

Extensions in C and Fortran

Why?

- C and Fortran are compiled languages
 - Source code is translated to machine instructions by the compiler before you run.
 - Ex:
 - `gfortran -o mycode mycode.f90`
 - `gcc -o mycode mycode.c`
 - No interpreter
- Generally, there are two reasons for wanting to use a compiled language from python
 - Speed
 - Reusing existing code

Extension Methods

- C
 - **C-API**: Brute force use the python/NumPy headers and C-API to access your python data structures
 - **ctypes**: a module that allows you to easily call functions in shared libraries
 - **Cython**: superset of python that can convert python into compiled C code (a fork of the Pyrex language)
- Fortran
 - **f2py**: part of NumPy. Allows for easy calling of Fortran from python
- Not discussed
 - **weave**: part of SciPy. Allows for inlining of C code in python. Seems to be deprecated
 - **Forthon**: a separate effort from f2py—not clear what the differences are)

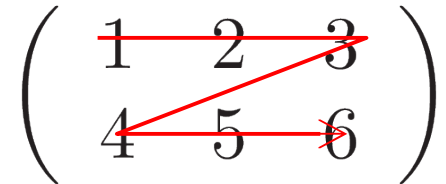
C-API

- The standard python interpreter (Cpython) is written in C.
 - You can access the API directly to include your own C functions.
- Pros
 - Very powerful
 - Underlies most (if not all) of the other techniques we'll see
- Cons
 - Harder to write, many need to change to keep up with API changes
 - Use requires compilation
 - Limited to specific python implementations (CPython, not PyPy)
- These days, there are better methods for most applications

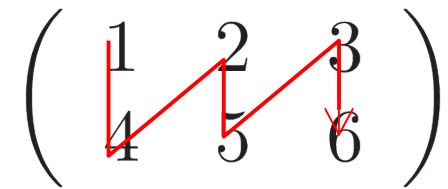
See http://scipy-lectures.github.io/advanced/interfacing_with_c/interfacing_with_c.html#id5

Arrays

- Row vs. Column major: $A(m,n)$
 - First index is called the row
 - Second index is called the column
 - Multi-dimensional arrays are flattened into a one-dimensional sequence for storage
 - Row-major (C, python): rows are stored one after the other
 - Column-major (Fortran, matlab): columns are stored one after the other
- Ordering matters for:
 - Passing arrays between languages
 - Deciding which index to loop over first
- f2py handles the translation automatically, if needed, but that may involve an implicit copy



Row major



Column major

C-API Example

```
/* Example of wrapping cos function from math.h with the Python-C-API. */
#include <Python.h>
#include <math.h>

/* wrapped cosine function */
static PyObject* cos_func(PyObject* self, PyObject* args)
{
    double value;
    double answer;
    /* parse the input, from python float to c double */
    if (!PyArg_ParseTuple(args, "d", &value))
        return NULL;
    /* if the above function returns -1, an appropriate Python exception will
     * have been set, and the function simply returns NULL
     */
    /* call cos from libm */
    answer = cos(value);
    /* construct the output from cos, from c double to python float */
    return Py_BuildValue("f", answer);
}

/* define functions in module */
static PyMethodDef CosMethods[] =
{
    {"cos_func", cos_func, METH_VARARGS, "evaluate the cosine"},
    {NULL, NULL, 0, NULL}
};

/* module initialization */
PyMODINIT_FUNC
initcos_module(void)
{
    (void) Py_InitModule("cos_module", CosMethods);
}
```

C-API Example

- Things to be careful about:
 - Multi-dimensional arrays in C require care
 - Standard approach for 2-d is to define a `double **` type and allocate a vector of pointers that point to the proper location in a 1-d contiguous memory block.
- Building:
 - Python has a `distutils` package that makes it easy to build the shared-object library

ctypes

- `ctypes` allows you to interface with an existing library.
 - You likely don't need to modify your C code
 - Simply need to define the interface to the C function in python
 - Support for NumPy through `numpy.ctypeslib`
- Pros:
 - You don't need to explicitly write the interfaces to your libraries
 - Part of standard python
- Cons:
 - Still need to compile a shared-object library

ctypes Example

Python:

```
import numpy as np
import ctypes as C

_cfunc = np.ctypeslib.load_library('cfunc', '.')
_cfunc.my_subroutine.restype = C.c_int
_cfunc.my_subroutine.argtypes = [C.POINTER(C.c_double), C.c_int]

def my_subroutine(A):
    return _cfunc.my_subroutine(A.ctypes.data_as(C.POINTER(C.c_double)), len(A))

A = np.arange(10, dtype=np.float64)
n = my_subroutine(A)
```

C:

```
#include <stdio.h>

int my_subroutine(double *array, int len) {

    printf("in my_subroutine\n");

    int i;
    for (i = 0; i < len; i++) {
        printf("%3d: %f\n", i, array[i]);
    }

    return 0;
}
```

f2py

- Very easy way to interface with Fortran
- Fortran and C use different orderings of array data

```
>>> a = numpy.zeros((6,6), dtype=numpy.float64)
>>> a.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

“Usually there is no need to worry about how the arrays are stored in memory and whether the wrapped functions, being either Fortran or C functions, assume one or another storage order. F2PY automatically ensures that wrapped functions get arguments with proper storage order; the corresponding algorithm is designed to make copies of arrays only when absolutely necessary. However, when dealing with very large multidimensional input arrays with sizes close to the size of the physical memory in your computer, then a care must be taken to use always proper-contiguous and proper type arguments.”

f2py

- You can initialize NumPy arrays in Fortran order from the start

```
>>> a = np.zeros( (4,6), dtype=np.float64, order="F")
>>> a.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

- f2py takes directives (as comments) to help decide what to do
 - For more advanced usage, you can manually create a signature file
 - To see the signature file from your Fortran routine, do:
f2py -h numpy_in_f.pyf numpy_in_f.f90

Cython

- *“The Cython language is a superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes.”*
- Cython has its origins in the pyrex project
- Code looks a lot like python
 - Declare the variable types with `cdef`—it looks like C declarations
 - Performance can be really great when you are dealing with cases where you need to explicitly write out loops over NumPy array indices
- Cython also has the ability to wrap C functions—see their documentation

Performance

- Let's look at performance of these different extensions.
- We'll follow the example from “*Speeding up Python (NumPy, Cython, and Weave)*” by Travis Oliphant
- Consider Laplace's equation:

$$\nabla^2 u = 0$$

- A simple discretization of this is:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = 0$$

- An we can solve this iteratively via the Gauss-Seidel method

$$u_{i,j} \leftarrow \frac{[u_{i+1,j} + u_{i-1,j}] \Delta y^2 + [u_{i,j+1} + u_{i,j-1}] \Delta x^2}{2 [\Delta x^2 + \Delta y^2]}$$

Performance

- Aside:
 - Gauss-Seidel makes use of the new data as soon as it is available.
 - There is another type of iteration called Jacobi iteration. The difference is that each zone is updated using only the old data:

$$u_{i,j}^{(k+1)} = \frac{\left[u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} \right] \Delta y^2 + \left[u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} \right] \Delta x^2}{2 [\Delta x^2 + \Delta y^2]}$$

- These methods have different convergence properties

Performance

- Results:

```
pure python: 10.9347379208 s
NumPy: 0.0756549835205 s
Cython: 0.0472810268402 s
f2py: 0.0393579006195 s
ctypes: 0.149159908295 s
C-API: 0.0446910858154 s
```

```
max diff NumPy: 0.0877975718815
max diff Cython: 0.0
max diff F90: 0.0
max diff ctypes: 0.0
max diff C-API: 0.0
```

- Notes:

- The pure python, NumPy, and Cython examples are identical to the code from the original comparison
 - The NumPy example there does Jacobi iteration instead of Gauss-Seidel
- 64 x 64 array for 1000 smoothings.
- For Fortran, we used `order='F'` in creating the array

Calling External Programs

- What about simply calling an external program?
 - Older way: `os.system()`
 - `subprocess` is the modern replacement (see: <https://docs.python.org/2/library/subprocess.html#subprocess-replacements>)
- `os` provides various routines to interact with the system
 - Note: availability of some routines varies by OS.
- `shutil` allows you to copy files, etc.

Example: Capturing `stdout` & `stderr`

- Example: issue a `git` command and capture `stdout` and `stderr`
 - Yes, python does have a `git` module itself we could use instead, but we want to see how to capture output
- `subprocess.Popen()` does what we want
 - Look at code...

Managing Files

- `shutil` allows for file operations
 - <https://docs.python.org/2/library/shutil.html>