

レガシー Fortran プログラムを 利用する Python プログラミング (2018年11月24日改訂)

本田 宏明^{1,2,3)}

¹⁾ ハイドロ総合技術研究所

²⁾ (前)九州大学 情報基盤研究開発センター

²⁾ (前) JST-CREST

環境設定、配布ソフトについて

1. **A1** 頁を確認下さい.
2. Bourne 系シェルであることを前提にしています.
A1 頁のセットアップ作業中に,
“**..profile**” を実行することを忘れないようにして下さい.

あらすじ

- Python 言語について
- Fortran 関数の呼び出し
- Fortran の fortファイル読み書き
 - Fortran の wrapper 関数を利用する方法
 - Python Binary 直接読み書きを利用する方法
- 付録
 - 配布ソフト

本資料の目的

1. 自身の Fortran, C プログラムを Python から利用できるようになる.
 1. (すみませんが) 基本的には Fortran77 を対象としています
 2. ctypes を利用する Fortran 関数ライブラリ呼び出し
 3. Fortran の unformatted ファイル、いわゆる fort ファイルの読み書き

□ 以下は自習にお任せします¹⁾

- 種々のエラーメッセージの理解
- モジュールと名前空間
- オブジェクト指向, Python でのクラスの取り扱い
- ctypes 以外の種々の Fortran, C コード呼び出し方法
- Cython を利用した高速計算
- インタラクティブモードや IPython での計算
- Python2 と Python3 の違い (本講習ソフトは 2.7 と 3.6 両方にて確認)

- 以下のリンクを参照下さい
- Numpy と ctypes の連携についての説明はあまり本には書いてなく、現状ではネットの方が良いかと思います。

1) <http://docs.python.jp/2.7/reference/index.html>
<http://docs.python.jp/3.6/reference/index.html>

1. Python 言語について

□ Python 言語について

- 汎用プログラミング言語
- 動的型付け, ガベージコレクション, オブジェクト指向
- ポータブル (Unix系, OSX, Windows)
- オープンソース
- FFI の一つである ctypes の方法により、容易に C や Fortran と連携可能
- 利点
 - コード可読性に配慮
 - インデントによるコードブロック形成
 - 豊富な標準ライブラリ
 - sys, re, subprocess, ...
 - 豊富なThird party によるライブラリ (機械学習などに頻繁に利用)
 - NumPy, SciPy, mpi4py, ...

課題1. Fortran 関数呼び出し

1.1 行列積計算

1.2 量子化学計算分子積分(応用課題)

Fortran 副プログラムの C からの利用

- Python から他のコンパイル型言語のライブラリを利用する際には、それぞれの関数について、C 言語からのインターフェースを基準にしている
- Fortran のインターフェースはそのままでは利用出来ない

```
subroutine bsortf( n, keys, values )  
integer n  
integer keys(n)  
double precision values(n)
```



C言語から呼び出す場合の関数インターフェース

```
void bsortf_( int *n, int *keys, double *values ) ;
```

注意

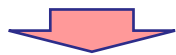
- Fortran の 関数の引数はいつでも参照型と考える
- subroutine は C からは void とみなす
- Fortran の 1, 2 次元配列はいつでも C からは 1 次元配列として利用する (と良い)
- 関数名も変化するため, nm コマンド等により調べる (と納得できる)

ctypes による外部プログラムの利用

ctypes は、C と互換性のあるデータ型を提供し
共有ライブラリ等の内部に定義された関数の呼び出しを可能とする方法
(モジュール) の1つ

Path to lib/libaaa.so 内で定義されている関数を利用する場合

```
返回值 function func1( arg1, arg2, ... )  
subroutine func2( arg1, arg2, ... )
```



基本はこれだけ

ctypes による func 呼び出し記述

```
import ctypes  
module_aaa = ctypes.CDLL( "Path_to_library/libaaa.so" )  
module_aaa.func1.restype = 戻り値の型  
module_aaa.func1.argtypes = [ 引数1の型, 引数2の型, ... ]  
...  
arg1 = ...  
arg2 = ...  
retval = module_aaa.func1( arg1, arg2, ... )
```

- **module_aaa** の
名称は自由に定める事が可能
- Subroutine の戻り値は void
- Fortran の場合は “_” の有無に
注意！！

課題1.1: 行列積計算

課題

Fortran77 で記述された、
2次元行列の行列積の計算ライブラリを Python から利用
関数の引数は以下の通り。

numpac の MULMVW と同様な関数¹⁾を想定

```
subroutine mmult( A, B, C, KA, KB, KC, L, M, N )  
integer KA, KB, KC, L, M, N  
double precision A( KA, M ), B( KB, N ), C( KC, N )
```

関数のライブラリ化

Fortran ライブラリ側で予め共有ライブラリ
libmmult.so を作成

```
$ make
gfortran -Wall -g -fPIC -I. -c mmult.f
gfortran -Wall -g -fPIC -Wl,-soname,libmmult.so -shared ¥
-o libmmult.so mmult.o
gfortran -Wall -g -fPIC -I. -c main_mmult.f
gfortran -Wall -g -fPIC -L. -lmmult -o test_mmult main_mmult.o
```

比較対象の Fortran テストプログラム:
test_mmult

共有ライブラリ中の各関数名を取得する

```
$ nm libmmult.so | grep mmult
0000000000000075c T mmult_
```



関数名は mmult_ と “_” が付いていることに注意

Python から利用する場合には、ライブラリ中の名前が必要となる

(ちょっと脱線)


コンパイルとライブラリ作成, 関数名調査

種々の言語で各プログラムをコンパイル後, 共有ライブラリを作成してみる

```
$ make
gcc -Wall -fPIC -g -I. -c sort_c.c
gfortran -Wall -fPIC -g -c sort_f.f
gfortran -Wall -fPIC -g -c sort_f95.f95
g++ -Wall -fPIC -g -I. -c sort_cpp.cpp
gcc -Wall -fPIC -g -shared -Wl,-soname,libsort.so -lgfortran -lstdc++ sort_c.o
sort_f.o sort_f95.o sort_cpp.o -o libsort.so (mac なら、*.dylib)
```

共有ライブラリ中の各関数名を取得する

```
$ nm libsort.so | grep sort
0000000000001954 T _Z14bsortcpp_arrayiPiPd
0000000000001b52 T _Z15bsortcpp_structiP11key_value_t
0000000000001523 T __mysort_MOD_bsorth95_array
0000000000001138 T __mysort_MOD_bsorth95_struct
00000000000009cc T bsorthc_array
0000000000000bc5 T bsorthc_struct
0000000000000d88 T bsorthf_
```



C++	array	関数
C++	struct	関数
F95	array	関数
F95	struct	関数
C	array	関数
C	struct	関数
F	array	関数

一般に, C 言語以外の関数は各コンパイラにより決められた規則により名前が変更されライブラリに保存される. Python から利用する場合には, ライブラリ中の名前が必要となる

Python 側の共有ライブラリ利用設定

ポイント：

1. ライブラリ名は **libmmult.so**
2. ライブラリ中の関数名は **mmult_** だった
3. 関数の 返り値と引数 は以下の通りだった

```
subroutine mmult( A, B, C, KA, KB, KC, L, M, N )  !: Fortran
```

```
void mmult_( double *A, double *B, double *C,  
             int *KA, int *KB, int *KC, int *L, int *M, int * N ) //: C
```

```
import ctypes  
from ctypes import *  
  
T_PTR_VOID    = c_void_p  
T_INT         = c_int32  
T_DOUBLE      = c_double  
T_PTR_INT     = POINTER( ctypes.c_int32 )  
T_PTR_DOUBLE  = POINTER( ctypes.c_double )
```

```
libmat2 = CDLL( "./libmmult.so" )  
libmat2.mmult_.restype = T_PTR_VOID  
libmat2.mmult_.argtypes = [ T_PTR_DOUBLE, T_PTR_DOUBLE, T_PTR_DOUBLE,  
                             T_PTR_INT, T_PTR_INT, T_PTR_INT,  
                             T_PTR_INT, T_PTR_INT, T_PTR_INT ]
```

2次元配列なので
本当はポインタのポインタとすべき！！
(ですが、単なるポインタでも動作します)

共有ライブラリ指定

返り値と
引数の指定

関数名指定

Python 側の共有ライブラリ関数利用

ポイント：

1. Numpy を試してみる（配列の Fortran 並びに注意）
2. Fortran 関数の引数へは参照を渡すので、引数の参照を準備する
3. 共有ライブラリ利用設定時のモジュール名と関数名を使用

#整数変数 ka, m,... の定義

ka = 15

...

ptr_ka = byref(c_int32(ka))

...

A = np.empty([ka, m], dtype = np.float64, order = 'F')

ptr_A = A.ctypes.data_as(T_PTR_DOUBLE)

#行列 A, B の定義

...

`libmat2.mmult_`(ptr_A, ptr_B, ptr_C,
ptr_ka, ptr_kb, ptr_kc, ptr_l, ptr_m, ptr_n)

整数変数を 32bit 変数として、
参照を取得

・Numpy 配列を Fortran
インデックスで生成、
・参照を取得

関数を呼び出す
引数はすべて参照型

`libmat2.mmult_` 関数名 とする

Python 側とライブラリ側の2次元配列の 組み合わせ (Numpy 利用時)

ポイント：

1. Numpy を生成する際には `order = 'F'` を記述
2. 2次元配列の行と列は、ライブラリ側と揃える
(Fortran で書いているように、Python 側で書けばいい)

#Numpy 空配列生成時

```
A = np.empty( [ ka, m ], dtype = np.float64, order = 'F' )
for i in range( 1 ):
    for j in range( m ):
        A[ i, j ] = ( (i+1) * 4.3893e0 + (j+1) * 3.930933e0 )
```

#ライブラリ側配列利用

```
subroutine mmult( A, B, C, KA, KB, KC, L, M, N )
double precision A( KA, M ),...
...
C ( i, j ) = C( i, j ) + A( i, k ) * B( k, j )
...
```

注意！！！（実験的に調べたところ、）

Numpy の `order='F'` は、メモリイメージまで Fortran には一致していない様子。
一致させるには、転置する必要がある（ここはちゃんと調べる必要あり）

応用課題1.2: 量子化学分子積分計算

課題

- 自身で作成した量子化学計算の分子積分ルーチンを呼び出す。
- 本資料は重なり積分について。関数は、以下に示すとおり、
init, (複数回利用される)calc, finalize に分かれている。
- プログラムライブラリとしては、運動エネルギー、各引力エネルギー、
電子反発積分が利用可能。
- オリジナル積分は C 言語で記述されているが、Fortran から呼び出し
可能なように、Fortran wrapper 関数が用意されている。

```
subroutine int1_s_init_f (  
&          numb_atom, numb_shell, numb_prim,  
&          shel_lqn,  shel_tem,  shel_atm,  shel_add,  
&          atom_zz,   atom_xyz,   prim_exp,  prim_coe,  
&          tol,       level_dbg_print )  
  subroutine int1_s_calc_f ( ish, jsh, inttype, nsize_int, Dint )  
  subroutine int1_s_print_f (ish, jsh, inttype, nsize_int, Dint )
```

関数のライブラリ化

Fortran ライブラリ側で予め共有ライブラリ
libmolint.so を作成

```
$ cd eri/molint make
...
gfortran -O2 -Wall -g -fPIC -I. -c int1_s_drv_f.f
...
gcc -O2 -Wall -g -fPIC -Wl,-soname,libmolint.so -shared ¥
    -o libmolint.so fmt.o int1_coefs.o ...
$ cd ../fc ; make
gfortran -Wall -g -fPIC -c main_intf.f
gfortran -Wall -g -fPIC -o test_intf main_intf.o -L../molint -lmolint
```

Fortran テストプログラム

共有ライブラリ中の各関数名を取得する

```
$ nm libmolint.so | grep int1_s_
000000000005dbd0 T int1_s_calc_f_
...
```



関数名は int1_s_init_f_, int1_s_calc_f_ 等と “_” が付いていることに注意

Python から利用する場合には、ライブラリ中の名前が必要となる

Python 側の共有ライブラリ利用設定

ポイント：

1. ラッピングする関数が多いため、wrapper.py に共有ライブラリの利用設定を記述
2. setlib でロードしたライブラリ変数を使用して、setfuncs 関数内で関数の利用設定

wrapper.py 内

```
# T_PTR_VOID, T_INT, T_DOUBLE, T_PTR_INT, T_PTR_DOUBLE の変数別名は mmult と同じ
```

```
def setlib( sofile ):  
    lib = CDLL( sofile )  
    return lib
```

共有ライブラリ指定関数

```
def setfuncs( lib ):  
    lib.int1_s_init_f_.restype = T_INT  
    lib.int1_s_init_f_.argtypes = [ T_PTR_INT, T_PTR_INT, T_PTR_INT, T_PTR_INT,  
                                     T_PTR_INT, T_PTR_INT, T_PTR_INT, T_PTR_INT,  
                                     T_PTR_DOUBLE, T_PTR_DOUBLE, T_PTR_DOUBLE, T_PTR_DOUBLE,  
                                     T_PTR_DOUBLE, T_PTR_INT ]
```

返り値と
引数の指定をする関数

lib 変数は上記 setlib 関数
での返り値の利用を想定

Python 側からの分子積分ライブラリ利用 (1/2)

ポイント：

1. Wrapper.py 内の setlib を利用して共有ライブラリをロード
2. setfuncs を利用して共有ライブラリにアクセスする変数 lib に分子積分関数の引数等を設定

eri.py 内

```
import ctypes
from ctypes import *
import numpy as np
import sys
from rhf_input import *
from wrapper import *

lib = setlib( "../molint/libmolint.so" )
setfuncs( lib )
```

ctypes, rhf_input, wrapper モジュール
の関数を、モジュール名なしで利用する

- ・共有ライブラリをロード、
lib 変数で利用可能
- ・lib に libmolint.so 内の
関数の関数名と引数等を設定

Python 側からの分子積分ライブラリ利用 (2/2)

ポイント：

1. Fortran の関数の引数はすべて参照呼び出しのため、Python 側でも関数に渡す変数への参照を用意する
2. Fortran の integer*4 への変数は 32 ビット変数にした後参照を取得

eri.py 内 (続き)

```
ptr_numb_atom = byref( c_int32( numb_atom ) )
...
ptr_shel_lqn = (shel_lqn).ctypes.data_as( T_PTR_INT )
...
shel_lqn = np.array( [0, 0, 0,... ,1 ], dtype = 'int32' )

retval = lib.int1_s_init_f_(
    ptr_numb_atom, ptr_numb_shell, ptr_numb_prim,
    ptr_shel_lqn, ptr_shel_tem, ptr_shel_atm, ptr_shel_add,
    ptr_atom_zz, ptr_atom_xyz, ptr_prim_exp, ptr_prim_coe,
    ptr_tol, ptr_lvl_dbg )
```

rhf_input.py 内で定義されている
numb_atom変数や shel_lqn 等の
Numpy 配列への参照変数を取得

32bits にした後参照取得

shel_lqn は int32 型の配列

参照変数を利用して関数呼出し

2. Fortran の fort ファイルの読み書き

2.1 Fortran 関数を利用

2.2 直接 Binary 読み書き

2.2.1 自身の作成したファイル

2.2.2 実際の AlchemyII プログラム

中間ファイルの読み書き（応用課題）

fort ファイルのフォーマット

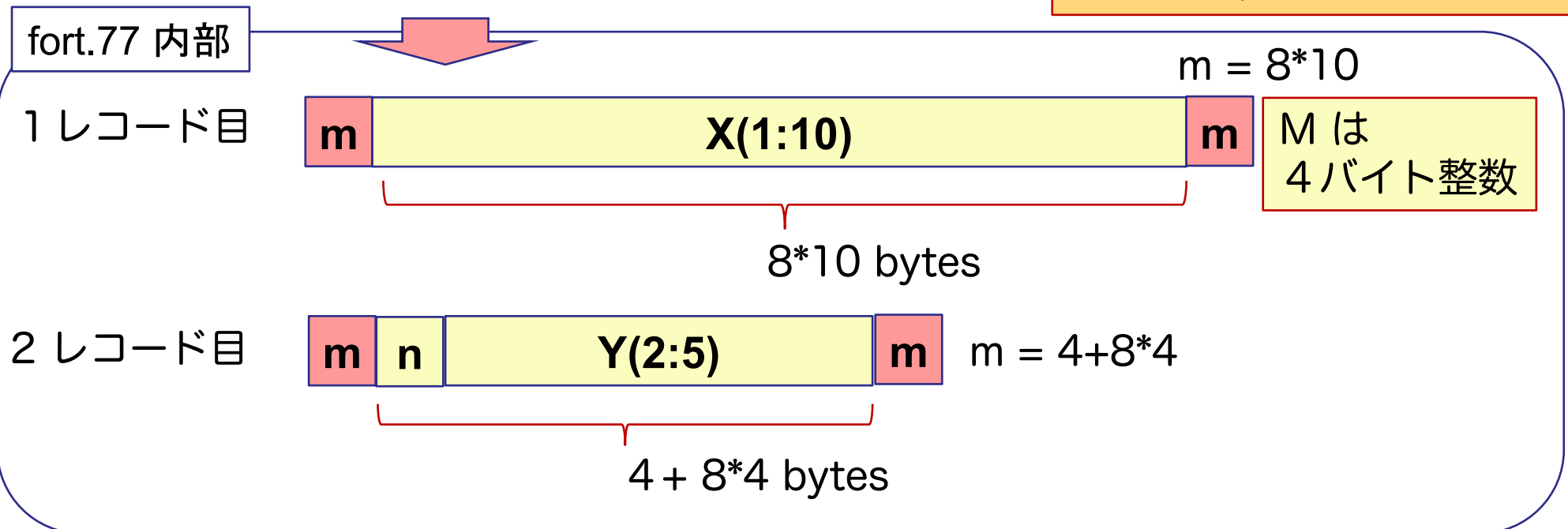
ポイント：

1. 1レコード内のコンテンツの前後にコンテンツのバイト数を示す、4バイト整数が挿入されている（これだけ）

```
integer i, n  
double precision X(10), Y(20)  
n = 5  
write(77) x  
write(77) n, (Y(i), i=2, n)
```

- 実際には全てのレコードは連続して配置
- 1レコードに複数変数を配置可能

fort.77 内部



課題2.1: Fortran 関数を経由した fort ファイルアクセス

解決方法 1

Fortran77 で記述された、
fort ファイルアクセス用関数を利用する
⇒ 課題 1.1、1.2 と同様の Fortran 関数呼び出しの
方法を利用すれば良い

readfort.f (元のコードを読んで自分でなんとか用意するしか、、、)

```
subroutine read_fort( iu, natom, zz, xyz, na, nb, amat )  
  integer nrow, ncol, nat  
  parameter( nrow = 200, ncol = 100, nat = 100 )  
  integer iu, i, idummy, j, na, nb, natom  
  real*8 zz(nat), xyz(3,nat), amat(nrow, ncol)
```

- 各配列の要素数はプログラム内の
配列サイズと一致している様にした方が良い

Fortran で fort ファイルの生成、読み出しを実行

```
$ make
gfortran -Wall -g -I. -fPIC -c readfort.f
gfortran -Wall -g -fPIC -Wl,-soname,libreadfort.so -lgfortran -lstdc++
-shared -o libreadfort.so readfort.o
gfortran -Wall -g -I. -fPIC -o test_writef test_write.f
gfortran -Wall -g -I. -fPIC -o test_readf test_read.f
gcc -Wall -g -I. -fPIC -o test_readc test_read.c
```

- libreadfort.so 中に Fortran バイナリファイル読み込み関数を準備
⇒ Python プログラムから利用
- test_writef にて fort.99 を生成
- デモで C 言語コードも用意

```
$ ./test_writef
$ ls fort.99
fort.99
$ ./test_readf
出力を確認
```

Python 側の共有ライブラリ利用設定

ポイント：

1. ライブラリ名は **libreadfort.so**
2. ライブラリ中の関数名は **read_fort_** だった
3. 関数の 返り値と引数 は以下の通りだった

```
subroutine read_fort( iu, natom, zz, xyz, na, nb, amat ) // F
```

```
void read_fort_( int *iu, int *natom, double *zz, double *xyz,  
                int *na, int *na, double *amat ) // C
```

read_viafunc.py 内

```
import ctypes  
from ctypes import *  
  
# T_PTR_VOID 等は mmult.f で設定された通り  
  
lib = CDLL( "./libreadfort.so" )  
lib.read_fort_.restype = T_PTR_VOID  
lib.read_fort_.argtypes = [ T_PTR_INT, T_PTR_INT, T_PTR_DOUBLE,  
                           T_PTR_DOUBLE, T_PTR_INT, T_PTR_INT, T_PTR_DOUBLE ]
```

amat は 2次元配列なので
本当はポインタのポインタとすべき！！
(ですが、単なるポインタでも動作します)

Python 側からの read_fort 関数利用(1/2)

ポイント：

1. 今回の関数の引数は全て intent(out) に対応
2. 整数については、32ビット変数の領域確保を予めしておく必要がある
3. Fortran の関数の引数はすべて参照呼び出しのため、Python 側でも関数に渡す変数への参照を用意する
4. Fortran の integer*4 への変数は 32ビット変数にした後参照を取得

shel_1c

read_viafunc.py 内 (続き)

```
iunit = 99
iunit32 = c_int32( iunit )
ptr_iunit = byref( c_int32( iunit ) )
natom32 = c_int32()
na32 = c_int32()
nb32 = c_int32()
ptr_natom = byref( natom32 )
ptr_na = byref( na32 )
ptr_nb = byref( nb32 )
```

32ビットの空変数と参照を用意

```
zz = np.empty( [ 100 ], dtype = np.float64 )
xyz = np.empty( [ 3, 100 ], dtype = np.float64, order = 'F' )
amat = np.empty( [ 200, 100 ], dtype = np.float64, order = 'F' )
ptr_zz = zz.ctypes.data_as( T_PTR_DOUBLE )
ptr_xyz = xyz.ctypes.data_as( T_PTR_DOUBLE )
ptr_amat = amat.ctypes.data_as( T_PTR_DOUBLE )
```

Numpy 配列についても参照を用意

Python 側からの read_fort 関数利用(2/2)

ポイント：

1. c_int32() で作成した変数は直接に Python の整数に代入できない
2. c_int32.value を利用して整数に代入する

shel_1c

read_viafunc.py 内 (続き)

```
lib.read_fort_( ptr_iunit, ptr_natom, ptr_zz, ptr_xyz,  
                ptr_na,    ptr_nb,    ptr_amat )
```

```
natom = natom32.value  
na     = na32.value  
nb     = nb32.value
```

**read_fort 関数により、
fort ファイルデータを所得**

**Ctypes c_int32() のデータの中身の
数値を取得**

課題2.2.1: 直接バイナリファイル アクセス（自身のファイル）

解決方法 2

Python のバイナリファイルアクセスの方法で
直接に fort ファイルを読み書きする。

その際、1 レコードをバッファに読み込んだ後の処理に
2 通り

1. 数バイトずつ読み込み

⇒ read_direct_struct.py スクリプト

2. 大きな配列については一度に読み込み

⇒ read_direct_frombuffer.py スクリプト

上記 2 の方法を利用した fort ファイルの生成

⇒ main_writefort.py スクリプト

バイナリファイルに対する入出力の基本

□ open

```
ifs = open(filename, 'rb' )  
iofs = open( filename, 'r+b' )
```

‘r’ は読み込み

‘b’ はバイナリ

‘w’ とすると書き出しのみ

□ read

```
bytebuf = ifs.read( nbyte )
```

この後、bytes 型の bytebuf から種々のデータを抽出

- ファイルからデータを読み込み
- バイト数指定を行う

□ write

種々のデータを一旦 bytes 型変数 bytebuf へパッキング後

```
iofs.write( bytebuf )
```

- ファイルへデータを書き込み
- bytes 型のオブジェクト自身がサイズ属性を持つため、正確に bytebuf のサイズ分のみの書き込みがなされる

□ close

上記を利用すれば、fort ファイルを1レコードずつ読み書きするプログラムを作成可能 (seek も利用できる)

Python からの fort レコード読み書き

▣ fort ファイルを 1 レコード分読む関数

```
def read_record( ifs ):  
    len_bin  = ifs.read( 4 )  
    len      = (struct.unpack( 'i', len_bin ))[0]  
    bytebuf  = ifs.read( len )  
    len_bin  = ifs.read( 4 )  
    len      = (struct.unpack( 'i', len_bin ))[0]  
    return bytebuf
```

レコード長
読み込み

実データ
読み込み

レコード長
終り読み込み

▣ fort ファイルを 1 レコード分書く関数

```
def write_record( ofs, bytebuf, nbyte ):  
    ofs.write( struct.pack( 'i', nbyte ) )  
    ofs.write( bytebuf )  
    ofs.write( struct.pack( 'i', nbyte ) )
```

レコード長
を書き込み

実データ
読み込み

レコード長
終り書き込み

bytes 型と通常のオブジェクトの変換(1/2)

▣ bytes ⇒ 通常オブジェクト

ctypes.struct.unpack_from() 関数を利用する

返り値は、第1引数で指定したフォーマットの数の分のタプル型データ

```
n = struct.unpack_from( 'i', bytebuf, offset )[0]
```

```
nab = struct.unpack_from( 'ii', bytebuf, offset )
```

```
na = nab[ 0 ]
```

```
nb = nab[ 1 ]
```

- ‘c’ で1文字型、‘i’ で整数型、‘d’ は 8 バイト浮動小数点型データの抽出ができる。
- 複数の型のデータを混ぜるとうまく動作していないように見える。(確認が必要)
- エンディアン指定も可能

▣ Bytes ⇒ numpy

Numpy の frombuffer() 関数を利用する (2次元にも利用可能)

```
nab = np.frombuffer( bytebuf, dtype='int32', count=2, offset=0 )
```

```
lmat = np.frombuffer( bytebuf, dtype = 'float64',  
                    count = 200 * 100, offset = 8 )
```

```
amat = np.reshape( lmat, [ 200, 100 ], order = 'F' )
```

bytes 型と通常のオブジェクトの変換(2/2)

□ 通常オブジェクト => bytes

ctypes.struct.pack() 関数を利用する (unpack_from と逆に動作)

```
bytebuf = bytes()
bytebuf += struct.pack( 'i', i )
bytebuf += struct.pack( 'dddd', zz[i], xyz[0,i], xyz[1,i ], xyz[2,i] )
↓ データをバイトデータにパッキングした後に、、
iofs.write(bytebuf)
```

□ Numpy ⇒ Bytes

Numpy の tobytes() 関数を利用する (全要素をバイトデータにパック)

```
bytebuf = bytes()
bytebuf += struct.pack( 'ii', na, nb)
bytebuf += (amat.transpose()).tobytes()
```

実験的にはそうだが、
正確には調べる必要あり

Numpyの2次元配列の order='F' のデータは実メモリ上では'C'のままなので、fort ファイルと同じ order でパッキングするために、transpose() が必要

応用課題2.2.2: Alchemyll fort.38 の読み書き

課題 (いずれも直接に binary ファイル読み書きを試みよう！！)

1.

Alchemyll プログラムの生成した fort.38 ファイル
(nso[isym] 個の基底による nmo[isym] 個のベクトルが
格納されている。今回は isym=1 のみ)
を、読み込んでみる。

2.

fort.38 の コピーの fort.39 の 6 ~ nmo[isym] のベクトルを、
単位ベクトルに変更して、fort.39 に書き戻してみる。

結局、もとの fort.38 の構造をオリジナルプログラムから
読み取れるかが問題で、技術的に難しい点はほとんどない

配布ソフト内のファイル： 1. ./alc/main_r38.py, 2. ./alc/main_rwr38.py

bytes 型と文字列の変換

▣ ホレリス定数の読み出し

- ホレリス定数も文字列と同じ取扱で読み込み可能

オリジナルプログラム

```
DIMENSION TITLE(18)
```

```
DIMENSION SYMINP(4)
```

```
DATA SYMINP/8H*****,8H          ,8H          ,8HSYMTRANS/
```

Unpack_from により、
144 文字のタプルが取得されるため、
文字列関数を利用して結合している

```
bytebuf = read_record( ifs )
title = ''.join(struct.unpack_from( '144c', bytebuf, 0 ) )

bytebuf = readrecord( ifs )
syminp = []
syminp.append( ''.join(struct.unpack_from('8c', bytebuf, 0)) )
syminp.append( ''.join(struct.unpack_from('8c', bytebuf, 8)) )
syminp.append( ''.join(struct.unpack_from('8c', bytebuf, 16)) )
syminp.append( ''.join(struct.unpack_from('8c', bytebuf, 24)) )
```

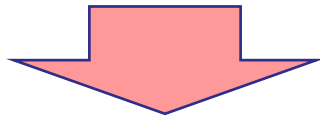
ベクトルの読み込みと書き込み

- [nso[isym], nmo[isym]] のサイズの 2 次元の Numpy 配列を対称性毎にリストとして保持する場合

```
amo = []
for i in range( nsym ):
    bytearray = read_record( ifs )
    buff = np.frombuffer( bytearray, dtype = 'float64', count = nso[ isym ] * nmo[ isym ],
                        offset = 4)

    buff_2d = np.reshape( buff, [ nso[ isym ], nmo[ isym ] ], order = 'F' )
    amo.append( buff_2d )
```

一旦 1 次元データとして読んで、
2 次元へ reshape



```
for isym in range( nsym ):
    bytearray = bytes()
    bytearray += struct.pack( 'i', isym )
    bytearray += amo[ isym ].transpose().tobytes()
    write_record( iofs, bytearray, 4 + 8 * nso[ isym ] * nmo[ isym ] )
```

書き込む際に
データを転置する

A1: 配布ソフトの実行方法

実行前に

```
$ cd ./tutorial_forpy  
$ . .profile
```

LD_LIBRARY_PATH を設定しています

課題 1.1

```
$ cd ./mmult  
$ make  
$ ./test_mmult  
$ python main_mmult.py
```

課題 1.2

```
$ cd ./eri/molint  
$ make  
$ cd ../fc  
$ make  
$ ./test_intf  
$ cd ../py  
$ python eri.py
```

課題2.1, 2.2.1

```
$ cd ./binary  
$ make  
$ ./test_writef  
$ ls fort.99  
fort.99  
$ ./test_readf  
$ python read_viafunc.py  
$ python read_direct_struct.py  
$ python read_direct_frombuffer.py
```

課題2.2.2

```
$ cd ./alc  
$ cp fort.38 fort.39  
$ python main_r38.py  
$ python main_rwr38.py  
$ # おまけ  
$ python read8.py
```