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
  - `Forthon`: a separate effort from f2py—not clear what the differences are)
  - `SWIG`: more support for C++ and interfaces to a lot of different languages
  - `CFFI`: a new alternative to ctypes that might eventually be included in the standard python modules. Supposedly faster.

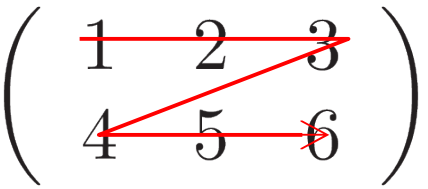
C-API

- 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, may 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

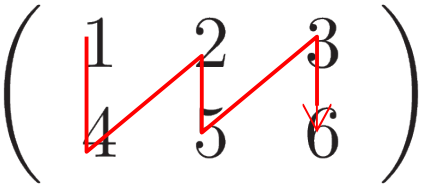
See http://scipy-lectures.github.io/advanced/interfacing_with_c/interfacing_with_c.html#id5

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

```
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION

#include <Python.h>
#include <numpy/arrayobject.h>
#include <math.h>

static PyObject* ex_function(PyObject* self, PyObject* args)
{

  PyArrayObject *iarray, *oarray;
  double **iA, **oA;
  int i, j, m, n, dims[2];

  if (!PyArg_ParseTuple(args, "O!", &PyArray_Type, &iarray)) return NULL;
  if (NULL == iarray) return NULL;

  if (PyArray_DTYPE(iarray)->type_num != NPY_DOUBLE ||
      PyArray_NDIM(iarray) != 2) {
    PyErr_SetString(PyExc_ValueError, "wrong input array type");
    return NULL;
  }

  n = dims[0] = PyArray_DIM(iarray, 0);
  m = dims[1] = PyArray_DIM(iarray, 1);

  oarray = (PyArrayObject *) PyArray_FromDims(2, dims, NPY_DOUBLE);

  /* change contigous arrays into C ** arrays */
  iA = (double **) malloc( (size_t) (n*sizeof(double)));
  for (i = 0; i < n; i++) {
    iA[i] = (double *) PyArray_DATA(iarray) + i*m;
  }

  oA = (double **) malloc( (size_t) (n*sizeof(double)));
  for (i = 0; i < n; i++) {
    oA[i] = (double *) PyArray_DATA(oarray) + i*m;
  }
```

Note: this example is for python 3.x—it will not work with python 2

C-API Example

```
  for (i = 0; i < n; i ++) {
    for (j = 0; j < m; j++) {
      oA[i][j] = iA[i][j]*iA[i][j];
    }
  }

  free (iA);
  free (oA);

  return PyArray_Return(oarray);
}


/* this is the table for function names that Python will see */
static PyMethodDef numpy_in_cMethods[] = {
  {"example", ex_function, METH_VARARGS,
   "a simple example: square the elements of an array"},
  {NULL, NULL}
};

static struct PyModuleDef moduledef = {
  PyModuleDef_HEAD_INIT,
  "numpy_in_c",  // name
   "a simple example: square the elements of an array",  // documentation
  -1,  // size
  numpy_in_cMethods,  // methods
};

/* this tells python what to do when it first imports this module --
   the name follows directly from the table name above */
PyMODINIT_FUNC PyInit_numpy_in_c(void) {
  PyObject *m;
  m = PyModule_Create(&moduledef);
  import_array();
  return m;
}
```

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
- Advice: avoid this
  - Using the C-API is usually unnecessary
  - It is also not portable to other python interpreters

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`
- Compile with `-fPIC` and create a shared library at linking
- 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 (although, there is a way to do this automagically in the python code)
  - The python ↔ C bridge can be slow (this is something CFFI might fix in the future)

ctypes

- NumPy provides an `ndpointer()` function to simplify passing NumPy arrays to C
  - It is more flexible than using `.data_as(POINTER(c_double))`, since you can check properties and add restrictions via flags

ctypes Example

Python:

```
import numpy as np
import numpy.ctypeslib as npc
import ctypes as C

cfunc = npc.load_library('cfunc', '.')
cfunc.my_subroutine.restype = C.c_int
cfunc.my_subroutine.argtypes = [npc.ndpointer(C.c_double, flags="C_CONTIGUOUS"),
                                C.c_int]

def my_subroutine(A):
    return cfunc.my_subroutine(A, len(A))

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

}
```

Note: this uses the NumPy `ndpointer()` to simplify the passing of the NumPy arrays. See https://docs.scipy.org/doc/numpy/reference/routines.ctypeslib.html

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."

http://docs.scipy.org/doc/numpy-dev/f2py/python-usage.html

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
```

http://docs.scipy.org/doc/numpy-dev/f2py/python-usage.html

f2py

- Example Fortran:

```
subroutine square(a, b, nx, ny)

  implicit none

  integer, intent(in) :: nx, ny
  double precision, intent(in) :: a(nx, ny)
  double precision, intent(out) :: b(nx, ny)

!f2py depend(nx, ny) :: a, b
!f2py intent(in) :: a
!f2py intent(out) :: b

! call this as b = numpy_in_f.square(a, nx, ny)

  integer :: i, j

  do j = 1, ny
     do i = 1, nx
        b(i,j) = a(i,j)**2
     enddo
  enddo

end subroutine square
```

This is all of the f2py markup

```
f2py3 --fcompiler=gnu95 -c numpy_in_f.f90 -m numpy_in_f
```

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

Cython

```
cimport numpy as np
import numpy as np

#cython: boundscheck=False
#cython: wraparound=False

def cy_square(np.ndarray[double, ndim=2] A):

    # in Cython, we should use cdef to declare the types of the variables
    # this helps with generating the C code
    cdef int nx = A.shape[0]
    cdef int ny = A.shape[1]

    # return array -- we allocate it much like in straight NumPy
    cdef np.ndarray B = np.zeros( (nx, ny), dtype=np.float64)

    cdef int i, j
    for i in range(nx):
        for j in range(ny):
            B[i,j] = A[i,j]**2

    return B
```

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

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

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:

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

  - 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/3/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/3/library/shutil.html