Combining Python with Fortran and C/C++

Numpy, F2py and Cython

InSuk Joung

School of Computational Sciences
Korea Institute for Advanced Study

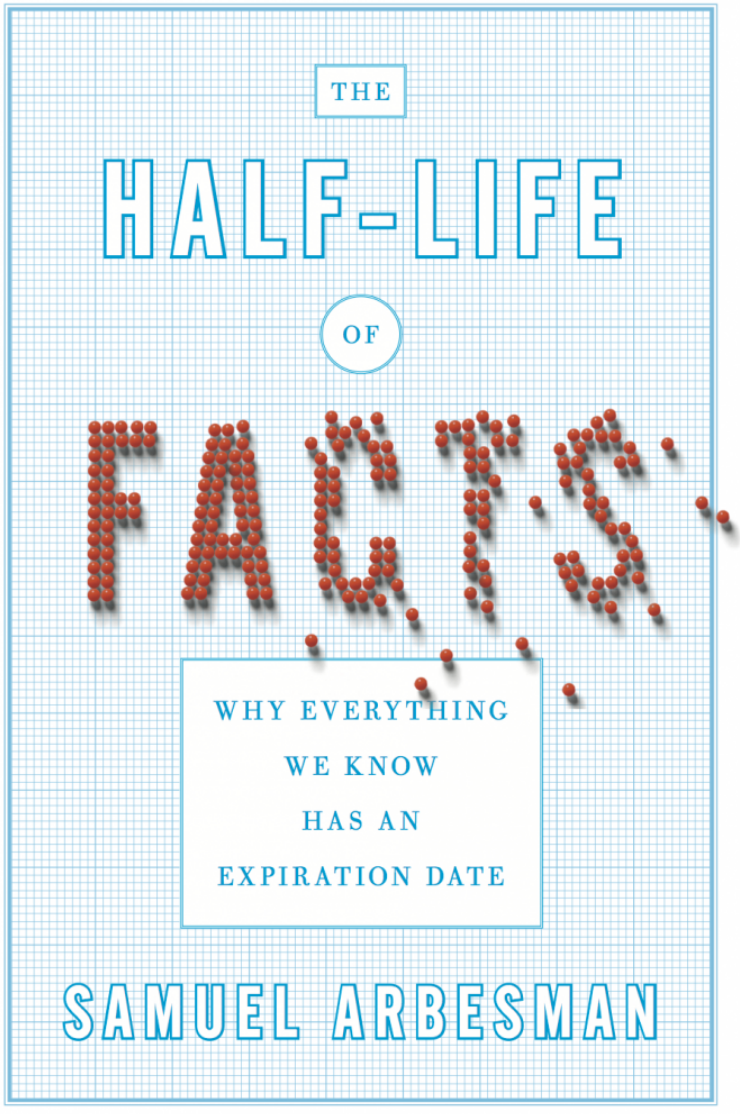
Summer School for Scientific Computing
Jul 09, 2015

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Python as a Glue Language

- Python is a mature scripting language.
- Python has a large user community.
- Program development using Python is 5-10 times faster than using C/C++.
- C-implementation of Python (CPython) provides API for inter-converting between Python objects and C types.

Fast Coding vs Fast Running



HALF-LIFE
OF
CODES?

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

To Speed Up Python Codes

- Use specialized modules: **numpy**, scipy and etc.
- Use pypy and numba
- Combine with Fortran/C/C++
  - Fortran: **f2py**
  - C/C++: **Cython**, ctypes, SWIG

Differences between Python and Fortran/C/C++

	Python	Fortran/C/C++
speed	slow	fast
execution	interpret	compile
memory management	dynamic allocation	footprint
memory allocation	garbage collected	user-responsiblity

Numpy

Broadcasting (Implicit Looping)

```
>>> a
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20]])
>>> b
array([0, 1, 2])
>>> a.shape (3, 3)
>>> b.shape (3,)
>>> a+b
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22]])
```

Examples

$(3,3)$ op (3): $(3,3)$ # (n) is $(1,n)$

$(m,n,1)$ op (n,p): (m,n,p) # (n,p) is $(1,n,p)$

$(a,1,b,1,c)$ op $(a,n,1,m,c)$: (a,n,b,m,c)

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Masking

```
>>> a = arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> mask = a>5
>>> mask
array([False, False, False, False, False, False,
True,  True,  True,  True], dtype=bool)
>>> a[mask]
array([6, 7, 8, 9])
```

Vectorizing

```
def myfunc(a,b):
    if a > b:
        return a*2.0
    else:
        return 0.0

def main():
    a = np.random.random((10,3))
    b = 0.5
    #print myfunc(a,b) # error

    vec_myfunc = np.vectorize(myfunc)

    print vec_myfunc(a,b)
```

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

F2py

Usage of f2py

Compile in one step.

```
$ f2py -c -m module_name fort01.f90 fort02.
    f90 ...
```

Generate a signature file.

```
$ f2py -h mysig.pyf -m module_name fort01.f90
     fort02.f90 ...
```

Compile a signature file and link Fortran object files.

```
$ f2py -c mysig.pyf fort01.o fort02.o ...
```

Generate a C wrapper using a signature file.

```
$ f2py mysig.pyf
```

General Compilation Process

1. Create object files from source files.
2. Link object files.
   - static link without `main` subroutine: archive (static library) file (*.a)
   - static link with `main` subroutine: (transferable) executable file
   - dynamic link without `main` subroutine: dynamic library file (*.so)
   - dynamic link with with `main` subroutine: (non-transferable) executable file

Compiler options

dyanmic compile: -fpic
static link: -static
dyanmic link: -shared

mysig.pyf

sig for sub01
sig for sub02
sig for sub03
...

wrapper.c (wrapper.o)

wrapper for sub01
wrapper for sub02
wrapper for sub03
...

fort01.f (fort01.o)

sub01
sub02

fort02.f (fort02.o)

sub03

fort03.f (fort03.o)

sub04
sub05

Link all the object files !

Signature File

```
python module wrap
    interface
        subroutine add_series(a,b,ret)
            integer intent(in) :: a
            integer intent(in) :: b
            integer intent(out) :: ret
        end subroutine add_series
    end interface
end python module wrap
```

Some Important Attributes

optional, intent(in), intent(out), intent(hide), depend

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Callback

1. Prepare a `*__user__*` module in the signature file.
2. Add the signature of the Python subroutine into the module.
3. In Fortran side, the function is passed as an argument (use `external` keyword).

F2py and Numpy

- Use `intent(hide)` if the argument is calculable using other arguments
- Use `optional` if the argument is optional. Default value should be given. (*Caveat*: The argument is moved to the end)
- Use `depend` if the argument is dependent on the other arguments.
- Use `check` if the argument needs to meet a special condition.

Wrapping C

add_series.c

```
#include "add_series.h"

void add_series(int a, int b, int *ret) {
    int i;
    ret[0] = 0;
    for (i = a; i <= b; i++) {
        ret[0] += i;
    }
}
```

wrap.pyf

```
python module wrap
 interface
  subroutine add_series(a,b,ret)
   intent(c) add_series
   intent(c)
   integer,intent(in):: a
   integer,intent(in):: b
   integer,intent(out),dimension(1):: ret
  end subroutine add_series
 end interface
end python module wrap
```

Return values should be (numpy) arrays.

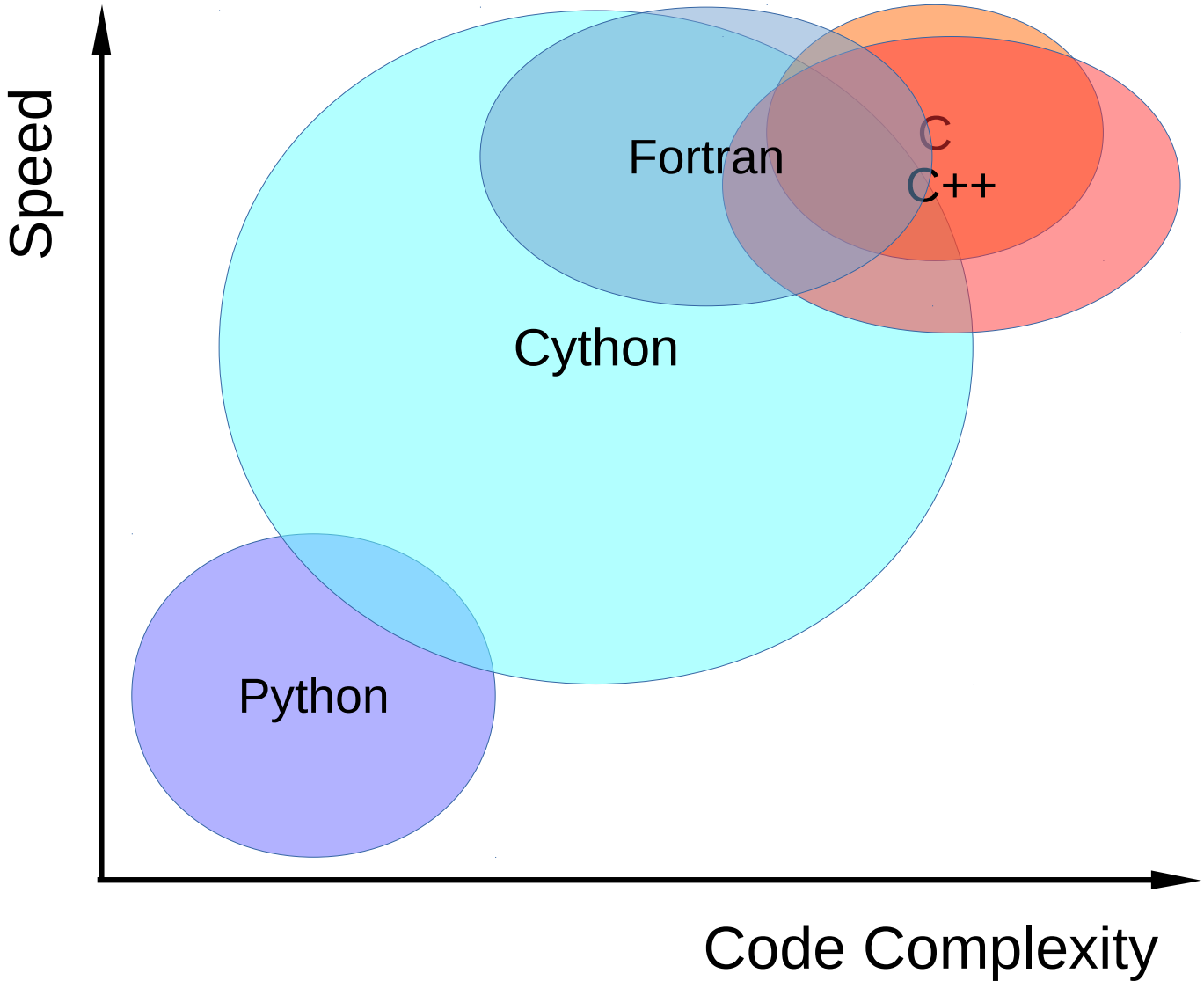
KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Cython

Cython

- Cython is a language.
- Cython is not CPython.
- Cython is a superset of Python
- Cython is a converter of Python code to C/C++ code

Complexity and Speed of Languages



KOREA INSTITUTE FOR ADVANCED STUDY

Simple Wrapping

wrap.pyx

```
cdef extern from "header.h":
    # C declarations
    cdef void hello_world()
    cdef int calc1(int a, int b)

# python wrappers that call
# the imported C functions
def py_hello_world():
    hello_world()

def py_calc1(a,b):
    return calc1(a,b)
```

Compilation

Use pyximport

```
import pyximport; pyximport.install()
```

Use Python distutils (setup.py)

```
from distutils.core import setup, Extension
from Cython.Build import cythonize
exts = cythonize([
    Extension(name="wrap",sources=["wrap.pyx"]),
])
setup(ext_modules = exts)
```

Compile manually

```
$ cython wrap.pyx
$ gcc -c -fpic wrap.c $(python-config --cflags) -o wrap.o
$ gcc -shared wrap.o c1.o c2.o $(python-config --ldflags) -o
    wrap.so
```

def / cdef / cpdef

- `def`
  - Python functions
  - returns a Python object
  - not accessible from C-side
- `cdef`
  - c functions
  - returns c types
  - not accessible from Python-side
- `cpdef`
  - c functions
  - returns c types
  - accessible from both sides, Python and C

Pointers

```
from cython.operator cimport dereference as deref

cdef int *a, *b
cdef int c
a = NULL # NULL pointer
c = 3
a = &c # address

# dereference
print a[0]
print deref(a)
```

Extra C types for Python

Static typing for speed! Use the following ***Python types*** *as well as C types for the static declaration of variables.*

- type, object
- bool
- complex
- basestring, str, unicode, bytes, bytearray
- list, tuple, dict, set, frozenset
- array
- slice
- date, time, datetime, timedelta, tzinfo

Type Conversion between Python and C

Python	⇒	C	⇒	Python
int, long		[unsigned] char [unsigned] short int, long		int
int, long		unsigned int unsigned long [unsigned] long long		long
bool		bint		bool
int, long, float		float, double, long double		float
str/bytes		char*		str/bytes
		struct		dict

Casting

```
cdef double real(int a):
    cdef double b
    b = <double> a
    return b
```

struct, enum and union

struct

```
cdef struct mystruct:
    int a
    double b
cdef mystruct var
var.a = 3
var.b = 0.7
# type conversion to dict
print var
```

enum

```
cdef enum fruit:
    apple
    orange
    grape
cdef fruit myfruit
myfruit = orange
# type conversion to long
print myfruit
```

union

```
cdef union myunion:
    int a
    double b
    char c

cdef myunion var
var.a = 1
# type conversion to dict
print var
var.b = 1.0
print var
var.c = 'a'
print var
```

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Type Extension (C vs C++)

C

```
from libc.stdlib cimport malloc, free

cdef extern from "shape.h":
    cdef struct _Circle:
        double x,y
        double r

    ctypedef _Circle Circle

    cdef double area(Circle *c)
    cdef double circumference(Circle *c)
    cdef void center(Circle *c, double *
         x, double *y)

cdef class circle:
    cdef Circle *_pt
    def __cinit__(self,x,y,r):
        self._pt = <Circle*> malloc(
             sizeof(Circle))
        self._pt.x = x
        self._pt.y = y
        self._pt.r = r

    def area(self):
        return area(self._pt)

    def __dealloc__(self):
        free(self._pt)
```

C++

```
cdef extern from "shape.h" namespace "
     shape":
    cdef cppclass Circle:
        Circle(double xin, double yin,
             double rin)
        double area()
        double circumference()
        void center(double *xout, double
             *yout)

cdef class circle:
    cdef Circle *_pt

    def __cinit__(self,x,y,r):
        self._pt = new Circle(x,y,r)

    def area(self):
        return self._pt.area()

    def __dealloc__(self):
        del self._pt
```

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Template and Type Conversion between Python and C++

```
cdef extern from "<vector>" namespace "std":
    cdef cppclass vector[T]:
        cppclass iterator:
            T operator*()
            iterator operator++()
            bint operator==(iterator)
            bint operator!=(iterator)
        vector()
        void push_back(T&)
        T& operator[](int)
        T& at(int)
        iterator begin()
        iterator end()
```

Python	⇒	C++	⇒	Python
bytes		std::string		bytes
iterable		std::vector		list
iterable		std::list		list
iterable		std::set		set
iterable (len 2)		std::pair		tuple (len 2)

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Calling Python Subroutines in C

pyend.pyx

```
cdef public void hello_world():
    print "Hello, I am Cython."
```

main.c

```
#include "Python.h"
#include "pyend.h"

int main(int argc, char **argv) {
    Py_Initialize();
    initpyend();   // initialize the cython module
    hello_world(); // call the cython function
    Py_Finalize();
    return 0;
};
```

Cython Standalone

main.pyx

```
cdef void hello_world():
    print "I am Cython!"

hello_world()
```

```
$ cython --embed main.pyx
```

Cython and Numpy

- `cimport numpy as np` in your pyx.
- Numpy array types
  - `np.ndarray` # *indexing is not efficient without template*
  - `np.ndarray [np.double_t, ndim=2]`
  - `np.ndarray [np.int_t, ndim=1]`
  - `np.ndarray [np.uint8, ndim=1, mode='c']`
- Numpy buffer can be converted to Memoryview.

```
cdef double [:,:] mview
cdef np.ndarray [np.double_t,ndim=2] narr
mview = narr # casting
```

- Slicing memoryview is generally faster than slicing ndarray.
- The indexing speed of memoryview and ndarray are similar.

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Wrapping Fortran

- Fortran accepts only C pointers as arguments
- Fortran subroutines are `void` type
- _ is added at the end of entity names (may use `-fno-underscoring`)
- Array is alway 1-dimensional Fortran-continuous. (no pointer-to-pointer)

a(3,2)-array

C-continuous	a(0,0)	a(0,1)	a(1,0)	a(1,1)	a(2,0)	a(2,2)
F-continuous	a(0,0)	a(1,0)	a(2,0)	a(0,1)	a(1,1)	a(2,1)

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Cython Compiler Directives

boundscheck (**True**/False)	embedsignature (True/**False**)
wraparound (**True**/False)	nonecheck(True/**False**)
profile (True/**False**)	initializedcheck(True/**False**)

On top of pyx files

```
# cython: boundscheck=True, wraparound=False
```

On top of subroutines

```
cimport cython
...
@cython.boundscheck(False)
@cython.wraparound(False)
cdef int csub(int a, int b):
    ...
```

In setup.py

```
exts = cythonize(
    [Extension(name="wrap",sources=["wrap.pyx"])],
    compiler_directives={'boundscheck':False}
)
```

KIAS KOREA INSTITUTE FOR ADVANCED STUDY

Program Development Process

1. Write Python codes.
2. Debug the program.
3. Profile the performance of the subroutines.
4. Cythonize subroutines or convert them into Fortran.
   - Statically type variables.
   - Replace python objects with C types.
5. Repeat 2-4 as desired.