Mixed Python/C programming with Cython

Ben Dudson, 22nd September 2017

Cython

`http://cython.org/`

- Compiles Python code to native machine code
- Extends Python by adding type information, enabling optimisations
- Makes calling C or C++ code relatively straightforward
- Included with many python distributions (Anaconda, Enthought, Pythonxy, Sage)
- Already installed on most University of York Linux machines
- Can install using pip:

```
pip install --user Cython
```

Application

This code calls a C function defined in “idamclient.h” which takes a character string input, and returns an integer.

```
cdef extern from "idamclient.h":
    bint getIdamProperty(const char *property)

def getProperty(property):
    "Get a property for client/server behavior"
    return getIdamProperty(property)
```

- cdef to define the external function, specifying input and output types
- def to define a Python function which handles type conversion as needed and can be called from other python code

(see post `http://www-users.york.ac.uk/~bd512/2013/10/24/cython/` and code `https://github.com/bendudson/pyidam2`)

First example: C code

This example calls a C function which is defined in scalars.c

```c
double timesTwo(double in) {
  return 2.*in;
}
```

and a header file scalars.h

```c
double timesTwo(double in);
```

First example: Building C code

We need a library file: .a or .so on Linux

```
gcc -c scalars.c -o scalars.o
ar cr libscalars.a scalars.o
```

or in a makefile:

```
libscalars.a: scalars.o makefile
	ar cr $@ $^

%.o: %.c %.h makefile
	gcc -c $< -o $@
```

Note: name of library file should start with “lib”

First example: Cython code

Here called `cscalars.pyx`

- If in the same directory as the C code, don't give it the same name. Cython will generate `cscalars.c` from `scalars.pyx` so could overwrite your files
- Conventional to have a Python module starting with `c` or `_c` as a direct wrapper around the C functions, then a more "pythonic" interface as a separate module.

```
"""
Module of things
"""
cdef extern from "scalars.h":
    double timesTwo(double value) # C function which only Cython will see

def times_two(value):     # This is the function which Python will see
    "Multiplies value by 2"
    return timesTwo(value)
```

First example: Building Cython code

To build a Cython module you need a `setup.py`

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

setup(
    name = "Something",  # Not the name of the module

    cmdclass = {"build_ext":build_ext},  # magic

    ext_modules = [ Extension("mymodule",     # The name of the module
                              ["cscalars.pyx"],
                              libraries=["scalars"]) ] # <lib>scalars<.a>

)
```

First example: Building Cython code

Then to build the module in the current directory

```
CFLAGS="-I." LDFLAGS="-L." python setup.py build_ext -i
```

The `CFLAGS` and `LDFLAGS` variables specify where the ".h" and ".a" files are. May not be needed in this case (depending on paths), but doesn't hurt.

Or in a makefile...

```
all: libscalars.a cscalars.pyx setup.py makefile
    CFLAGS="-I." LDFLAGS="-L." python setup.py build_ext -i
```

First example: Using Cython code

Building this creates a ".so" file, `mymodule.cpython-36m-x86_64-linux-gnu.so` on my system. To use it:

```python
import mymodule
help(mymodule)
mymodule.times_two(3.2)
```

Output:

```
NAME
    mymodule - Module of things

FUNCTIONS
    times_two(...)
        Multiplies value by 2

Out: 6.4
```

Second example: Arrays

A more useful example is where we are operating on arrays of numbers

- Python manages its own memory: Don't mix C malloc/free or C++ new/delete with Python arrays, or pass arrays created in C to Python.
- The easiest way is to let Python handle creating arrays, and pass them to C
- If you don't know the size of the array needed for return data:
  - Create a separate function to first get the size
  - Allocate the data in Python
  - Then call C again to fill in the array values
- Note that multidimensional NumPy arrays are really 1D arrays of data in memory, so a 2D NumPy array is passed to C as double* not double**

Second example: C code

```c
int timesTwo(int length, double* in, double* out) {
  int i;
  for (i = 0; i < length; i++) {
    out[i] = 2.*in[i];
  }
  return 0;
}
```

Second example: Cython code

```
import numpy as np    # For the Python interface
cimport numpy as np   # For the C interface

cdef extern from "arrays.h":
    bint timesTwo(bint, double* inputarray, double* outputarray)

def times_two(value):
    "Multiplies value by 2"
    size = len(value)
    inputvalues = value.astype(np.float64) # double precision
    outputvalues = np.empty(size, dtype=np.float64) # Create array
    status = timesTwo(size,
                      <double*> np.PyArray_DATA(inputvalues),
                      <double*> np.PyArray_DATA(outputvalues))
    assert status == 0
    return outputvalues
```

Second example: Running

```python
import mymodule
import numpy as np

a = np.ones(5)

mymodule.times_two(a)

Out: array([ 2.,  2.,  2.,  2.])
```

Things to try

- Download the examples
- Extend the scalars code to pass in two scalars rather than one
- Extend the arrays code to do a matrix-vector multiply
- Try calling a library (e.g. FFTW, GSL), adding the library to link to `setup.py`