated from old scipy weave and Numeric....

This section has a few benchmarks – that's all people want to see anyway right? These are mostly taken from running files in the `weave/example` directory and also from the test scripts. Without more information about what the test actually do, their value is limited. Still, their here for the curious. Look at the example scripts for more specifics about what problem was actually solved by each run. These examples are run under windows 2000 using Microsoft Visual C++ and python2.1 on a 850 MHz PIII laptop with 320 MB of RAM. Speed up is the improvement (degradation) factor of `weave` compared to conventional Python functions. `The blitz()` comparisons are shown compared to NumPy.

inline and ext_tools

Algorithm	**Speed up**
binary search	1.50
fibonacci (recursive)	82.10
fibonacci (loop)	9.17
return None	0.14
map	1.20
dictionary sort	2.54
vector quantization	37.40

blitz – double precision

Algorithm	**Speed up**
a = b + c 512x512	3.05
a = b + c + d 512x512	4.59
5 pt avg. filter, 2D Image 512x512	9.01
Electromagnetics (FDTD) 100x100x100	8.61

The benchmarks shown `blitz` in the best possible light. NumPy (at least on my machine) is significantly worse for double precision than it is for single precision calculations. If your interested in single precision results, you can pretty much divide the double precision speed up by 3 and you'll be close.

Inline

`inline()` compiles and executes C/C++ code on the fly. Variables in the local and global Python scope are also available in the C/C++ code. Values are passed to the C/C++ code by assignment much like variables are passed into a standard Python function. Values are returned from the C/C++ code through a special argument called return_val. Also, the contents of mutable objects can be changed within the C/C++ code and the changes remain after the C code exits and returns to Python. (more on this later)

Here's a trivial `printf` example using `inline()`:

```
>>> import weave
>>> a  = 1
>>> weave.inline('printf("%d\\n",a);',['a'])
1
```

In this, its most basic form, `inline(c_code, var_list)` requires two arguments. `c_code` is a string of valid C/C++ code. `var_list` is a list of variable names that are passed from Python into C/C++. Here we have a simple `printf` statement that writes the Python variable `a` to the screen. The first time you run this, there will be a pause while the code is written to a .cpp file, compiled into an extension module, loaded into Python, cataloged for future use, and executed. On windows (850 MHz PIII), this takes about 1.5 seconds when using Microsoft's C++ compiler (MSVC) and 6-12 seconds using gcc (mingw32 2.95.2). All subsequent executions of the code will happen very quickly because the code only needs to be compiled once. If you kill and restart the interpreter and then execute the same code fragment again, there will be a much shorter delay in the fractions of seconds range. This is because `weave` stores a catalog of all previously compiled functions in an on disk cache. When it sees a string that has been compiled, it loads the already compiled module and executes the appropriate function.

> **Note:**
> If you try the `printf` example in a GUI shell such as IDLE, PythonWin, PyShell, etc., you're unlikely to see the output. This is because the C code is writing to stdout, instead of to the GUI window. This doesn't mean that inline doesn't work in these environments – it only means that standard out in C is not the same as the standard out for Python in these cases. Non input/output functions will work as expected.

Although effort has been made to reduce the overhead associated with calling inline, it is still less efficient for simple code snippets than using equivalent Python code. The simple `printf` example is actually slower by 30% or so than using Python `print` statement. And, it is not difficult to create code fragments that are 8-10 times slower using inline than equivalent Python. However, for more complicated algorithms, the speedup can be worthwhile – anywhere from 1.5-30 times faster. Algorithms that have to manipulate Python objects (sorting a list) usually only see a factor of 2 or so improvement. Algorithms that are highly computational or manipulate NumPy arrays can see much larger improvements. The examples/vq.py file shows a factor of 30 or more improvement on the vector quantization algorithm that is used heavily in information theory and classification problems.

More with printf

MSVC users will actually see a bit of compiler output that distutils does not suppress the first time the code executes:

```
>>>
>>> weave.inline(r'printf("%d\n",a);',['a'])
sc_e013937dbc8c647ac62438874e5795131.cpp
   Creating library C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp
   \Release\sc_e013937dbc8c647ac62438874e5795131.lib and
   object C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_e
013937dbc8c647ac62438874e5795131.exp
1
```

Nothing bad is happening, its just a bit annoying. * Anyone know how to turn this off?*

This example also demonstrates using 'raw strings'. The `r` preceding the code string in the last example denotes that this is a 'raw string'. In raw strings, the backslash character is not interpreted as an escape character, and so it isn't necessary to use a double backslash to indicate that the 'n' is meant to be interpreted in the C `printf` statement instead of by Python. If your C code contains a lot of strings and control characters, raw strings might make things easier. Most of the time, however, standard strings work just as well.

The `printf` statement in these examples is formatted to print out integers. What happens if `a` is a string? `inline` will happily, compile a new version of the code to accept strings as input, and execute the code. The result?

```
>>>
>>> a = 'string'
>>> weave.inline(r'printf("%d\n",a);',['a'])
32956972
```

In this case, the result is non-sensical, but also non-fatal. In other situations, it might produce a compile time error because `a` is required to be an integer at some point in the code, or it could produce a segmentation fault. Its possible to protect against passing `inline` arguments of the wrong data type by using asserts in Python.

```
>>> a = 'string'
>>> def protected_printf(a):
...     assert(type(a) == type(1))
...     weave.inline(r'printf("%d\n",a);',['a'])
>>> protected_printf(1)
 1
>>> protected_printf('string')
AssertError...
```

For printing strings, the format statement needs to be changed. Also, weave doesn't convert strings to char*. Instead it uses CXX Py::String type, so you have to do a little more work. Here we convert it to a C++ std::string and then ask cor the char* version.

```
>>> a = 'string'
>>> weave.inline(r'printf("%s\n",std::string(a).c_str());',['a'])
string
```

XXX:
This is a little convoluted. Perhaps strings should convert to `std::string` objects instead of CXX objects. Or maybe to `char*`.

As in this case, C/C++ code fragments often have to change to accept different types. For the given printing task, however, C++ streams provide a way of a single statement that works for integers and strings. By default, the stream objects live in the std (standard) namespace and thus require the use of `std::`.

```
>>> weave.inline('std::cout << a << std::endl;',['a'])
1
>>> a = 'string'
>>> weave.inline('std::cout << a << std::endl;',['a'])
string
```

Examples using `printf` and `cout` are included in examples/print_example.py.

More examples

This section shows several more advanced uses of `inline`. It includes a few algorithms from the Python Cookbook (https://code.activestate.com/recipes/langs/python/) that have been re-written in inline C to improve speed as well as a couple examples using NumPy and wxPython.

Binary search

Lets look at the example of searching a sorted list of integers for a value. For inspiration, we'll use Kalle Svensson's binary_search() (https://code.activestate.com/recipes/81188/) algorithm from the Python Cookbook. His recipe follows:

```
def binary_search(seq, t):
    min = 0; max = len(seq) - 1
    while 1:
        if max < min:
            return -1
        m = (min  + max)  / 2
        if seq[m] < t:
            min = m  + 1
        elif seq[m] > t:
            max = m  - 1
        else:
            return m
```

This Python version works for arbitrary Python data types. The C version below is specialized to handle integer values. There is a little type checking done in Python to assure that we're working with the correct data types before heading into C. The variables `seq` and `t` don't need to be declared because `weave` handles converting and declaring them in the C code. All other temporary variables such as `min, max`, etc. must be declared – it is C after all. Here's the new mixed Python/C function:

```
def c_int_binary_search(seq,t):
    # do a little type checking in Python
    assert(type(t) == type(1))
    assert(type(seq) == type([]))

    # now the C code
    code = """
           #line 29 "binary_search.py"
           int val, m, min = 0;
           int max = seq.length() - 1;
           PyObject *py_val;
           for(;;)
           {
               if (max < min  )
               {
                   return_val =  Py::new_reference_to(Py::Int(-1));
                   break;
               }
               m =  (min + max) /2;
               val = py_to_int(PyList_GetItem(seq.ptr(),m),"val");
               if (val  < t)
                   min = m  + 1;
               else if (val >  t)
                   max = m - 1;
               else
               {
                   return_val = Py::new_reference_to(Py::Int(m));
                   break;
               }
           }
           """
    return inline(code,['seq','t'])
```

We have two variables `seq` and `t` passed in. `t` is guaranteed (by the `assert`) to be an integer. Python integers are converted to C int types in the transition from Python to C. `seq` is a Python list. By default, it is translated to a CXX list object. Full documentation for the CXX library can be found at its website (http://cxx.sourceforge.net/). The basics are that the CXX provides C++ class equivalents for Python objects that simplify, or at least object orientify, working with Python objects in C/C++. For example, `seq.length()` returns the length of the list. A little more about CXX and its class methods, etc. is in the *Type Conversions* section.

Note:
CXX uses templates and therefore may be a little less portable than another alternative by Gordan McMillan called SCXX which was inspired by CXX. It doesn't use templates so it should compile faster and be more portable. SCXX has a few less features, but it appears to me that it would mesh with the needs of weave quite well. Hopefully xxx_spec files will be written for SCXX in the future, and we'll be able to compare on a more empirical basis. Both sets of spec files will probably stick around, it just a question of which becomes the default.

Most of the algorithm above looks similar in C to the original Python code. There are two main differences. The first is the setting of `return_val` instead of directly returning from the C code with a `return` statement. `return_val` is an automatically defined variable of type `PyObject*` that is returned from the C code back to Python. You'll have to handle reference counting issues when setting this variable. In this example, CXX classes and functions handle the dirty work. All CXX functions and classes live in the namespace `Py::`. The following code converts the integer `m` to a CXX `Int()` object and then to a `PyObject*` with an incremented reference count using `Py::new_reference_to()`.

```
return_val = Py::new_reference_to(Py::Int(m));
```

The second big differences shows up in the retrieval of integer values from the Python list. The simple Python `seq[i]` call balloons into a C Python API call to grab the value out of the list and then a separate call to `py_to_int()` that converts the PyObject* to an integer. `py_to_int()` includes both a NULL check and a `PyInt_Check()` call as well as the conversion call. If either of the checks fail, an exception is raised. The entire C++ code block is executed with in a `try/catch` block that handles exceptions much like Python does. This removes the need for most error checking code.

It is worth note that CXX lists do have indexing operators that result in code that looks much like Python. However, the overhead in using them appears to be relatively high, so the standard Python API was used on the `seq.ptr()` which is the underlying `PyObject*` of the List object.

The `#line` directive that is the first line of the C code block isn't necessary, but it's nice for debugging. If the compilation fails because of the syntax error in the code, the error will be reported as an error in the Python file "binary_search.py" with an offset from the given line number (29 here).

So what was all our effort worth in terms of efficiency? Well not a lot in this case. The examples/binary_search.py file runs both Python and C versions of the functions As well as using the standard `bisect` module. If we run it on a 1 million element list and run the search 3000 times (for 0- 2999), here are the results we get:

```
C:\home\ej\wrk\scipy\weave\examples> python binary_search.py
Binary search for 3000 items in 1000000 length list of integers:
speed in python: 0.159999966621
speed of bisect: 0.121000051498
speed up: 1.32
speed in c: 0.110000014305
speed up: 1.45
speed in c(no asserts): 0.0900000333786
speed up: 1.78
```

So, we get roughly a 50-75% improvement depending on whether we use the Python asserts in our C version. If we move down to searching a 10000 element list, the advantage evaporates. Even smaller lists might result in the Python version being faster. I'd like to say that moving to NumPy lists (and getting rid of the GetItem() call) offers a substantial speed up, but my preliminary efforts didn't produce one. I think the log(N) algorithm is to blame. Because the algorithm is nice, there just isn't much time spent computing things, so moving to C isn't that big of a win. If there are ways to reduce conversion overhead of values, this may improve the C/Python speed up. Anyone have other explanations or faster code, please let me know.

Dictionary Sort

The demo in examples/dict_sort.py is another example from the Python CookBook. This submission (https://code.activestate.com/recipes/52306/), by Alex Martelli, demonstrates how to return the values from a dictionary sorted by their keys:

```
def sortedDictValues3(adict):
    keys = adict.keys()
    keys.sort()
    return map(adict.get, keys)
```

Alex provides 3 algorithms and this is the 3rd and fastest of the set. The C version of this same algorithm follows:

```
def c_sort(adict):
    assert(type(adict) == type({}))
    code = """
    #line 21 "dict_sort.py"
    Py::List keys = adict.keys();
    Py::List items(keys.length()); keys.sort();
    PyObject* item = NULL;
    for(int i = 0;  i < keys.length();i++)
    {
        item = PyList_GET_ITEM(keys.ptr(),i);
        item = PyDict_GetItem(adict.ptr(),item);
        Py_XINCREF(item);
        PyList_SetItem(items.ptr(),i,item);
    }
    return_val = Py::new_reference_to(items);
    """
    return inline_tools.inline(code,['adict'],verbose=1)
```

Like the original Python function, the C++ version can handle any Python dictionary regardless of the key/value pair types. It uses CXX objects for the most part to declare python types in C++, but uses Python API calls to manipulate their contents. Again, this choice is made for speed. The C++ version, while more complicated, is about a factor of 2 faster than Python.

```
C:\home\ej\wrk\scipy\weave\examples> python dict_sort.py
Dict sort of 1000 items for 300 iterations:
 speed in python: 0.319999933243
[0, 1, 2, 3, 4]
 speed in c: 0.151000022888
 speed up: 2.12
[0, 1, 2, 3, 4]
```

NumPy – cast/copy/transpose

CastCopyTranspose is a function called quite heavily by Linear Algebra routines in the NumPy library. Its needed in part because of the row-major memory layout of multi-dimensional Python (and C) arrays vs. the col-major order of the underlying Fortran algorithms. For small matrices (say 100x100 or less), a significant portion of the common routines such as LU decomposition or singular value decomposition are spent in this setup routine. This shouldn't happen. Here is the Python version of the function using standard NumPy operations.

```
def _castCopyAndTranspose(type, array):
    if a.typecode() == type:
        cast_array = copy.copy(NumPy.transpose(a))
    else:
        cast_array = copy.copy(NumPy.transpose(a).astype(type))
    return cast_array
```

And the following is a inline C version of the same function:

```
from weave.blitz_tools import blitz_type_factories
from weave import scalar_spec
from weave import inline
def _cast_copy_transpose(type,a_2d):
    assert(len(shape(a_2d)) == 2)
    new_array = zeros(shape(a_2d),type)
    NumPy_type = scalar_spec.NumPy_to_blitz_type_mapping[type]
    code = \
    """
    for(int i = 0;i < _Na_2d[0]; i++)
        for(int j = 0;  j < _Na_2d[1]; j++)
            new_array(i,j) = (%s) a_2d(j,i);
    """ % NumPy_type
    inline(code,['new_array','a_2d'],
           type_factories = blitz_type_factories,compiler='gcc')
    return new_array
```

This example uses blitz++ arrays instead of the standard representation of NumPy arrays so that indexing is simpler to write. This is accomplished by passing in the blitz++ "type factories" to override the standard Python to C++ type conversions. Blitz++ arrays allow you to write clean, fast code, but they also are sloooow to compile (20 seconds or more for this snippet). This is why they aren't the default type used for Numeric arrays (and also because most compilers can't compile blitz arrays...). `inline()` is also forced to use 'gcc' as the compiler because the default compiler on Windows (MSVC) will not compile blitz code. ('gcc' I think will use the standard compiler on Unix machine instead of explicitly forcing gcc (check this)) Comparisons of the Python vs inline C++ code show a factor of 3 speed up. Also shown are the results of an "inplace" transpose routine that can be used if the output of the linear algebra routine can overwrite the original matrix (this is often appropriate). This provides another factor of 2 improvement.

```
#C:\home\ej\wrk\scipy\weave\examples> python cast_copy_transpose.py
# Cast/Copy/Transposing (150,150)array 1 times
#  speed in python: 0.870999932289
#  speed in c: 0.25
#  speed up: 3.48
#  inplace transpose c: 0.129999995232
#  speed up: 6.70
```

wxPython

`inline` knows how to handle wxPython objects. That's nice in and of itself, but it also demonstrates that the type conversion mechanism is reasonably flexible. Chances are, it won't take a ton of effort to support special types you might have. The examples/wx_example.py borrows the scrolled window example from the wxPython demo, accept that it mixes inline C code in the middle of the drawing function.

```
def DoDrawing(self, dc):

    red = wxNamedColour("RED");
    blue = wxNamedColour("BLUE");
    grey_brush = wxLIGHT_GREY_BRUSH;
    code = \
    """
    #line 108 "wx_example.py"
    dc->BeginDrawing();
    dc->SetPen(wxPen(*red,4,wxSOLID));
    dc->DrawRectangle(5,5,50,50);
    dc->SetBrush(*grey_brush);
    dc->SetPen(wxPen(*blue,4,wxSOLID));
    dc->DrawRectangle(15, 15, 50, 50);
    """
    inline(code,['dc','red','blue','grey_brush'])

    dc.SetFont(wxFont(14, wxSWISS, wxNORMAL, wxNORMAL))
    dc.SetTextForeground(wxColour(0xFF, 0x20, 0xFF))
    te = dc.GetTextExtent("Hello World")
    dc.DrawText("Hello World", 60, 65)

    dc.SetPen(wxPen(wxNamedColour('VIOLET'), 4))
    dc.DrawLine(5, 65+te[1], 60+te[0], 65+te[1])
    ...
```

Here, some of the Python calls to wx objects were just converted to C++ calls. There isn't any benefit, it just demonstrates the capabilities. You might want to use this if you have a computationally intensive loop in your drawing code that you want to speed up. On windows, you'll have to use the MSVC compiler if you use the standard wxPython DLLs distributed by Robin Dunn. That's because MSVC and gcc, while binary compatible in C, are not binary compatible for C++. In fact, its probably best, no matter what platform you're on, to specify that `inline` use the same compiler that was used to build wxPython to be on the safe side. There isn't currently a way to learn this info from the library – you just have to know. Also, at least on the windows platform, you'll need to install the wxWindows libraries and link to them. I think there is a way around this, but I haven't found it yet – I get some linking errors dealing with wxString. One final note. You'll probably have to tweak weave/wx_spec.py or weave/wx_info.py for your machine's configuration to point at the correct directories etc. There. That should sufficiently scare people into not even looking at this... :)

Keyword Option

The basic definition of the `inline()` function has a slew of optional variables. It also takes keyword arguments that are passed to `distutils` as compiler options. The following is a formatted cut/paste of the argument section of `inline's` doc-string. It explains all of the variables. Some examples using various options will follow.

```
def inline(code,arg_names,local_dict = None, global_dict = None,
           force = 0,
           compiler='',
           verbose = 0,
           support_code = None,
           customize=None,
           type_factories = None,
           auto_downcast=1,
           **kw):
```

`inline` has quite a few options as listed below. Also, the keyword arguments for distutils extension modules are accepted to specify extra information needed for compiling.

Inline Arguments

code string. A string of valid C++ code. It should not specify a return statement. Instead it should assign results that need to be returned to Python in the return_val. arg_names list of strings. A list of Python variable names that should be transferred from Python into the C/C++ code. local_dict optional. dictionary. If specified, it is a dictionary of values that should be used as the local scope for the C/C++ code. If local_dict is not specified the local dictionary of the calling function is used. global_dict optional. dictionary. If specified, it is a dictionary of values that should be used as the global scope for the C/C++ code. If global_dict is not specified the global dictionary of the calling function is used. force optional. 0 or 1. default 0. If 1, the C++ code is compiled every time inline is called. This is really only useful for debugging, and probably only useful if you're editing support_code a lot. compiler optional. string. The name of compiler to use when compiling. On windows, it understands 'msvc' and 'gcc' as well as all the compiler names understood by distutils. On Unix, it'll only understand the values understood by distutils. (I should add 'gcc' though to this).

On windows, the compiler defaults to the Microsoft C++ compiler. If this isn't available, it looks for mingw32 (the gcc compiler).

On Unix, it'll probably use the same compiler that was used when compiling Python. Cygwin's behavior should be similar.

verbose optional. 0,1, or 2. default 0. Specifies how much information is printed during the compile phase of inlining code. 0 is silent (except on windows with msvc where it still prints some garbage). 1 informs you when compiling starts, finishes, and how long it took. 2 prints out the command lines for the compilation process and can be useful if you're having problems getting code to work. Its handy for finding the name of the .cpp file if you need to examine it. verbose has no affect if the compilation isn't necessary. support_code optional. string. A string of valid C++ code declaring extra code that might be needed by your compiled function. This could be declarations of functions, classes, or structures. customize optional. base_info.custom_info object. An alternative way to specify support_code, headers, etc. needed by the function see the weave.base_info module for more details. (not sure this'll be used much). type_factories optional. list of type specification factories. These guys are what convert Python data types to C/C++ data types. If you'd like to use a different set of type conversions than the default, specify them here. Look in the type conversions section of the main documentation for examples. auto_downcast optional. 0 or 1. default 1. This only affects functions that have Numeric arrays as input variables. Setting this to 1 will cause all floating point values to be cast as float instead of double if all the NumPy arrays are of type float. If even one of the arrays has type double or double complex, all variables maintain there standard types.

Distutils keywords

`inline()` also accepts a number of `distutils` keywords for controlling how the code is compiled. The following descriptions have been copied from Greg Ward's `distutils.extension.Extension` class doc- strings for convenience: sources [string] list of source filenames, relative to the distribution root (where the setup script lives), in Unix form (slashseparated) for portability. Source files may be C, C++, SWIG (.i), platform- specific resource files, or whatever else is recognized by the “build_ext” command as source for a Python extension. Note: The module_path file is always appended to the front of this list include_dirs [string] list of directories to search for C/C++ header files (in Unix form for portability) define_macros [(name : string, value : string|None)] list of macros to define; each macro is defined using a 2-tuple, where ‘value’ is either the string to define it to or None to define it without a particular value (equivalent of “#define FOO” in source or -DFOO on Unix C compiler command line) undef_macros [string] list of macros to undefine explicitly library_dirs [string] list of directories to search for C/C++ libraries at link time libraries [string] list of library names (not filenames or paths) to link against runtime_library_dirs [string] list of directories to search for C/C++ libraries at run time (for shared extensions, this is when the extension is loaded) extra_objects [string] list of extra files to link with (eg. object files not implied by ‘sources’, static library that must be explicitly specified, binary resource files, etc.) extra_compile_args [string] any extra platform- and compiler-specific information to use when compiling the source files in ‘sources’. For platforms and compilers where “command line” makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything. extra_link_args [string] any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for ‘extra_compile_args’. export_symbols [string] list of symbols to be exported from a shared extension. Not used on all platforms, and not generally necessary for Python extensions, which typically export exactly one symbol: “init” + extension_name.

Keyword Option Examples

We’ll walk through several examples here to demonstrate the behavior of `inline` and also how the various arguments are used. In the simplest (most) cases, `code` and `arg_names` are the only arguments that need to be specified. Here’s a simple example run on Windows machine that has Microsoft VC++ installed.

```
>>>
>>> from weave import inline
>>> a = 'string'
>>> code = """
...        int l = a.length();
...        return_val = Py::new_reference_to(Py::Int(l));
...        """
>>> inline(code,['a'])
 sc_86e98826b65b047ffd2cd5f479c627f12.cpp
Creating
   library C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_
86e98826b65b047ffd2cd5f479c627f12.lib
and object C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_
86e98826b65b047ff
d2cd5f479c627f12.exp
6
>>> inline(code,['a'])
6
```

When `inline` is first run, you’ll notice that pause and some trash printed to the screen. The “trash” is actually part of the compiler’s output that distutils does not suppress. The name of the extension file, `sc_bighonkingnumber.cpp`, is generated from the SHA-256 check sum of the C/C++ code fragment. On Unix or windows machines with only gcc installed, the trash will not appear. On the second call, the code fragment is not compiled since it already exists, and only the answer is returned. Now kill the interpreter and restart, and run the same code with a different string.

>>>

```
>>> from weave import inline
>>> a = 'a longer string'
>>> code = """
...         int l = a.length();
...         return_val = Py::new_reference_to(Py::Int(l));
...         """
>>> inline(code,['a'])
15
```

Notice this time, `inline()` did not recompile the code because it found the compiled function in the persistent catalog of functions. There is a short pause as it looks up and loads the function, but it is much shorter than compiling would require.

You can specify the local and global dictionaries if you'd like (much like `exec` or `eval()` in Python), but if they aren't specified, the "expected" ones are used – i.e. the ones from the function that called `inline()`. This is accomplished through a little call frame trickery. Here is an example where the local_dict is specified using the same code example from above:

>>>

```
>>> a = 'a longer string'
>>> b = 'an even  longer string'
>>> my_dict = {'a':b}
>>> inline(code,['a'])
15
>>> inline(code,['a'],my_dict)
21
```

Every time the `code` is changed, `inline` does a recompile. However, changing any of the other options in inline does not force a recompile. The `force` option was added so that one could force a recompile when tinkering with other variables. In practice, it is just as easy to change the `code` by a single character (like adding a space some place) to force the recompile.

Note:
It also might be nice to add some methods for purging the cache and on disk catalogs.

I use `verbose` sometimes for debugging. When set to 2, it'll output all the information (including the name of the .cpp file) that you'd expect from running a make file. This is nice if you need to examine the generated code to see where things are going haywire. Note that error messages from failed compiles are printed to the screen even if `verbose` is set to 0.

The following example demonstrates using gcc instead of the standard msvc compiler on windows using same code fragment as above. Because the example has already been compiled, the `force=1` flag is needed to make `inline()` ignore the previously compiled version and recompile using gcc. The verbose flag is added to show what is printed out:

```
>>>inline(code,['a'],compiler='gcc',verbose=2,force=1)
running build_ext
building 'sc_86e98826b65b047ffd2cd5f479c627f13' extension
c:\gcc-2.95.2\bin\g++.exe -mno-cygwin -mdll -O2 -w -Wstrict-prototypes -IC:
\home\ej\wrk\scipy\weave -IC:\Python21\Include -c C:\DOCUME~1\eric\LOCAL
S~1\Temp\python21_compiled\sc_86e98826b65b047ffd2cd5f479c627f13.cpp
-o C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_86e98826
b65b04ffd2cd5f479c627f13.o
skipping C:\home\ej\wrk\scipy\weave\CXX\cxxextensions.c
(C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\cxxextension
s.o up-to-date)
skipping C:\home\ej\wrk\scipy\weave\CXX\cxxsupport.cxx
(C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\cxxsupport.o
 up-to-date)
skipping C:\home\ej\wrk\scipy\weave\CXX\IndirectPythonInterface.cxx
(C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\indirectpytho
ninterface.o up-to-date)
skipping C:\home\ej\wrk\scipy\weave\CXX\cxx_extensions.cxx
(C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\cxx_extension
s.o
up-to-date)
writing C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_86e
98826b65b047ffd2cd5f479c627f13.def
c:\gcc-2.95.2\bin\dllwrap.exe --driver-name g++ -mno-cygwin
-mdll -static --output-lib
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\libsc_86e98826
b65b047ffd2cd5f479c627f13.a --def
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_86e98826b65
b047ffd2cd5f479c627f13.def
-sC:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\sc_86e98826b
65b047ffd2cd5f479c627f13.o
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\cxxextensions.
o
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\cxxsupport.o
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\indirectpython
interface.o
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\temp\Release\cxx_extension
s.o -LC:\Python21\libs
-lpython21 -o
C:\DOCUME~1\eric\LOCALS~1\Temp\python21_compiled\sc_86e98826b65b047ffd2cd5f4
79c627f13.pyd
15
```

That’s quite a bit of output. `verbose=1` just prints the compile time.

```
>>>inline(code,['a'],compiler='gcc',verbose=1,force=1)
Compiling code...
finished compiling (sec):  6.00800001621
15
```

Note:
I’ve only used the `compiler` option for switching between ‘msvc’ and ‘gcc’ on windows. It may have use on Unix also, but I don’t know yet.

The `support_code` argument is likely to be used a lot. It allows you to specify extra code fragments such as function, structure or class definitions that you want to use in the `code` string. Note that changes to `support_code` do *not* force a recompile. The catalog only relies on `code` (for performance reasons) to determine whether recompiling is necessary. So, if you make a change to support_code, you'll need to alter `code` in some way or use the `force` argument to get the code to recompile. I usually just add some innocuous whitespace to the end of one of the lines in `code` somewhere. Here's an example of defining a separate method for calculating the string length:

```
>>>
>>> from weave import inline
>>> a = 'a longer string'
>>> support_code = """
...                 PyObject* length(Py::String a)
...                 {
...                     int l = a.length();
...                     return Py::new_reference_to(Py::Int(l));
...                 }
...                 """
>>> inline("return_val = length(a);",['a'],
...         support_code = support_code)
15
```

`customize` is a left over from a previous way of specifying compiler options. It is a `custom_info` object that can specify quite a bit of information about how a file is compiled. These `info` objects are the standard way of defining compile information for type conversion classes. However, I don't think they are as handy here, especially since we've exposed all the keyword arguments that distutils can handle. Between these keywords, and the `support_code` option, I think `customize` may be obsolete. We'll see if anyone cares to use it. If not, it'll get axed in the next version.

The `type_factories` variable is important to people who want to customize the way arguments are converted from Python to C. We'll talk about this in the next chapter **xx** of this document when we discuss type conversions.

`auto_downcast` handles one of the big type conversion issues that is common when using NumPy arrays in conjunction with Python scalar values. If you have an array of single precision values and multiply that array by a Python scalar, the result is upcast to a double precision array because the scalar value is double precision. This is not usually the desired behavior because it can double your memory usage. `auto_downcast` goes some distance towards changing the casting precedence of arrays and scalars. If your only using single precision arrays, it will automatically downcast all scalar values from double to single precision when they are passed into the C++ code. This is the default behavior. If you want all values to keep there default type, set `auto_downcast` to 0.

Returning Values

Python variables in the local and global scope transfer seamlessly from Python into the C++ snippets. And, if `inline` were to completely live up to its name, any modifications to variables in the C++ code would be reflected in the Python variables when control was passed back to Python. For example, the desired behavior would be something like:

```
# THIS DOES NOT WORK
>>> a = 1
>>> weave.inline("a++;",['a'])
>>> a
2
```

Instead you get:

```
                                                                        >>>
>>> a = 1
>>> weave.inline("a++;",['a'])
>>> a
1
```

Variables are passed into C++ as if you are calling a Python function. Python's calling convention is sometimes called "pass by assignment". This means its as if a `c_a = a` assignment is made right before `inline` call is made and the `c_a` variable is used within the C++ code. Thus, any changes made to `c_a` are not reflected in Python's `a` variable. Things do get a little more confusing, however, when looking at variables with mutable types. Changes made in C++ to the contents of mutable types *are* reflected in the Python variables.

```
                                                                        >>>
>>> a= [1,2]
>>> weave.inline("PyList_SetItem(a.ptr(),0,PyInt_FromLong(3));",['a'])
>>> print a
[3, 2]
```

So modifications to the contents of mutable types in C++ are seen when control is returned to Python. Modifications to immutable types such as tuples, strings, and numbers do not alter the Python variables. If you need to make changes to an immutable variable, you'll need to assign the new value to the "magic" variable `return_val` in C++. This value is returned by the `inline()` function:

```
                                                                        >>>
>>> a = 1
>>> a = weave.inline("return_val = Py::new_reference_to(Py::Int(a+1));",
['a'])
>>> a
2
```

The `return_val` variable can also be used to return newly created values. This is possible by returning a tuple. The following trivial example illustrates how this can be done:

```
# python version
def multi_return():
    return 1, '2nd'

# C version.
def c_multi_return():
    code =  """
              py::tuple results(2);
              results[0] = 1;
              results[1] = "2nd";
              return_val = results;
            """
    return inline_tools.inline(code)
```

The example is available in `examples/tuple_return.py`. It also has the dubious honor of demonstrating how much `inline()` can slow things down. The C version here is about 7-10 times slower than the Python version. Of course, something so trivial has no reason to be written in C anyway.

The issue with `locals()`

`inline` passes the `locals()` and `globals()` dictionaries from Python into the C++ function from the calling function. It extracts the variables that are used in the C++ code from these dictionaries, converts then to C++ variables, and then calculates using them. It seems like it would be trivial, then, after the calculations were finished to then insert the new values back into the `locals()` and `globals()` dictionaries so that the modified values were reflected in Python. Unfortunately, as pointed out by the Python manual, the locals() dictionary is not writable.

I suspect `locals()` is not writable because there are some optimizations done to speed lookups of the local namespace. I'm guessing local lookups don't always look at a dictionary to find values. Can someone "in the know" confirm or correct this? Another thing I'd like to know is whether there is a way to write to the local namespace of another stack frame from C/C++. If so, it would be possible to have some clean up code in compiled functions that wrote final values of variables in C++ back to the correct Python stack frame. I think this goes a long way toward making `inline` truly live up to its name. I don't think we'll get to the point of creating variables in Python for variables created in C – although I suppose with a C/C++ parser you could do that also.

A quick look at the code

`weave` generates a C++ file holding an extension function for each `inline` code snippet. These file names are generated using from the SHA-256 signature of the code snippet and saved to a location specified by the PYTHONCOMPILED environment variable (discussed later). The cpp files are generally about 200-400 lines long and include quite a few functions to support type conversions, etc. However, the actual compiled function is pretty simple. Below is the familiar `printf` example:

```
>>>
>>> import weave
>>> a = 1
>>> weave.inline('printf("%d\\n",a);',['a'])
1
```

And here is the extension function generated by `inline`:

```
static PyObject* compiled_func(PyObject*self, PyObject* args)
{
    py::object return_val;
    int exception_occurred = 0;
    PyObject *py__locals = NULL;
    PyObject *py__globals = NULL;
    PyObject *py_a;
    py_a = NULL;

    if(!PyArg_ParseTuple(args,"OO:compiled_func",&py__locals,&py__globals))
        return NULL;
    try
    {
        PyObject* raw_locals = py_to_raw_dict(py__locals,"_locals");
        PyObject* raw_globals = py_to_raw_dict(py__globals,"_globals");
        /* argument conversion code */
        py_a = get_variable("a",raw_locals,raw_globals);
        int a = convert_to_int(py_a,"a");
        /* inline code */
        /* NDARRAY API VERSION 90907 */
        printf("%d\n",a);    /*I would like to fill in changed locals and gl
obals here...*/
    }
    catch(...)
    {
        return_val =  py::object();
        exception_occurred = 1;
    }
    /* cleanup code */
    if(!(PyObject*)return_val && !exception_occurred)
    {
        return_val = Py_None;
    }
    return return_val.disown();
}
```

Every inline function takes exactly two arguments – the local and global dictionaries for the current scope. All variable values are looked up out of these dictionaries. The lookups, along with all `inline` code execution, are done within a C++ `try` block. If the variables aren't found, or there is an error converting a Python variable to the appropriate type in C++, an exception is raised. The C++ exception is automatically converted to a Python exception by SCXX and returned to Python. The `py_to_int()` function illustrates how the conversions and exception handling works. py_to_int first checks that the given PyObject* pointer is not NULL and is a Python integer. If all is well, it calls the Python API to convert the value to an `int`. Otherwise, it calls `handle_bad_type()` which gathers information about what went wrong and then raises a SCXX TypeError which returns to Python as a TypeError.

```
int py_to_int(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyInt_Check(py_obj))
        handle_bad_type(py_obj,"int", name);
    return (int) PyInt_AsLong(py_obj);
}
```

```
void handle_bad_type(PyObject* py_obj, char* good_type, char* var_name)
{
    char msg[500];
    sprintf(msg,"received '%s' type instead of '%s' for variable '%s'",
            find_type(py_obj),good_type,var_name);
    throw Py::TypeError(msg);
}

char* find_type(PyObject* py_obj)
{
    if(py_obj == NULL) return "C NULL value";
    if(PyCallable_Check(py_obj)) return "callable";
    if(PyString_Check(py_obj)) return "string";
    if(PyInt_Check(py_obj)) return "int";
    if(PyFloat_Check(py_obj)) return "float";
    if(PyDict_Check(py_obj)) return "dict";
    if(PyList_Check(py_obj)) return "list";
    if(PyTuple_Check(py_obj)) return "tuple";
    if(PyFile_Check(py_obj)) return "file";
    if(PyModule_Check(py_obj)) return "module";

    //should probably do more interrogation (and thinking) on these.
    if(PyCallable_Check(py_obj) && PyInstance_Check(py_obj)) return "callabl
e";
    if(PyInstance_Check(py_obj)) return "instance";
    if(PyCallable_Check(py_obj)) return "callable";
    return "unknown type";
}
```

Since the `inline` is also executed within the `try/catch` block, you can use CXX exceptions within your code. It is usually a bad idea to directly `return` from your code, even if an error occurs. This skips the clean up section of the extension function. In this simple example, there isn't any clean up code, but in more complicated examples, there may be some reference counting that needs to be taken care of here on converted variables. To avoid this, either uses exceptions or set `return_val` to NULL and use `if/then's` to skip code after errors.

Technical Details

There are several main steps to using C/C++ code within Python:

1. Type conversion
2. Generating C/C++ code
3. Compile the code to an extension module
4. Catalog (and cache) the function for future use

Items 1 and 2 above are related, but most easily discussed separately. Type conversions are customizable by the user if needed. Understanding them is pretty important for anything beyond trivial uses of `inline`. Generating the C/C++ code is handled by `ext_function` and `ext_module` classes and . For the most part, compiling the code is handled by distutils. Some customizations were needed, but they were relatively minor and do not require changes to distutils itself. Cataloging is pretty simple in concept, but surprisingly required the most code to implement (and still likely needs some work). So, this section covers items 1 and 4 from the list. Item 2 is covered later in the chapter covering the `ext_tools` module, and distutils is covered by a completely separate document xxx.

Passing Variables in/out of the C/C++ code

Note:
Passing variables into the C code is pretty straight forward, but there are subtleties to how variable modifications in C are returned to Python. see Returning Values for a more thorough discussion of this issue.

Type Conversions

Note:
Maybe `xxx_converter` instead of `xxx_specification` is a more descriptive name. Might change in future version?

By default, `inline()` makes the following type conversions between Python and C++ types.

Default Data Type Conversions

Python	C++
int	int
float	double
complex	std::complex
string	py::string
list	py::list
dict	py::dict
tuple	py::tuple
file	FILE*
callable	py::object
instance	py::object
numpy.ndarray	PyArrayObject*
wxXXX	wxXXX*

The `Py::` namespace is defined by the SCXX library which has C++ class equivalents for many Python types. `std::` is the namespace of the standard library in C++.

Note:
- I haven't figured out how to handle `long int` yet (I think they are currently converted to int - - check this).
- Hopefully VTK will be added to the list soon

Python to C++ conversions fill in code in several locations in the generated `inline` extension function. Below is the basic template for the function. This is actually the exact code that is generated by calling `weave.inline("")`.

The `/* inline code */` section is filled with the code passed to the `inline()` function call. The `/*argument conversion code*/` and `/* cleanup code */` sections are filled with code that handles conversion from Python to C++ types and code that deallocates memory or manipulates reference counts before the function returns. The following sections demonstrate how these two areas are filled in by the default conversion methods. * Note: I'm not sure I have reference counting correct on a few of these. The only thing I increase/decrease the ref count on is NumPy arrays. If you see an issue, please let me know.

NumPy Argument Conversion

Integer, floating point, and complex arguments are handled in a very similar fashion. Consider the following inline function that has a single integer variable passed in:

```
>>> a = 1
>>> inline("",['a'])
```

The argument conversion code inserted for `a` is:

```
/* argument conversion code */
int a = py_to_int (get_variable("a",raw_locals,raw_globals),"a");
```

`get_variable()` reads the variable `a` from the local and global namespaces. `py_to_int()` has the following form:

```
static int py_to_int(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyInt_Check(py_obj))
        handle_bad_type(py_obj,"int", name);
    return (int) PyInt_AsLong(py_obj);
}
```

Similarly, the float and complex conversion routines look like:

```
static double py_to_float(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyFloat_Check(py_obj))
        handle_bad_type(py_obj,"float", name);
    return PyFloat_AsDouble(py_obj);
}

static std::complex py_to_complex(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyComplex_Check(py_obj))
        handle_bad_type(py_obj,"complex", name);
    return std::complex(PyComplex_RealAsDouble(py_obj),
                                PyComplex_ImagAsDouble(py_obj));
}
```

NumPy conversions do not require any clean up code.

String, List, Tuple, and Dictionary Conversion

Strings, Lists, Tuples and Dictionary conversions are all converted to SCXX types by default. For the following code,

```
>>> a = [1]
>>> inline("",['a'])
```

>>>

The argument conversion code inserted for `a` is:

```
/* argument conversion code */
Py::List a = py_to_list(get_variable("a",raw_locals,raw_globals),"a");
```

`get_variable()` reads the variable `a` from the local and global namespaces. `py_to_list()` and its friends have the following form:

```
static Py::List py_to_list(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyList_Check(py_obj))
        handle_bad_type(py_obj,"list", name);
    return Py::List(py_obj);
}

static Py::String py_to_string(PyObject* py_obj,char* name)
{
    if (!PyString_Check(py_obj))
        handle_bad_type(py_obj,"string", name);
    return Py::String(py_obj);
}

static Py::Dict py_to_dict(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyDict_Check(py_obj))
        handle_bad_type(py_obj,"dict", name);
    return Py::Dict(py_obj);
}

static Py::Tuple py_to_tuple(PyObject* py_obj,char* name)
{
    if (!py_obj || !PyTuple_Check(py_obj))
        handle_bad_type(py_obj,"tuple", name);
    return Py::Tuple(py_obj);
}
```

SCXX handles reference counts on for strings, lists, tuples, and dictionaries, so clean up code isn't necessary.

File Conversion

For the following code,

>>>

```
>>> a = open("bob",'w')
>>> inline("",['a'])
```

The argument conversion code is:

```
/* argument conversion code */
PyObject* py_a = get_variable("a",raw_locals,raw_globals);
FILE* a = py_to_file(py_a,"a");
```

`get_variable()` reads the variable `a` from the local and global namespaces. `py_to_file()` converts PyObject* to a FILE* and increments the reference count of the PyObject*:

```
FILE* py_to_file(PyObject* py_obj, char* name)
{
    if (!py_obj || !PyFile_Check(py_obj))
        handle_bad_type(py_obj,"file", name);

    Py_INCREF(py_obj);
    return PyFile_AsFile(py_obj);
}
```

Because the PyObject* was incremented, the clean up code needs to decrement the counter

```
/* cleanup code */
Py_XDECREF(py_a);
```

Its important to understand that file conversion only works on actual files – i.e. ones created using the `open()` command in Python. It does not support converting arbitrary objects that support the file interface into C `FILE*` pointers. This can affect many things. For example, in initial `printf()` examples, one might be tempted to solve the problem of C and Python IDE's (PythonWin, PyCrust, etc.) writing to different stdout and stderr by using `fprintf()` and passing in `sys.stdout` and `sys.stderr`. For example, instead of

```
>>> weave.inline('printf("hello\\n");')
```

You might try:

```
>>> buf = sys.stdout
>>> weave.inline('fprintf(buf,"hello\\n");',['buf'])
```

This will work as expected from a standard python interpreter, but in PythonWin, the following occurs:

```
>>> buf = sys.stdout
>>> weave.inline('fprintf(buf,"hello\\n");',['buf'])
Traceback (most recent call last):
    File "", line 1, in ?
    File "C:\Python21\weave\inline_tools.py", line 315, in inline
        auto_downcast = auto_downcast,
    File "C:\Python21\weave\inline_tools.py", line 386, in compile_function
        type_factories = type_factories)
    File "C:\Python21\weave\ext_tools.py", line 197, in __init__
        auto_downcast, type_factories)
    File "C:\Python21\weave\ext_tools.py", line 390, in assign_variable_type
s
        raise TypeError, format_error_msg(errors)
    TypeError: {'buf': "Unable to convert variable 'buf' to a C++ type."}
```

The traceback tells us that `inline()` was unable to convert 'buf' to a C++ type (If instance conversion was implemented, the error would have occurred at runtime instead). Why is this? Let's look at what the `buf` object really is:

```
>>> buf
pywin.framework.interact.InteractiveView instance at 00EAD014
```

PythonWin has reassigned `sys.stdout` to a special object that implements the Python file interface. This works great in Python, but since the special object doesn't have a FILE* pointer underlying it, `fprintf` doesn't know what to do with it (well this will be the problem when instance conversion is implemented...).

Callable, Instance, and Module Conversion

Note:
Need to look into how ref counts should be handled. Also, Instance and Module conversion are not currently implemented.

```
>>> def a():
    pass
>>> inline("",['a'])
```

Callable and instance variables are converted to PyObject*. Nothing is done to their reference counts.

```
/* argument conversion code */
PyObject* a = py_to_callable(get_variable("a",raw_locals,raw_globals),"a");
```

`get_variable()` reads the variable a from the local and global namespaces. The `py_to_callable()` and `py_to_instance()` don't currently increment the ref count.

```
PyObject* py_to_callable(PyObject* py_obj, char* name)
{
    if (!py_obj || !PyCallable_Check(py_obj))
        handle_bad_type(py_obj,"callable", name);
    return py_obj;
}

PyObject* py_to_instance(PyObject* py_obj, char* name)
{
    if (!py_obj || !PyFile_Check(py_obj))
        handle_bad_type(py_obj,"instance", name);
    return py_obj;
}
```

There is no cleanup code for callables, modules, or instances.

Customizing Conversions

Converting from Python to C++ types is handled by `xxx_specification` classes. A type specification class actually serve in two related but different roles. The first is in determining whether a Python variable that needs to be converted should be represented by the given class. The second is as a code generator that generates C++ code needed to convert from Python to C++ types for a specific variable.

When

```
>>> a = 1
>>> weave.inline('printf("%d",a);',['a'])
```

is called for the first time, the code snippet has to be compiled. In this process, the variable 'a' is tested against a list of type specifications (the default list is stored in weave/ext_tools.py). The *first* specification in the list is used to represent the variable.

Examples of `xxx_specification` are scattered throughout numerous "xxx_spec.py" files in the `weave` package. Closely related to the `xxx_specification` classes are `yyy_info` classes. These classes contain compiler, header, and support code information necessary for including a certain set of capabilities (such as blitz++ or CXX support) in a compiled module. `xxx_specification` classes have one or more `yyy_info` classes associated with them. If you'd like to define your own set of type specifications, the current best route is to examine some of the existing spec and info files. Maybe looking over sequence_spec.py and cxx_info.py are a good place to start. After defining specification classes, you'll need to pass them into `inline` using the `type_factories` argument. A lot of times you may just want to change how a specific variable type is represented. Say you'd rather have Python strings converted to `std::string` or maybe `char*` instead of using the CXX string object, but would like all other type conversions to have default behavior. This requires that a new specification class that handles strings is written and then prepended to a list of the default type specifications. Since it is closer to the front of the list, it effectively overrides the default string specification. The following code demonstrates how this is done: ...

The Catalog

`catalog.py` has a class called `catalog` that helps keep track of previously compiled functions. This prevents `inline()` and related functions from having to compile functions every time they are called. Instead, catalog will check an in memory cache to see if the function has already been loaded into python. If it hasn't, then it starts searching through persistent catalogs on disk to see if it finds an entry for the given function. By saving information about compiled functions to disk, it isn't necessary to re-compile functions every time you stop and restart the interpreter. Functions are compiled once and stored for future use.

When `inline(cpp_code)` is called the following things happen:

1. A fast local cache of functions is checked for the last function called for `cpp_code`. If an entry for `cpp_code` doesn't exist in the cache or the cached function call fails (perhaps because the function doesn't have compatible types) then the next step is to check the catalog.
2. The catalog class also keeps an in-memory cache with a list of all the functions compiled for `cpp_code`. If `cpp_code` has ever been called, then this cache will be present (loaded from disk). If the cache isn't present, then it is loaded from disk.

   If the cache is present, each function in the cache is called until one is found that was compiled for the correct argument types. If none of the functions work, a new function is compiled with the given argument types. This function is written to the on-disk catalog as well as into the in-memory cache.
3. When a lookup for `cpp_code` fails, the catalog looks through the on-disk function catalogs for the entries. The PYTHONCOMPILED variable determines where to search for these catalogs and in what order. If PYTHONCOMPILED is not present several platform dependent locations are searched. All functions found for `cpp_code` in the path are loaded into the in-memory cache with functions found earlier in the search path closer to the front of the call list.

   If the function isn't found in the on-disk catalog, then the function is compiled, written to the first writable directory in the PYTHONCOMPILED path, and also loaded into the in-memory cache.

Function Storage

Function caches are stored as dictionaries where the key is the entire C++ code string and the value is either a single function (as in the "level 1" cache) or a list of functions (as in the main catalog cache). On disk catalogs are stored in the same manor using standard Python shelves.

Early on, there was a question as to whether md5 checksums of the C++ code strings should be used instead of the actual code strings. I think this is the route inline Perl took. Some (admittedly quick) tests of the md5 vs. the entire string showed that using the entire string was at least a factor of 3 or 4 faster for Python. I think this is because it is more time consuming to compute the md5 value than it is to do look-ups of long strings in the dictionary. Look at the examples/md5_speed.py file for the test run.

Catalog search paths and the PYTHONCOMPILED variable

The default location for catalog files on Unix is ~/.pythonXX_compiled where XX is version of Python being used. If this directory doesn't exist, it is created the first time a catalog is used. The directory must be writable. If, for any reason it isn't, then the catalog attempts to create a directory based on your user id in the /tmp directory. The directory permissions are set so that only you have access to the directory. If this fails, I think you're out of luck. I don't think either of these should ever fail though. On Windows, a directory called pythonXX_compiled is created in the user's temporary directory.

The actual catalog file that lives in this directory is a Python shelf with a platform specific name such as “nt21compiled_catalog” so that multiple OSes can share the same file systems without trampling on each other. Along with the catalog file, the .cpp and .so or .pyd files created by inline will live in this directory. The catalog file simply contains keys which are the C++ code strings with values that are lists of functions. The function lists point at functions within these compiled modules. Each function in the lists executes the same C++ code string, but compiled for different input variables.

You can use the PYTHONCOMPILED environment variable to specify alternative locations for compiled functions. On Unix this is a colon (‘:’) separated list of directories. On windows, it is a (‘;’) separated list of directories. These directories will be searched prior to the default directory for a compiled function catalog. Also, the first writable directory in the list is where all new compiled function catalogs, .cpp and .so or .pyd files are written. Relative directory paths (‘.’ and ‘..’) should work fine in the PYTHONCOMPILED variable as should environment variables.

There is a “special” path variable called MODULE that can be placed in the PYTHONCOMPILED variable. It specifies that the compiled catalog should reside in the same directory as the module that called it. This is useful if an admin wants to build a lot of compiled functions during the build of a package and then install them in site-packages along with the package. User’s who specify MODULE in their PYTHONCOMPILED variable will have access to these compiled functions. Note, however, that if they call the function with a set of argument types that it hasn’t previously been built for, the new function will be stored in their default directory (or some other writable directory in the PYTHONCOMPILED path) because the user will not have write access to the site-packages directory.

An example of using the PYTHONCOMPILED path on bash follows:

```
PYTHONCOMPILED=MODULE:/some/path;export PYTHONCOMPILED;
```

If you are using python21 on linux, and the module bob.py in site-packages has a compiled function in it, then the catalog search order when calling that function for the first time in a python session would be:

```
/usr/lib/python21/site-packages/linuxpython_compiled
/some/path/linuxpython_compiled
~/.python21_compiled/linuxpython_compiled
```

The default location is always included in the search path.

Note:
hmmm. see a possible problem here. I should probably make a sub- directory such as /usr/lib/python21/site- packages/python21_compiled/linuxpython_compiled so that library files compiled with python21 are tried to link with python22 files in some strange scenarios. Need to check this.

The in-module cache (in `weave.inline_tools` reduces the overhead of calling inline functions by about a factor of 2. It can be reduced a little more for type loop calls where the same function is called over and over again if the cache was a single value instead of a dictionary, but the benefit is very small (less than 5%) and the utility is quite a bit less. So, we’ll stick with a dictionary as the cache.

Blitz

Note:
most of this section is lifted from old documentation. It should be pretty accurate, but there may be a few discrepancies.

`weave.blitz()` compiles NumPy Python expressions for fast execution. For most applications, compiled expressions should provide a factor of 2-10 speed-up over NumPy arrays. Using compiled expressions is meant to be as unobtrusive as possible and works much like pythons exec statement. As an example, the following code fragment takes a 5 point average of the 512x512 2d image, b, and stores it in array, a:

```
from scipy import *  # or from NumPy import *
a = ones((512,512), Float64)
b = ones((512,512), Float64)
# ...do some stuff to fill in b...
# now average
a[1:-1,1:-1] =  (b[1:-1,1:-1] + b[2:,1:-1] + b[:-2,1:-1] \
               + b[1:-1,2:] + b[1:-1,:-2]) / 5.
```

To compile the expression, convert the expression to a string by putting quotes around it and then use `weave.blitz`:

```
import weave
expr = "a[1:-1,1:-1] =  (b[1:-1,1:-1] + b[2:,1:-1] + b[:-2,1:-1]" \
                       "+ b[1:-1,2:] + b[1:-1,:-2]) / 5."
weave.blitz(expr)
```

The first time `weave.blitz` is run for a given expression and set of arguments, C++ code that accomplishes the exact same task as the Python expression is generated and compiled to an extension module. This can take up to a couple of minutes depending on the complexity of the function. Subsequent calls to the function are very fast. Furthermore, the generated module is saved between program executions so that the compilation is only done once for a given expression and associated set of array types. If the given expression is executed with a new set of array types, the code most be compiled again. This does not overwrite the previously compiled function – both of them are saved and available for execution.

The following table compares the run times for standard NumPy code and compiled code for the 5 point averaging.

Method Run Time (seconds) Standard NumPy 0.46349 blitz (1st time compiling) 78.95526 blitz (subsequent calls) 0.05843 (factor of 8 speedup)

These numbers are for a 512x512 double precision image run on a 400 MHz Celeron processor under RedHat Linux 6.2.

Because of the slow compile times, its probably most effective to develop algorithms as you usually do using the capabilities of scipy or the NumPy module. Once the algorithm is perfected, put quotes around it and execute it using `weave.blitz`. This provides the standard rapid prototyping strengths of Python and results in algorithms that run close to that of hand coded C or Fortran.

Requirements

Currently, the `weave.blitz` has only been tested under Linux with gcc-2.95-3 and on Windows with Mingw32 (2.95.2). Its compiler requirements are pretty heavy duty (see the blitz++ home page (http://www.oonumerics.org/blitz/)), so it won't work with just any compiler. Particularly MSVC++ isn't up to snuff. A number of other compilers such as KAI++ will also work, but my suspicions are that gcc will get the most use.

Limitations

1. Currently, `weave.blitz` handles all standard mathematical operators except for the ** power operator. The built-in trigonometric, log, floor/ceil, and fabs functions might work (but haven't been tested). It also handles all types of array indexing supported by the NumPy module.

numarray's NumPy compatible array indexing modes are likewise supported, but numarray's enhanced (array based) indexing modes are not supported.

`weave.blitz` does not currently support operations that use array broadcasting, nor have any of the special purpose functions in NumPy such as take, compress, etc. been implemented. Note that there are no obvious reasons why most of this functionality cannot be added to scipy.weave, so it will likely trickle into future versions. Using `slice()` objects directly instead of `start:stop:step` is also not supported.

2. Currently Python only works on expressions that include assignment such as

   ```
   >>> result = b + c + d
   ```

   This means that the result array must exist before calling `weave.blitz`. Future versions will allow the following:

   ```
   >>> result = weave.blitz_eval("b + c + d")
   ```

3. `weave.blitz` works best when algorithms can be expressed in a "vectorized" form. Algorithms that have a large number of if/thens and other conditions are better hand-written in C or Fortran. Further, the restrictions imposed by requiring vectorized expressions sometimes preclude the use of more efficient data structures or algorithms. For maximum speed in these cases, hand-coded C or Fortran code is the only way to go.

4. `weave.blitz` can produce different results than NumPy in certain situations. It can happen when the array receiving the results of a calculation is also used during the calculation. The NumPy behavior is to carry out the entire calculation on the right hand side of an equation and store it in a temporary array. This temporary array is assigned to the array on the left hand side of the equation. blitz, on the other hand, does a "running" calculation of the array elements assigning values from the right hand side to the elements on the left hand side immediately after they are calculated. Here is an example, provided by Prabhu Ramachandran, where this happens:

   ```
   # 4 point average.
   >>> expr = "u[1:-1, 1:-1] = (u[0:-2, 1:-1] + u[2:, 1:-1] + \
   ...                "u[1:-1,0:-2] + u[1:-1, 2:])*0.25"
   >>> u = zeros((5, 5), 'd'); u[0,:] = 100
   >>> exec (expr)
   >>> u
   array([[ 100.,  100.,  100.,  100.,  100.],
          [   0.,   25.,   25.,   25.,    0.],
          [   0.,    0.,    0.,    0.,    0.],
          [   0.,    0.,    0.,    0.,    0.],
          [   0.,    0.,    0.,    0.,    0.]])

   >>> u = zeros((5, 5), 'd'); u[0,:] = 100
   >>> weave.blitz (expr)
   >>> u
   array([[ 100.  ,  100.       ,  100.       ,  100.       , 100. ],
          [   0.  ,   25.       ,   31.25     ,   32.8125   , 0. ],
          [   0.  ,    6.25     ,    9.375    ,   10.546875 , 0. ],
          [   0.  ,    1.5625   ,    2.734375 ,    3.3203125, 0. ],
          [   0.  ,    0.       ,    0.       ,    0.       , 0. ]])
   ```

   You can prevent this behavior by using a temporary array.

```
>>> u = zeros((5, 5), 'd'); u[0,:] = 100
>>> temp = zeros((4, 4), 'd');
>>> expr = "temp = (u[0:-2, 1:-1] + u[2:, 1:-1] + "\
...         "u[1:-1,0:-2] + u[1:-1, 2:])*0.25;"\
...         "u[1:-1,1:-1] = temp"
>>> weave.blitz (expr)
>>> u
array([[ 100.,  100.,  100.,  100.,  100.],
       [   0.,   25.,   25.,   25.,    0.],
       [   0.,    0.,    0.,    0.,    0.],
       [   0.,    0.,    0.,    0.,    0.],
       [   0.,    0.,    0.,    0.,    0.]])
```

5. One other point deserves mention lest people be confused. `weave.blitz` is not a general purpose Python->C compiler. It only works for expressions that contain NumPy arrays and/or Python scalar values. This focused scope concentrates effort on the computationally intensive regions of the program and sidesteps the difficult issues associated with a general purpose Python->C compiler.

NumPy efficiency issues: What compilation buys you

Some might wonder why compiling NumPy expressions to C++ is beneficial since operations on NumPy array operations are already executed within C loops. The problem is that anything other than the simplest expression are executed in less than optimal fashion. Consider the following NumPy expression:

```
a = 1.2 * b + c * d
```

When NumPy calculates the value for the 2d array, `a`, it does the following steps:

```
temp1 = 1.2 * b
temp2 = c * d
a = temp1 + temp2
```

Two things to note. Since `c` is an (perhaps large) array, a large temporary array must be created to store the results of `1.2 * b`. The same is true for `temp2`. Allocation is slow. The second thing is that we have 3 loops executing, one to calculate `temp1`, one for `temp2` and one for adding them up. A C loop for the same problem might look like:

```
for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
        a[i,j] = 1.2 * b[i,j] + c[i,j] * d[i,j]
```

Here, the 3 loops have been fused into a single loop and there is no longer a need for a temporary array. This provides a significant speed improvement over the above example (write me and tell me what you get).

So, converting NumPy expressions into C/C++ loops that fuse the loops and eliminate temporary arrays can provide big gains. The goal, then, is to convert NumPy expression to C/C++ loops, compile them in an extension module, and then call the compiled extension function. The good news is that there is an obvious correspondence between the NumPy expression above and the C loop. The bad news is that NumPy is generally much more powerful than this simple example illustrates and handling all possible indexing possibilities results in loops that are less than straightforward to write. (Take a peek at NumPy for confirmation). Luckily, there are several available tools that simplify the process.

The Tools

`weave.blitz` relies heavily on several remarkable tools. On the Python side, the main facilitators are Jeremy Hylton's parser module and Travis Oliphant's NumPy module. On the compiled language side, Todd Veldhuizen's blitz++ array library, written in C++ (shhhh. don't tell David Beazley), does the heavy lifting. Don't assume that, because it's C++, it's much slower than C or Fortran. Blitz++ uses a jaw dropping array of template techniques (metaprogramming, template expression, etc) to convert innocent-looking and readable C++ expressions into to code that usually executes within a few percentage points of Fortran code for the same problem. This is good. Unfortunately all the template raz-ma-taz is very expensive to compile, so the 200 line extension modules often take 2 or more minutes to compile. This isn't so good. `weave.blitz` works to minimize this issue by remembering where compiled modules live and reusing them instead of re-compiling every time a program is re-run.

Parser

Tearing NumPy expressions apart, examining the pieces, and then rebuilding them as C++ (blitz) expressions requires a parser of some sort. I can imagine someone attacking this problem with regular expressions, but it'd likely be ugly and fragile. Amazingly, Python solves this problem for us. It actually exposes its parsing engine to the world through the `parser` module. The following fragment creates an Abstract Syntax Tree (AST) object for the expression and then converts to a (rather unpleasant looking) deeply nested list representation of the tree.

```
                                                                      >>>
>>> import parser
>>> import scipy.weave.misc
>>> ast = parser.suite("a = b * c + d")
>>> ast_list = ast.tolist()
>>> sym_list = scipy.weave.misc.translate_symbols(ast_list)
>>> pprint.pprint(sym_list)
['file_input',
 ['stmt',
  ['simple_stmt',
   ['small_stmt',
    ['expr_stmt',
     ['testlist',
      ['test',
       ['and_test',
        ['not_test',
         ['comparison',
          ['expr',
           ['xor_expr',
            ['and_expr',
             ['shift_expr',
              ['arith_expr',
               ['term',
                ['factor', ['power', ['atom', ['NAME', 'a']]]]]]]]]]]]]]],
     ['EQUAL', '='],
     ['testlist',
      ['test',
       ['and_test',
        ['not_test',
         ['comparison',
          ['expr',
           ['xor_expr',
            ['and_expr',
             ['shift_expr',
              ['arith_expr',
               ['term',
                ['factor', ['power', ['atom', ['NAME', 'b']]]],
                ['STAR', '*'],
                ['factor', ['power', ['atom', ['NAME', 'c']]]]],
               ['PLUS', '+'],
               ['term',
                ['factor', ['power', ['atom', ['NAME', 'd']]]]]]]]]]]]]]]]],
   ['NEWLINE', '']]],
 ['ENDMARKER', '']]
```

Despite its looks, with some tools developed by Jeremy H., it's possible to search these trees for specific patterns (sub-trees), extract the sub-tree, manipulate them converting python specific code fragments to blitz code fragments, and then re-insert it in the parse tree. The parser module documentation has some details on how to do this. Traversing the new blitzified tree, writing out the terminal symbols as you go, creates our new blitz++ expression string.

Blitz and NumPy

The other nice discovery in the project is that the data structure used for NumPy arrays and blitz arrays is nearly identical. NumPy stores "strides" as byte offsets and blitz stores them as element offsets, but other than that, they are the same. Further, most of the concept and capabilities of the two libraries are remarkably similar. It is satisfying that two completely different implementations solved the problem with similar basic architectures. It is also fortuitous. The work involved in converting NumPy expressions to blitz expressions was greatly diminished. As an example, consider the code for slicing an array in Python with a stride:

```
>>> a = b[0:4:2] + c                                                    >>>
>>> a
[0,2,4]
```

In Blitz it is as follows:

```
Array<2,int> b(10);
Array<2,int> c(3);
// ...
Array<2,int> a = b(Range(0,3,2)) + c;
```

Here the range object works exactly like Python slice objects with the exception that the top index (3) is inclusive where as Python's (4) is exclusive. Other differences include the type declarations in C++ and parentheses instead of brackets for indexing arrays. Currently, `weave.blitz` handles the inclusive/exclusive issue by subtracting one from upper indices during the translation. An alternative that is likely more robust/maintainable in the long run is to write a `PyRange` class that behaves like Python's `range`. This is likely very easy.

The stock blitz also doesn't handle negative indices in ranges. The current implementation of the `blitz()` has a partial solution to this problem. It calculates and index that starts with a '-' sign by subtracting it from the maximum index in the array so that:

```
                 upper index limit
                     /-----\
b[:-1] -> b(Range(0,Nb[0]-1-1))
```

This approach fails, however, when the top index is calculated from other values. In the following scenario, if `i+j` evaluates to a negative value, the compiled code will produce incorrect results and could even core-dump. Right now, all calculated indices are assumed to be positive.

```
b[:i-j] -> b(Range(0,i+j))
```

A solution is to calculate all indices up front using if/then to handle the +/- cases. This is a little work and results in more code, so it hasn't been done. I'm holding out to see if blitz++ can be modified to handle negative indexing, but haven't looked into how much effort is involved yet. While it needs fixin', I don't think there is a ton of code where this is an issue.

The actual translation of the Python expressions to blitz expressions is currently a two part process. First, all x:y:z slicing expression are removed from the AST, converted to slice(x,y,z) and re-inserted into the tree. Any math needed on these expressions (subtracting from the maximum index, etc.) are also performed here. _beg and _end are used as special variables that are defined as blitz::fromBegin and blitz::toEnd.

```
a[i+j:i+j+1,:] = b[2:3,:]
```

becomes a more verbose:

```
a[slice(i+j,i+j+1),slice(_beg,_end)] = b[slice(2,3),slice(_beg,_end)]
```

The second part does a simple string search/replace to convert to a blitz expression with the following translations:

```
slice(_beg,_end) -> _all  # not strictly needed, but cuts down on code.
slice            -> blitz::Range
[                -> (
]                -> )
_stp             -> 1
```

`_all` is defined in the compiled function as `blitz::Range.all()`. These translations could of course happen directly in the syntax tree. But the string replacement is slightly easier. Note that namespaces are maintained in the C++ code to lessen the likelihood of name clashes. Currently no effort is made to detect name clashes. A good rule of thumb is don't use values that start with '_' or 'py_' in compiled expressions and you'll be fine.

Type definitions and coercion

So far we've glossed over the dynamic vs. static typing issue between Python and C++. In Python, the type of value that a variable holds can change through the course of program execution. C/C++, on the other hand, forces you to declare the type of value a variables will hold prior at compile time. `weave.blitz` handles this issue by examining the types of the variables in the expression being executed, and compiling a function for those explicit types. For example:

```
a = ones((5,5),Float32)
b = ones((5,5),Float32)
weave.blitz("a = a + b")
```

When compiling this expression to C++, `weave.blitz` sees that the values for a and b in the local scope have type `Float32`, or 'float' on a 32 bit architecture. As a result, it compiles the function using the float type (no attempt has been made to deal with 64 bit issues).

What happens if you call a compiled function with array types that are different than the ones for which it was originally compiled? No biggie, you'll just have to wait on it to compile a new version for your new types. This doesn't overwrite the old functions, as they are still accessible. See the catalog section in the inline() documentation to see how this is handled. Suffice to say, the mechanism is transparent to the user and behaves like dynamic typing with the occasional wait for compiling newly typed functions.

When working with combined scalar/array operations, the type of the array is *always* used. This is similar to the savespace flag that was recently added to NumPy. This prevents issues with the following expression perhaps unexpectedly being calculated at a higher (more expensive) precision that can occur in Python:

```
>>>
>>> a = array((1,2,3),typecode = Float32)
>>> b = a * 2.1 # results in b being a Float64 array.
```

In this example,

```
>>>
>>> a = ones((5,5),Float32)
>>> b = ones((5,5),Float32)
>>> weave.blitz("b = a * 2.1")
```

the `2.1` is cast down to a `float` before carrying out the operation. If you really want to force the calculation to be a `double`, define `a` and `b` as `double` arrays.

One other point of note. Currently, you must include both the right hand side and left hand side (assignment side) of your equation in the compiled expression. Also, the array being assigned to must be created prior to calling `weave.blitz`. I'm pretty sure this is easily changed so that a compiled_eval expression can be defined, but no effort has been made to allocate new arrays (and discern their type) on the fly.

Cataloging Compiled Functions

See The Catalog section in the `weave.inline()` documentation.

Checking Array Sizes

Surprisingly, one of the big initial problems with compiled code was making sure all the arrays in an operation were of compatible type. The following case is trivially easy:

```
a = b + c
```

It only requires that arrays `a`, `b`, and `c` have the same shape. However, expressions like:

```
a[i+j:i+j+1,:] = b[2:3,:] + c
```

are not so trivial. Since slicing is involved, the size of the slices, not the input arrays, must be checked. Broadcasting complicates things further because arrays and slices with different dimensions and shapes may be compatible for math operations (broadcasting isn't yet supported by `weave.blitz`). Reductions have a similar effect as their results are different shapes than their input operand. The binary operators in NumPy compare the shapes of their two operands just before they operate on them. This is possible because NumPy treats each operation independently. The intermediate (temporary) arrays created during sub-operations in an expression are tested for the correct shape before they are combined by another operation. Because `weave.blitz` fuses all operations into a single loop, this isn't possible. The shape comparisons must be done and guaranteed compatible before evaluating the expression.

The solution chosen converts input arrays to "dummy arrays" that only represent the dimensions of the arrays, not the data. Binary operations on dummy arrays check that input array sizes are compatible and return a dummy array with the size correct size. Evaluating an expression of dummy arrays traces the changing array sizes through all operations and fails if incompatible array sizes are ever found.

The machinery for this is housed in `weave.size_check`. It basically involves writing a new class (dummy array) and overloading its math operators to calculate the new sizes correctly. All the code is in Python and there is a fair amount of logic (mainly to handle indexing and slicing) so the operation does impose some overhead. For large arrays (ie. 50x50x50), the overhead is negligible compared to evaluating the actual expression. For small arrays (ie. 16x16), the overhead imposed for checking the shapes with this method can cause the `weave.blitz` to be slower than evaluating the expression in Python.

What can be done to reduce the overhead? (1) The size checking code could be moved into C. This would likely remove most of the overhead penalty compared to NumPy (although there is also some calling overhead), but no effort has been made to do this. (2) You can also call `weave.blitz` with `check_size=0` and the size checking isn't done. However, if the sizes aren't compatible, it can cause a core-dump. So, foregoing size_checking isn't advisable until your code is well debugged.

Creating the Extension Module

`weave.blitz` uses the same machinery as `weave.inline` to build the extension module. The only difference is the code included in the function is automatically generated from the NumPy array expression instead of supplied by the user.

Extension Modules

`weave.inline` and `weave.blitz` are high level tools that generate extension modules automatically. Under the covers, they use several classes from `weave.ext_tools` to help generate the extension module. The main two classes are `ext_module` and `ext_function` (I'd like to add `ext_class` and `ext_method` also). These classes simplify the process of generating extension modules by handling most of the "boiler plate" code automatically.

Note:
`inline` actually sub-classes `weave.ext_tools.ext_function` to generate slightly different code than the standard `ext_function`. The main difference is that the standard class converts function arguments to C types, while inline always has two arguments, the local and global dicts, and the grabs the variables that need to be converted to C from these.

A Simple Example

The following simple example demonstrates how to build an extension module within a Python function:

```
# examples/increment_example.py
from weave import ext_tools

def build_increment_ext():
    """ Build a simple extension with functions that increment numbers.
        The extension will be built in the local directory.
    """
    mod = ext_tools.ext_module('increment_ext')

    a = 1 # effectively a type declaration for 'a' in the
          # following functions.

    ext_code = "return_val = Py::new_reference_to(Py::Int(a+1));"
    func = ext_tools.ext_function('increment',ext_code,['a'])
    mod.add_function(func)

    ext_code = "return_val = Py::new_reference_to(Py::Int(a+2));"
    func = ext_tools.ext_function('increment_by_2',ext_code,['a'])
    mod.add_function(func)

    mod.compile()
```

The function `build_increment_ext()` creates an extension module named `increment_ext` and compiles it to a shared library (.so or .pyd) that can be loaded into Python.. `increment_ext` contains two functions, `increment` and `increment_by_2`. The first line of `build_increment_ext()`,

> mod = ext_tools.ext_module('increment_ext')

creates an `ext_module` instance that is ready to have `ext_function` instances added to it. `ext_function` instances are created much with a calling convention similar to `weave.inline()`. The most common call includes a C/C++ code snippet and a list of the arguments for the function. The following:

```
ext_code = "return_val = Py::new_reference_to(Py::Int(a+1));"
func = ext_tools.ext_function('increment',ext_code,['a'])
```

creates a C/C++ extension function that is equivalent to the following Python function:

```
def increment(a):
    return a + 1
```

A second method is also added to the module and then,

```
mod.compile()
```

is called to build the extension module. By default, the module is created in the current working directory. This example is available in the `examples/increment_example.py` file found in the `weave` directory. At the bottom of the file in the module's "main" program, an attempt to import `increment_ext` without building it is made. If this fails (the module doesn't exist in the PYTHONPATH), the module is built by calling `build_increment_ext()`. This approach only takes the time-consuming (a few seconds for this example) process of building the module if it hasn't been built before.

```
if __name__ == "__main__":
    try:
        import increment_ext
    except ImportError:
        build_increment_ext()
        import increment_ext
    a = 1
    print 'a, a+1:', a, increment_ext.increment(a)
    print 'a, a+2:', a, increment_ext.increment_by_2(a)
```

Note:
If we were willing to always pay the penalty of building the C++ code for a module, we could store the SHA-256 checksum of the C++ code along with some information about the compiler, platform, etc. Then, `ext_module.compile()` could try importing the module before it actually compiles it, check the SHA-256 checksum and other meta-data in the imported module with the meta-data of the code it just produced and only compile the code if the module didn't exist or the meta-data didn't match. This would reduce the above code to:

```
if __name__ == "__main__":
    build_increment_ext()

    a = 1
    print 'a, a+1:', a, increment_ext.increment(a)
    print 'a, a+2:', a, increment_ext.increment_by_2(a)
```

Note:
There would always be the overhead of building the C++ code, but it would only actually compile the code once. You pay a little in overhead and get cleaner "import" code. Needs some thought.

If you run `increment_example.py` from the command line, you get the following:

```
[eric@n0]$ python increment_example.py
a, a+1: 1 2
a, a+2: 1 3
```

If the module didn't exist before it was run, the module is created. If it did exist, it is just imported and used.

Fibonacci Example

`examples/fibonacci.py` provides a little more complex example of how to use `ext_tools`. Fibonacci numbers are a series of numbers where each number in the series is the sum of the previous two: 1, 1, 2, 3, 5, 8, etc. Here, the first two numbers in the series are taken to be 1. One approach to calculating Fibonacci numbers uses recursive function calls. In Python, it might be written as:

```
def fib(a):
    if a <= 2:
        return 1
    else:
        return fib(a-2) + fib(a-1)
```

In C, the same function would look something like this:

```
int fib(int a)
{
    if(a <= 2)
        return 1;
    else
        return fib(a-2) + fib(a-1);
}
```

Recursion is much faster in C than in Python, so it would be beneficial to use the C version for fibonacci number calculations instead of the Python version. We need an extension function that calls this C function to do this. This is possible by including the above code snippet as “support code” and then calling it from the extension function. Support code snippets (usually structure definitions, helper functions and the like) are inserted into the extension module C/C++ file before the extension function code. Here is how to build the C version of the fibonacci number generator:

```
def build_fibonacci():
    """ Builds an extension module with fibonacci calculators.
    """
    mod = ext_tools.ext_module('fibonacci_ext')
    a = 1 # this is effectively a type declaration

    # recursive fibonacci in C
    fib_code = """
                  int fib1(int a)
                  {
                      if(a <= 2)
                          return 1;
                      else
                          return fib1(a-2) + fib1(a-1);
                  }
               """
    ext_code = """
                  int val = fib1(a);
                  return_val = Py::new_reference_to(Py::Int(val));
               """
    fib = ext_tools.ext_function('fib',ext_code,['a'])
    fib.customize.add_support_code(fib_code)
    mod.add_function(fib)

    mod.compile()
```

XXX More about custom_info, and what xxx_info instances are good for.

Note:
recursion is not the fastest way to calculate fibonacci numbers, but this approach serves nicely for this example.

Customizing Type Conversions – Type Factories

not written

Things I wish `weave` did

It is possible to get name clashes if you uses a variable name that is already defined in a header automatically included (such as `stdio.h`) For instance, if you try to pass in a variable named `stdout`, you'll get a cryptic error report due to the fact that `stdio.h` also defines the name. `weave` should probably try and handle this in some way. Other things...

Table Of Contents (../index.html)

- Weave (**`scipy.weave`**)
  - Outline
  - Introduction
  - Requirements
  - Installation
  - Testing
    - Testing Notes:
  - Benchmarks
  - Inline
    - More with printf
    - More examples
      - Binary search
      - Dictionary Sort
      - NumPy – cast/copy/transpose
      - wxPython
    - Keyword Option
    - Inline Arguments
    - Distutils keywords
      - Keyword Option Examples
      - Returning Values
        - The issue with **`locals()`**
      - A quick look at the code
    - Technical Details
    - Passing Variables in/out of the C/C++ code
    - Type Conversions
      - NumPy Argument Conversion
      - String, List, Tuple, and Dictionary Conversion
      - File Conversion
      - Callable, Instance, and Module Conversion
      - Customizing Conversions
    - The Catalog
      - Function Storage
      - Catalog search paths and the PYTHONCOMPILED variable
  - Blitz
    - Requirements
    - Limitations
    - NumPy efficiency issues: What compilation buys you
    - The Tools

        - Parser
        - Blitz and NumPy
    - Type definitions and coercion
    - Cataloging Compiled Functions
    - Checking Array Sizes
    - Creating the Extension Module
  - Extension Modules
    - A Simple Example
    - Fibonacci Example
  - Customizing Type Conversions – Type Factories
  - Things I wish **`weave`** did

Previous topic

File IO (**`scipy.io`**) (io.html)

Next topic

Contributing to SciPy (../hacking.html)