TUTORIAL -- INTERFACING FORTRAN AND PYTHON: USING CFFI, CTYPES AND CYTHON

HPC TRAINING EVENTS

DR. OSCAR FABIAN MOJICA LADINO
oscar.ladino@fieb.org.br

FORTRAN AND PYTHON, AT A GLANCE

Fortran at a glance

- A compiled, statically typed, parallel programming language
- Created by John Backus and his team at IBM in 1957
- Designed for scientific and engineering applications
- The oldest high-level programming language still in use 🦕
- Standardized since 1966
- In active development: Fortran 2018 released in November 2018, Fortran 202x is in the works

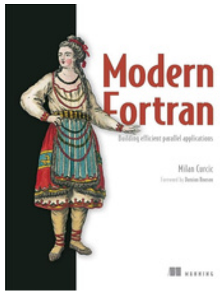
Python at a glance

- An interpreted, dynamically typed programming language
- Created by Guido van Rossum in 1993
- General purpose: Data analysis, web development, GUIs, real-time systems etc.
- Currently the fastest growing programming language 🚀
- In active development: Python 3.11.3 released in April 2023

TIOBE index

Key differences

- Compiled vs. interpreted
- Static, manifest vs. dynamic, inferred typing
- Focus on numerics and parallelism vs. rich standard library and ecosystem



COMPILED VS INTERPRETED

Building a Fortran executable

Building an executable program involves a few steps

Source code -> compiler -> linker -> executable

Alternatively, building a library is a tad simpler

Source code -> compiler -> library

Typical development flow

1. Write some code and save to a file
2. Does it compile? Yes, go to 3; no, go to 1
3. Does it run as expected? If no, go to 1

Compiler aids development

- Fortran code won't build until syntactically and semantically correct
- You'll spend a lot of time making the compiler happy
- Once it compiles, your program is more likely to be correct
- Now, all you gotta worry about is that your program runs correctly

Interpreter aids exploration and discovery

- Greater development productivity, especially early in the project
- Short feedback cycles -> rapid prototyping
- From idea to minimal working code in short time
- Great for testing ideas and quick experiments

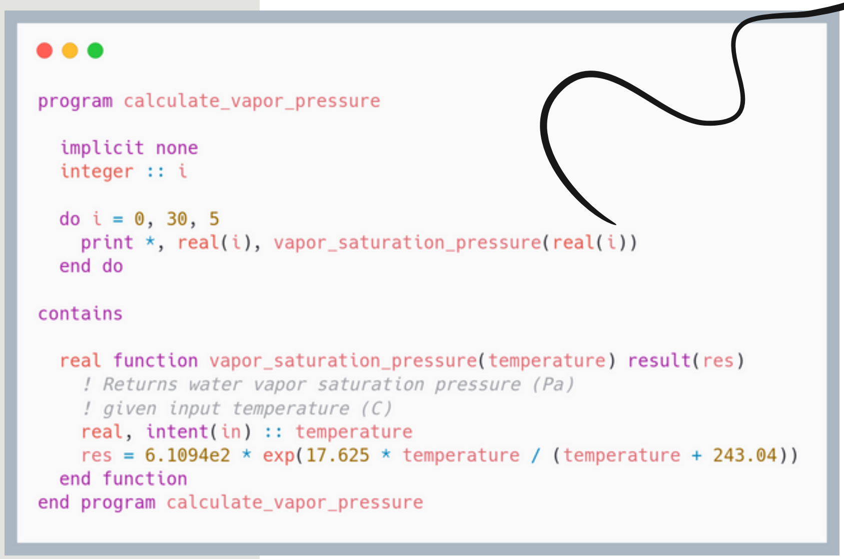
Fortran is compiled: Pros and cons

- Creates faster machine code 😀
- Slower development 😴
- Binaries specific to target machine ⚙️
- Modules often incompatible between:
  - compiler vendors
  - compiler versions of same vendor

TYPING

Fortran: Static, strong, manifest

Static typing
Types are checked at compile-time



```fortran
program calculate_vapor_pressure

  implicit none
  integer :: i

  do i = 0, 30, 5
    print *, real(i), vapor_saturation_pressure(real(i))
  end do

contains

  real function vapor_saturation_pressure(temperature) result(res)
    ! Returns water vapor saturation pressure (Pa)
    ! given input temperature (C)
    real, intent(in) :: temperature
    res = 6.1094e2 * exp(17.625 * temperature / (temperature + 243.04))
  end function
end program calculate_vapor_pressure
```

Let's compile and run it:

```
>nvfortran calculate_vapor_pressure.f90
>./a.out
  0.00000000     610.940002
  5.00000000     871.559570
  10.0000000     1226.02063
  15.0000000     1701.98303
  20.0000000     2333.44116
  25.0000000     3161.73633
  30.0000000     4236.65088
```

Strong typing: What happens if we call this function with a different type?

```fortran
do i = 0, 30, 5
   print *, real(i), vapor_saturation_pressure(i)
end do
```

Let's try to compile it again:

```
>nvfortran calculate_vapor_pressure.f90
calculate_vapor_pressure.f90:7:21:

    print *, real(i), vapor_saturation_pressure(i)
                    1
Error: Type mismatch in argument ‘temperature’ at (1); passed INTEGER(4) to REAL(4)
```

Static & strong typing

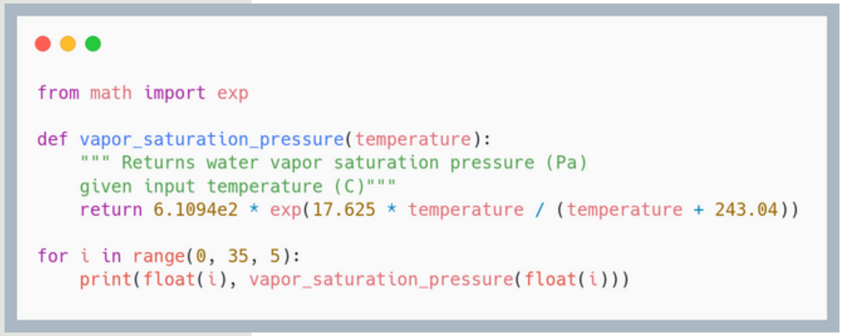
- Types are checked at compile-time, before execution
- Are argument types compatible at functions calls?
- Help catch bugs early & write correct programs

TYPING

Python: Dynamic, strong, inferred

Let's try the same in Python

Types are checked at run-time



```
from math import exp

def vapor_saturation_pressure(temperature):
    """ Returns water vapor saturation pressure (Pa)
    given input temperature (C)"""
    return 6.1094e2 * exp(17.625 * temperature / (temperature + 243.04))

for i in range(0, 35, 5):
    print(float(i), vapor_saturation_pressure(float(i)))
```

```
>python3 calculate_vapor_pressure.py
0.0 610.94
5.0 871.5595494938506
10.0 1226.020635023486
15.0 1701.9828147155868
20.0 2333.4406230993577
25.0 3161.7360356966915
30.0 4236.650251295475
```

Can we break it?

```
 for i in range(0, 35, 5):
    print(float(i), vapor_saturation_pressure(float(i)))

print('Will it take a string?', vapor_saturation_pressure('27'))
```

The error won't be triggered until the offending line is executed:

```
>python3 calculate_vapor_pressure.py
0.0 610.94
5.0 871.5595494938506
10.0 1226.020635023486
15.0 1701.9828147155868
20.0 2333.4406230993577
25.0 3161.7360356966915
30.0 4236.650251295475
Traceback (most recent call last):
  File "calculate_vapor_pressure.py", line 11, in <module>
    print('Will it take a string?', vapor_saturation_pressure('27'))
  File "calculate_vapor_pressure.py", line 6, in vapor_saturation_pressure
    return 6.1094e2 * exp(17.625 * temperature / (temperature + 243.04))
TypeError: can't multiply sequence by non-int of type 'float'
```

Python variables can also change types

A variable can freely change its type during the life of the program

```
>>> a = 2 # a is now an integer
>>> type(a)
<type 'int'>
>>> a = 3.141 # a is now a float
>>> type(a)
<type 'float'>
>>> a = 'spam, eggs, and ham' # a is now a string
>>> type(a)
<type 'str'>
```

Dynamic typing

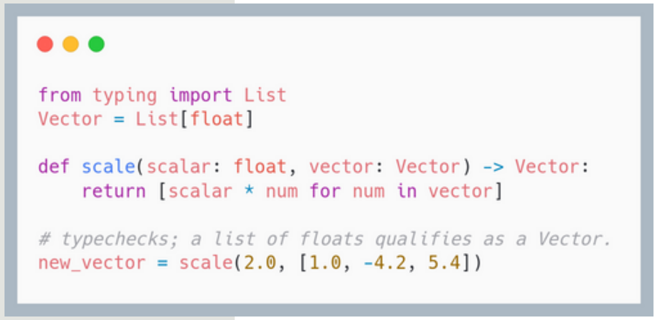
- Variable types are checked at run-time
- Allows for more flexible language design
- Prevents some optimizations
- Bugs can go unnoticed for a long time

TYPING

Python: Dynamic, strong, inferred

Enter Python type hints and mypy

Type hints introduced in Python 3.5 (typing module is built-in)
Example from the docs:

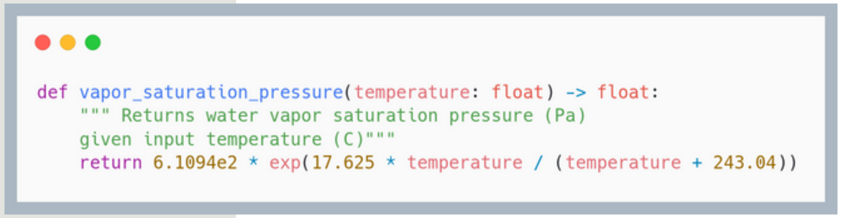


```python
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

mypy: Optional static typing for Python

```
>mypy --strict calculate_vapor_pressure.py
calculate_vapor_pressure.py:3: error: Function is missing a type annotation
calculate_vapor_pressure.py:9: error: Call to untyped function
```

Redefine our function with Type hints

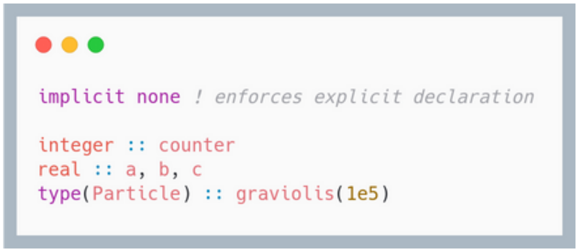


```python
def vapor_saturation_pressure(temperature: float) -> float:
    """ Returns water vapor saturation pressure (Pa)
    given input temperature (C)"""
    return 6.1094e2 * exp(17.625 * temperature / (temperature + 243.04))
```

```
>mypy --strict calculate_vapor_pressure_typed.py
```

Type hints + mypy yield development benefits of static typing without sacrificing flexibility

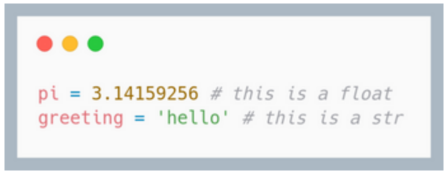
How about manifest typing?
Modern Fortran requires you to declare your variables before using them



```fortran
implicit none ! enforces explicit declaration

integer :: counter
real :: a, b, c
type(Particle) :: graviolis(1e5)
```

Useful as annotation and makes code easier to understand
More verbose -> slower development

The opposite is inferred typing
Python infers variable types on assignment



```python
pi = 3.14159256 # this is a float
greeting = 'hello' # this is a str
```

Less verbose -> faster development
Again, type annotations can help you

Type systems takeaways

- Static, manifest typing makes Fortran more rigid & verbose than Python
- A Fortran program that compiles is more likely to be correct during run-time
- A compromise between flexibility and type safety
- Python type hints + mypy can bring both static and manifest typing to the table **-> Best of both worlds!**

Fortran: It can solve some problems extremely well

- A small language that does a few things great
- Basically backwards compatible
- Several excellent compilers
- A treasure vault of battle-tested numerical libraries

- Arrays and whole-array arithmetic are like nothing else IMO
- Native, distributed parallelism with coarrays
- Math looks like math
- Small language that is easy to learn

VS

- Fortran carries a lot of baggage from its rich and long history
- Tedious and archaic I/O, weak string support
- Small standard library with focus on numerics
- No built-in collections beyond arrays
- No reference implementation

THEY WORK WELL TOGETHER

Let's talk about Python: It can solve almost every problem out there reasonably well

- Easy to get started with
- Comprehensive standard library: collections, algorithms, I/O ...
- Amazing library ecosystem
- Standard package management

- Extremely easy to set up development environment and install packages

```
python3 -m venv venv
source venv/bin/activate
pip install -U matplotlib numpy pandas
```

- Numeric and scientific computing: numpy, scipy, sympy, xarray...
- Visualization: matplotlib, plotly, bokeh...
- Tables and time series: pandas
- Machine Learning: scikit-learn, tensorflow, keras, pytorch

Python is a good solution for most problems

- Systems programming, real-time systems, networking
- Web development, Databases, Video games
- Numerical analysis, machine learning

WRITING A FORTRAN-C INTERFACE

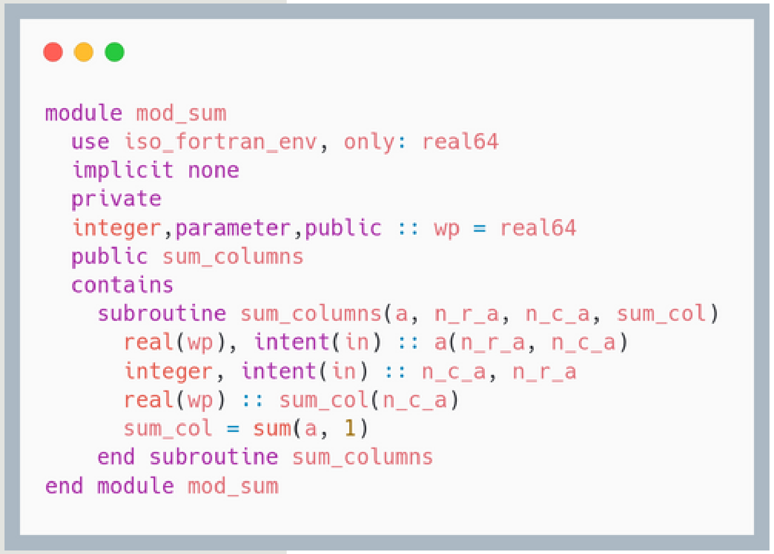
The reason why Fortran can be called from Python is the C interoperability features that were added with Fortran 2003 and expanded later. By writing our code to be interoperable with C, we can call procedures, module variables and derived types from C code. Many other programming languages, such as Python, then gain access to our Fortran code, as they are able to call C code.

C interoperable code
The two most important parts of making your Fortran code C interoperable are:

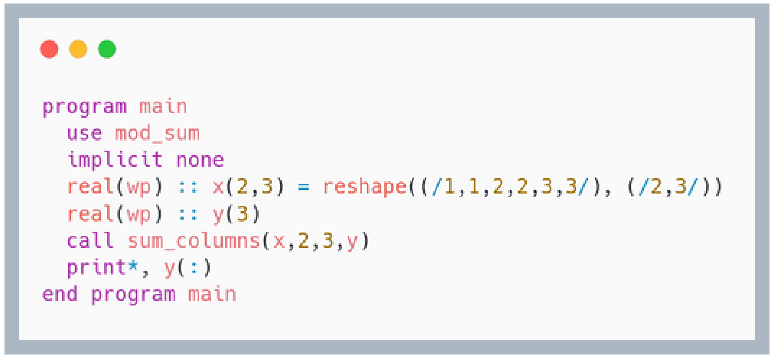
1. Using the bind(c) attribute
2. Using C interoperable types

bind(c) is built in, while the C interoperable types are available through the intrinsic module iso_c_binding that needs to be imported

There are several sites on the web with tables showing the interoperability between Fortran intrinsic types and C types (i.e. https://fortranwiki.org/fortran/show/iso_c_binding)



```fortran
module mod_sum
  use iso_fortran_env, only: real64
  implicit none
  private
  integer,parameter,public :: wp = real64
  public sum_columns
  contains
    subroutine sum_columns(a, n_r_a, n_c_a, sum_col)
      real(wp), intent(in) :: a(n_r_a, n_c_a)
      integer, intent(in) :: n_c_a, n_r_a
      real(wp) :: sum_col(n_c_a)
      sum_col = sum(a, 1)
    end subroutine sum_columns
end module mod_sum
```



```fortran
program main
  use mod_sum
  implicit none
  real(wp) :: x(2,3) = reshape((/1,1,2,2,3,3/), (/2,3/))
  real(wp) :: y(3)
  call sum_columns(x,2,3,y)
  print*, y(:)
end program main
```

```
>nvfortran mod_sum.f90 main.f90 -o exe
>./exe
  2.000000000000000       4.000000000000000       6.000000000000000
```

Part of the content of the slides about ctypes and cffi came from the prototype of a minibook about Interfacing Python with Fortran by Kjell Jorner, Rohit Goswami and Sebastian Ehlert

WRITING A FORTRAN-C INTERFACE

We start with the same function from the previous slide, but now we need to adapt it to be interoperable with C.

We needed to add a number of code elements to ensure C interoperability:

- use, intrinsic :: iso_c_binding, only: c_double, c_int

This imports the c_int (integer) and c_double (double) types from the intrinsic module iso_c_binding which provides support for C interoperability.

- integer(c_int)
- real(c_double)

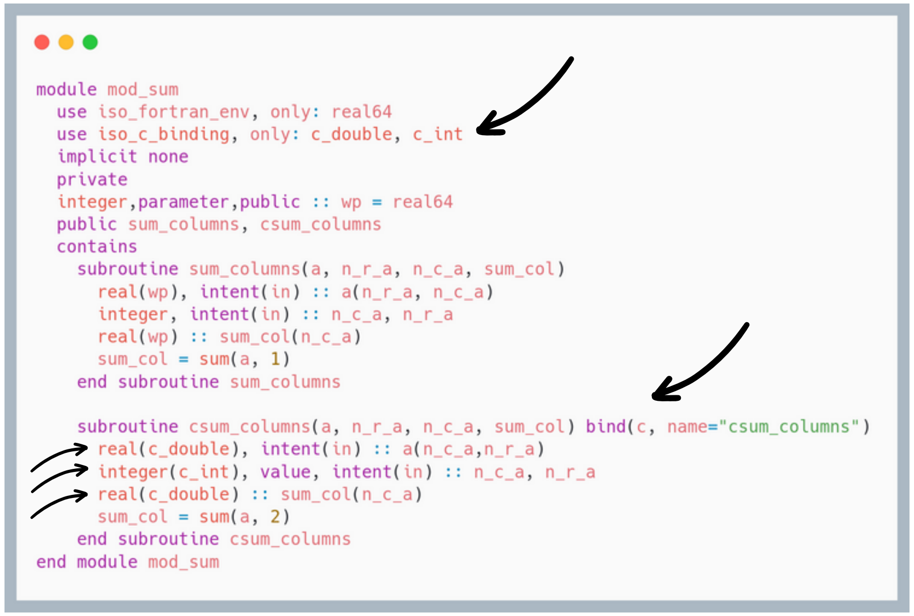
Variable types need to be C interoperable.

- bind(c)

Notifies the compiler that the function should be C interoperable.

- value

Notifies the compiler that the variable will be passed by value rather than reference.



```fortran
module mod_sum
  use iso_fortran_env, only: real64
  use iso_c_binding, only: c_double, c_int
  implicit none
  private
  integer,parameter,public :: wp = real64
  public sum_columns, csum_columns
  contains
    subroutine sum_columns(a, n_r_a, n_c_a, sum_col)
      real(wp), intent(in) :: a(n_r_a, n_c_a)
      integer, intent(in) :: n_c_a, n_r_a
      real(wp) :: sum_col(n_c_a)
      sum_col = sum(a, 1)
    end subroutine sum_columns

    subroutine csum_columns(a, n_r_a, n_c_a, sum_col) bind(c, name="csum_columns")
      real(c_double), intent(in) :: a(n_c_a,n_r_a)
      integer(c_int), value, intent(in) :: n_c_a, n_r_a
      real(c_double) :: sum_col(n_c_a)
      sum_col = sum(a, 2)
    end subroutine csum_columns
end module mod_sum
```

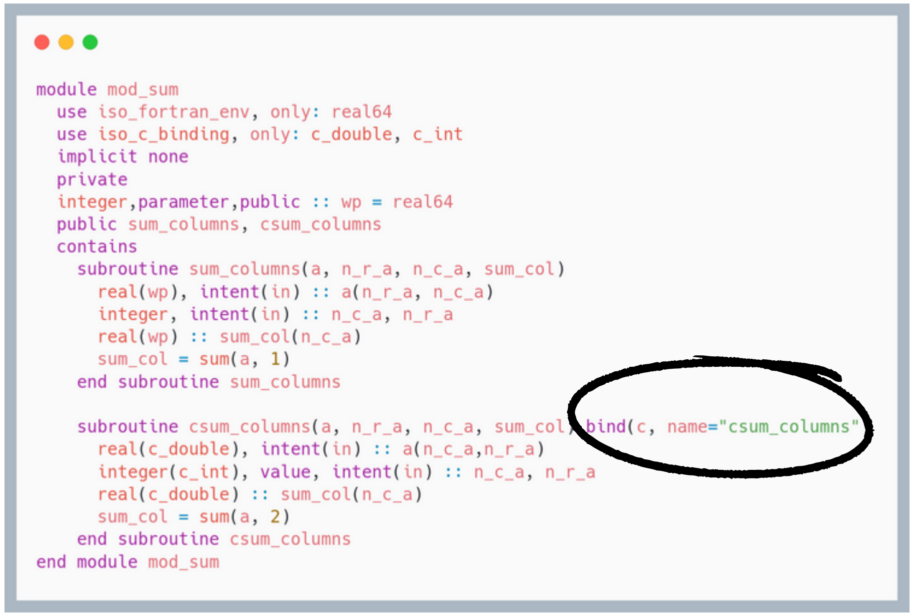
it is important to remember that C and Fortran employ different array orderings. Python normally employs the c-style row-major-order, while in Fortran the column-major-order is employed.

WRITING A FORTRAN-C INTERFACE

NAME MANGLING

In compiler construction, name mangling (also called name decoration) is a technique used to solve various problems caused by the need to resolve unique names for programming entities in many modern programming languages.
It provides a way of encoding additional information in the name of a function, structure, class or another datatype in order to pass more semantic information from the compiler to the linker.

Name mangling. (2023, Jul 17). In *Wikipedia.*
https://en.wikipedia.org/wiki/Name_mangling#C_name_decoration_in_Microsoft_Windows



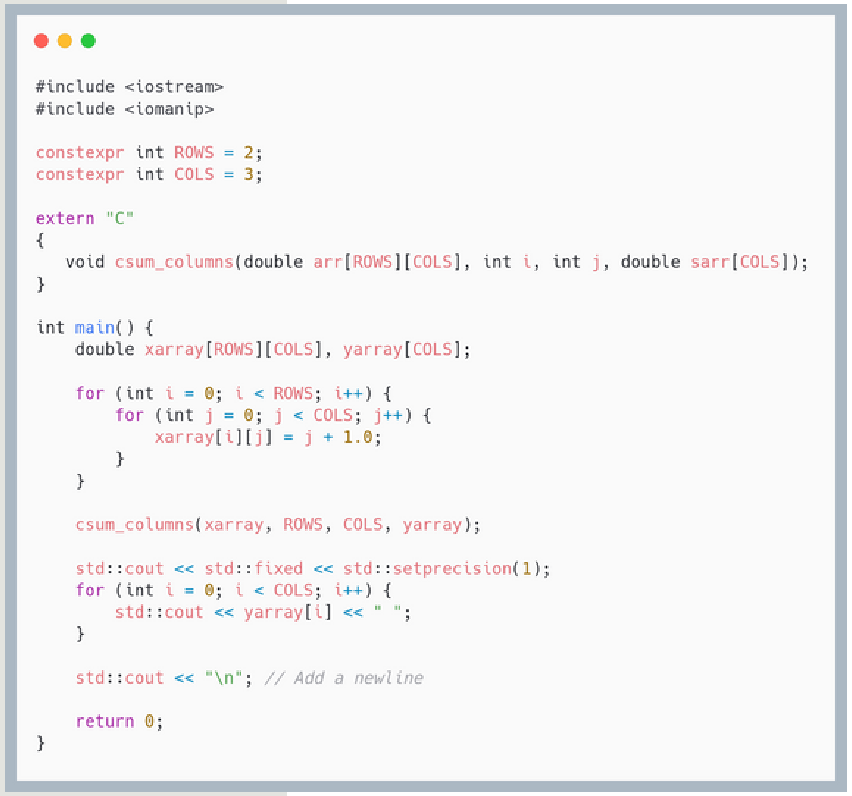
```
module mod_sum
  use iso_fortran_env, only: real64
  use iso_c_binding, only: c_double, c_int
  implicit none
  private
  integer,parameter,public :: wp = real64
  public sum_columns, csum_columns
  contains
    subroutine sum_columns(a, n_r_a, n_c_a, sum_col)
      real(wp), intent(in) :: a(n_r_a, n_c_a)
      integer, intent(in) :: n_c_a, n_r_a
      real(wp) :: sum_col(n_c_a)
      sum_col = sum(a, 1)
    end subroutine sum_columns

    subroutine csum_columns(a, n_r_a, n_c_a, sum_col) bind(c, name="csum_columns")
      real(c_double), intent(in) :: a(n_c_a,n_r_a)
      integer(c_int), value, intent(in) :: n_c_a, n_r_a
      real(c_double) :: sum_col(n_c_a)
      sum_col = sum(a, 2)
    end subroutine csum_columns
end module mod_sum
```

By default, Fortran compilers generate mangled names (for example, converting function names to lowercase or uppercase, often appending an underscore), and so to call a Fortran function from C you must pass the mangled identifier corresponding to the rule followed by your Fortran compiler.

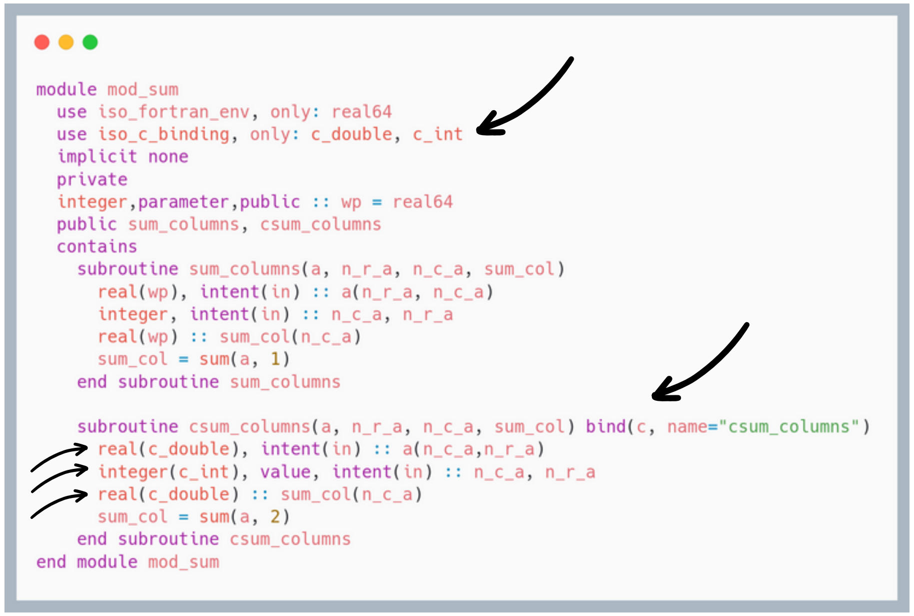
The BIND(C) attribute removes name mangling and uses C-style symbol name, that is, one-to-one mapping. If you need further control, add NAME=, in which case the string you give is treated exactly as if you had specified that in the "companion C processor".

"Subroutine FortXYZ (args) bind (C, name="realname")" will appear to the linker as "realname" without any mangling.

WRITING A FORTRAN-C INTERFACE



```cpp
#include <iostream>
#include <iomanip>

constexpr int ROWS = 2;
constexpr int COLS = 3;

extern "C"
{
   void csum_columns(double arr[ROWS][COLS], int i, int j, double sarr[COLS]);
}

int main() {
    double xarray[ROWS][COLS], yarray[COLS];

    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            xarray[i][j] = j + 1.0;
        }
    }

    csum_columns(xarray, ROWS, COLS, yarray);

    std::cout << std::fixed << std::setprecision(1);
    for (int i = 0; i < COLS; i++) {
        std::cout << yarray[i] << " ";
    }

    std::cout << "\n"; // Add a newline

    return 0;
}
```

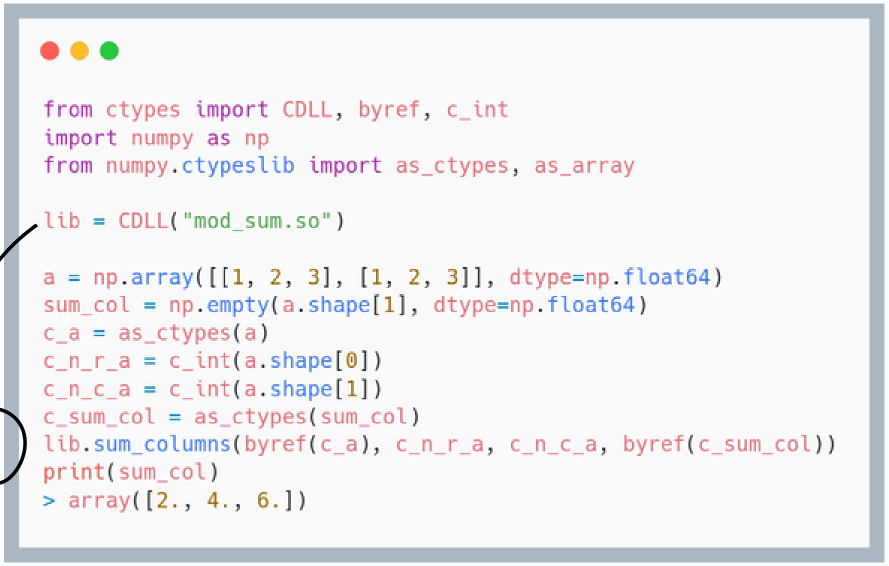


```fortran
module mod_sum
  use iso_fortran_env, only: real64
  use iso_c_binding, only: c_double, c_int
  implicit none
  private
  integer,parameter,public :: wp = real64
  public sum_columns, csum_columns
  contains
    subroutine sum_columns(a, n_r_a, n_c_a, sum_col)
      real(wp), intent(in) :: a(n_r_a, n_c_a)
      integer, intent(in) :: n_c_a, n_r_a
      real(wp) :: sum_col(n_c_a)
      sum_col = sum(a, 1)
    end subroutine sum_columns

    subroutine csum_columns(a, n_r_a, n_c_a, sum_col) bind(c, name="csum_columns")
      real(c_double), intent(in) :: a(n_c_a,n_r_a)
      integer(c_int), value, intent(in) :: n_c_a, n_r_a
      real(c_double) :: sum_col(n_c_a)
      sum_col = sum(a, 2)
    end subroutine csum_columns
end module mod_sum
```

```
>nvfortran -c mod_sum_c.f90
>nvc++ -Wall mod_sum_c.o sum_columns_sta.cc -fortranlibs
>./a.out
 2.0 4.0 6.0

>gfortran -c -o mod_sum_c.f90
>gcc -o exe mod_sum_c.o sum_columns_dyn.c
>./exe
 2.0 4.0 6.0

>gfortran -c -o mod_sum_c.f90
>g++ -o exe mod_sum_c.o sum_columns_dyn.cc
>./exe
 2.0 4.0 6.0
```

Ctypes

The ctypes library is part of the Python standard libary and provides convenient access to C compatible data types and loading of shared libraries. C datatypes like c_int, c_double, c_long etc. are supported and libraries are loaded with CDLL.



```python
from ctypes import CDLL, byref, c_int
import numpy as np
from numpy.ctypeslib import as_ctypes, as_array

lib = CDLL("mod_sum.so")

a = np.array([[1, 2, 3], [1, 2, 3]], dtype=np.float64)
sum_col = np.empty(a.shape[1], dtype=np.float64)
c_a = as_ctypes(a)
c_n_r_a = c_int(a.shape[0])
c_n_c_a = c_int(a.shape[1])
c_sum_col = as_ctypes(sum_col)
lib.sum_columns(byref(c_a), c_n_r_a, c_n_c_a, byref(c_sum_col))
print(sum_col)
> array([2., 4., 6.])
```

Compiling to a shared library

We now need to compile the function into a shared library

- Linux + nvfortran 23.5-0

nvfortran -O3 -march=native -fpic -c mod_sum.f90
nvfortran -shared -o mod_sum.so mod_sum.o

- Linux + ifort

ifort -O3 -march=native -fpic -c mod_sum.f90
ifort shared -o mod_sum.so mod_sum.o

- Linux + ifx

ifx -O3 -march=native -fpic -c mod_sum.f90
ifx shared -o mod_sum.so mod_sum.o

Importing the shared library from Python

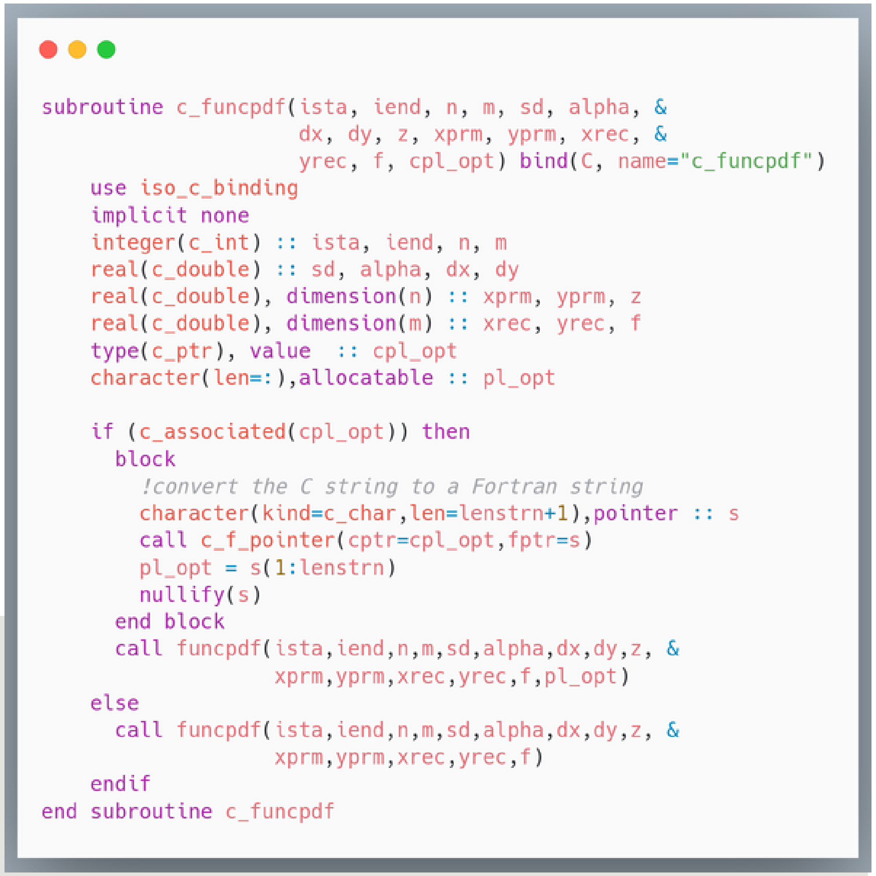
We can now import and use the function in Python with the built-in ctypes library. We must match our variable types to conform to those of our Fortran functions.

Working with NumPy arrays

The most convenient way to work with arrays is through numpy.ctypeslib. In particular, it includes the functions as_array, which converts a C array to a NumPy ndarray, and as_ctypes, which converts an ndarray to a C array. Note the explicit declaration of the datatypes for the Numpy arrays which avoids NumPy guessing the type from the input values. In accordance with C standard, arrays need to be passed by reference rather than by value. That is handled by the byref function of ctypes.

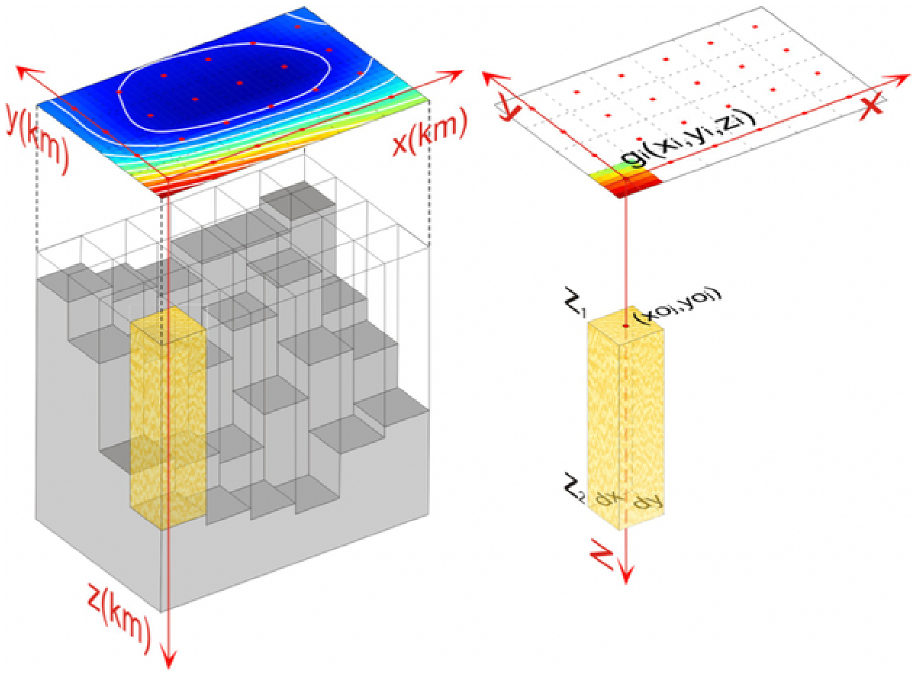
A Geophysical plain example

Gravity field caused by a sediment-basement interface



```fortran
subroutine c_funcpdf(ista, iend, n, m, sd, alpha, &
                     dx, dy, z, xprm, yprm, xrec, &
                     yrec, f, cpl_opt) bind(C, name="c_funcpdf")
    use iso_c_binding
    implicit none
    integer(c_int) :: ista, iend, n, m
    real(c_double) :: sd, alpha, dx, dy
    real(c_double), dimension(n) :: xprm, yprm, z
    real(c_double), dimension(m) :: xrec, yrec, f
    type(c_ptr), value  :: cpl_opt
    character(len=:),allocatable :: pl_opt

    if (c_associated(cpl_opt)) then
      block
        !convert the C string to a Fortran string
        character(kind=c_char,len=lenstrn+1),pointer :: s
        call c_f_pointer(cptr=cpl_opt,fptr=s)
        pl_opt = s(1:lenstrn)
        nullify(s)
      end block
      call funcpdf(ista,iend,n,m,sd,alpha,dx,dy,z, &
                   xprm,yprm,xrec,yrec,f,pl_opt)
    else
      call funcpdf(ista,iend,n,m,sd,alpha,dx,dy,z, &
                   xprm,yprm,xrec,yrec,f)
    endif
end subroutine c_funcpdf
```

Schematic representation of a gravity anomaly produced by the basement interface. The sedimentary pack is discretized into a grid of 3D vertical prisms with thicknesses (z2 − z1). The diagram on the right shows the contribution to the anomalous gravitational field gi(xi,yi,zi) at the ith observation point produced by the jth prism.

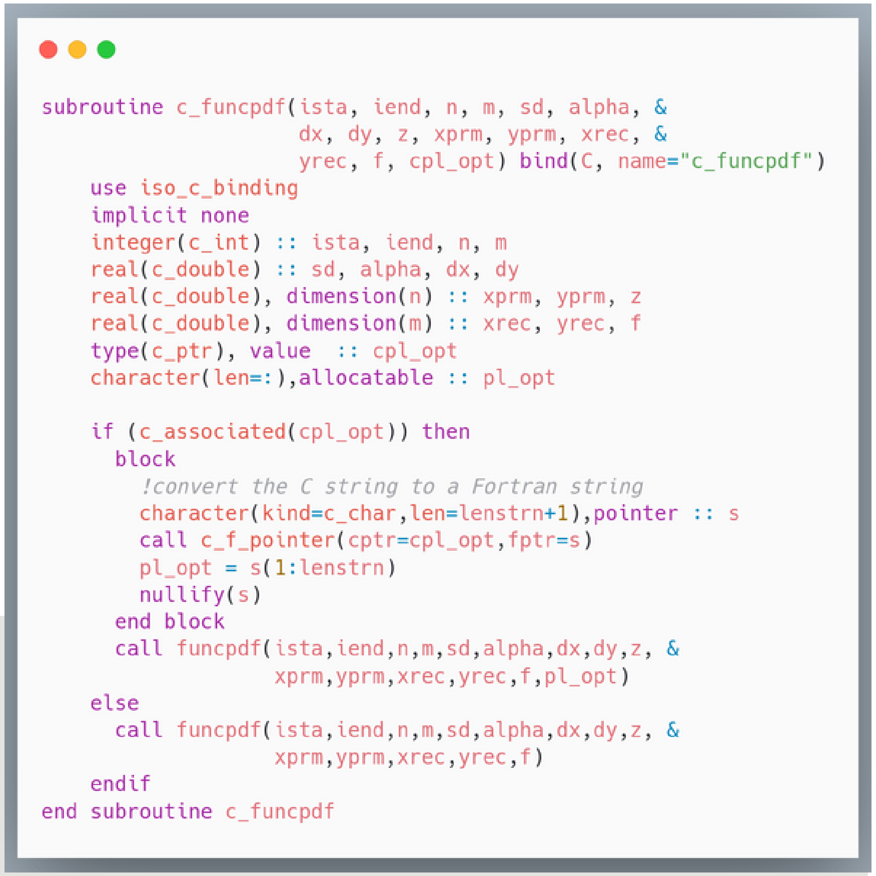


N–BODY PROBLEM

$$g_i(x_i, y_i, z_i) = \sum_{j=1}^{N} f_i(z_j, \Delta\rho_0, \alpha), \qquad i = 1, \ldots, M$$

A Geophysical plain example

Gravity field caused by a sediment-basement interface



```fortran
subroutine c_funcpdf(ista, iend, n, m, sd, alpha, &
                     dx, dy, z, xprm, yprm, xrec, &
                     yrec, f, cpl_opt) bind(C, name="c_funcpdf")
    use iso_c_binding
    implicit none
    integer(c_int) :: ista, iend, n, m
    real(c_double) :: sd, alpha, dx, dy
    real(c_double), dimension(n) :: xprm, yprm, z
    real(c_double), dimension(m) :: xrec, yrec, f
    type(c_ptr), value  :: cpl_opt
    character(len=:),allocatable :: pl_opt

    if (c_associated(cpl_opt)) then
      block
        !convert the C string to a Fortran string
        character(kind=c_char,len=lenstrn+1),pointer :: s
        call c_f_pointer(cptr=cpl_opt,fptr=s)
        pl_opt = s(1:lenstrn)
        nullify(s)
      end block
      call funcpdf(ista,iend,n,m,sd,alpha,dx,dy,z, &
                   xprm,yprm,xrec,yrec,f,pl_opt)
    else
      call funcpdf(ista,iend,n,m,sd,alpha,dx,dy,z, &
                   xprm,yprm,xrec,yrec,f)
    endif
end subroutine c_funcpdf
```

DEALING WITH C STRINGS

C strings are actually one-dimensional array of characters terminated by a nullcharacter '\0'

- Use "type(c_ptr)" for mapping "char *"
- Use "c_f_pointer" to associate the "char *" C pointer to a Fortran pointer of "character(kind=c_char,len=lenstrn+1), pointer :: s", within a block construct. Note that I know the string length 'lenstrn' in advance and I could have used an static character string like "character(len=lenstrn) :: cpl_opt". In cases where the string length is unknown, using a deferred-length (allocatable) string is necessary. Additionally, a function should be available to calculate the string length 'lenstrn'.
- Do the normal Fortran character type of operations on the fortran string 's' e.g., assignment of pl_opt = s (Allocation on assignment).
- Once done with the operations, nullify 's', typically before the block construct is exited.

Based on Degenerate conic Blog by Jacob Williams
https://degenerateconic.com/fortran-c-interoperability-part-2.html

A Geophysical plain example

Gravity field caused by a sediment-basement interface

Using Function Signatures to Call Functions

Although not strictly required in this case, we use ctypes's argtypes and restype attributes. They do 2 things:

1. argtypes guards the function by checking the arguments before calling the library code.
2. Using these attributes tells Python how to convert the input Python values to ctypes values, and how to convert the output back to a Python value.

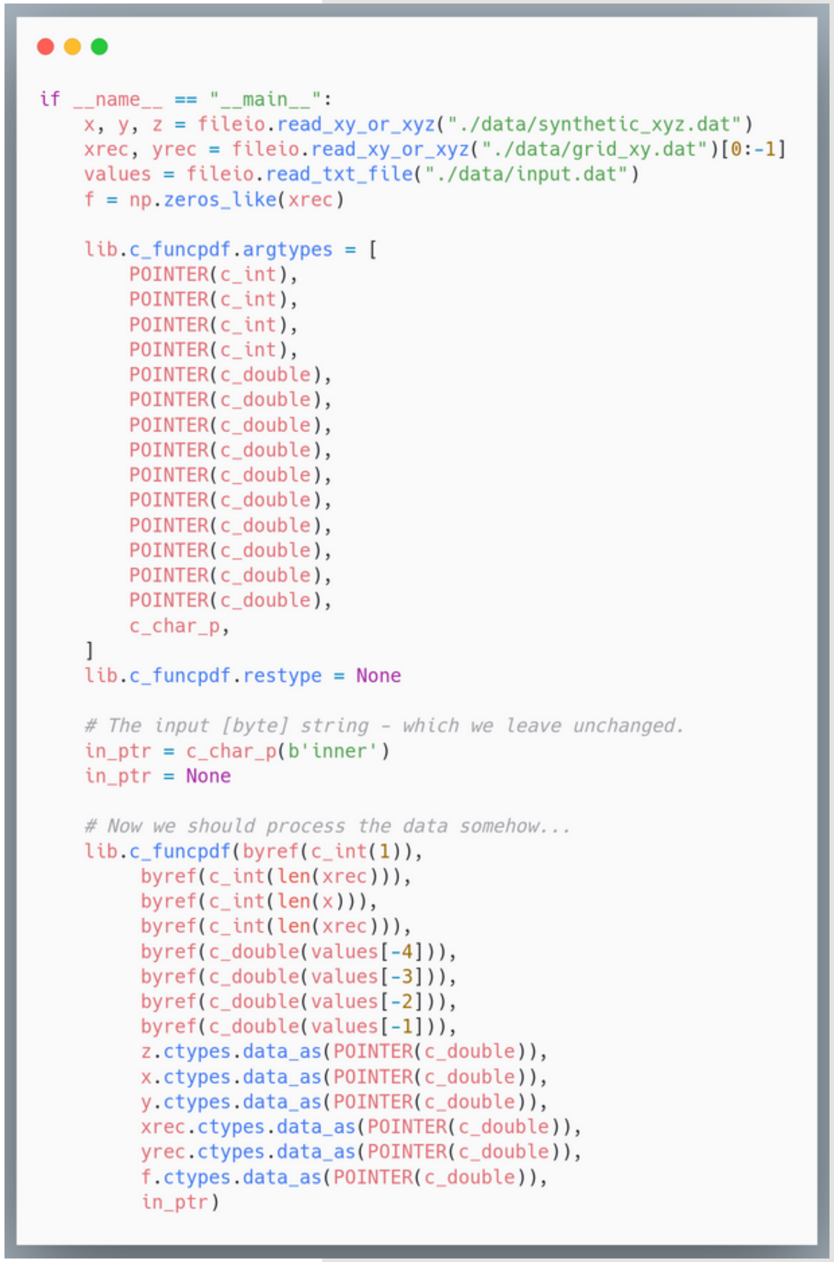
If you try to pass the wrong types to the function you will see a TypeError exception.

Strings in ctypes

- c_char_p: is a data type from the ctypes, which represents a C-style character pointer.
- b'inner': creates a bytes object (immutable sequence of elements that are bytes) initialized to the bytes determined by the ASCII* codes of each character in the string.

in_ptr = c_char_p(b'inner'): creates a c_char_p object named in_ptr and assigns it the memory address of the Python byte string b'inner'. The c_char_p object acts as a C-style pointer to the byte string.

The reason is that C treats all strings as byte sequences

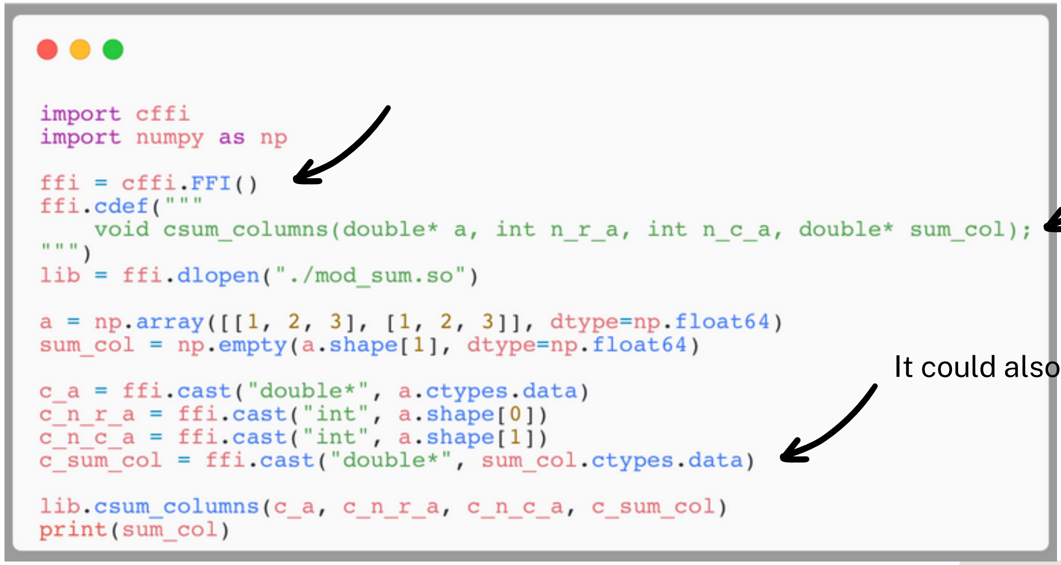


```python
if __name__ == "__main__":
    x, y, z = fileio.read_xy_or_xyz("./data/synthetic_xyz.dat")
    xrec, yrec = fileio.read_xy_or_xyz("./data/grid_xy.dat")[0:-1]
    values = fileio.read_txt_file("./data/input.dat")
    f = np.zeros_like(xrec)

    lib.c_funcpdf.argtypes = [
        POINTER(c_int),
        POINTER(c_int),
        POINTER(c_int),
        POINTER(c_int),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        POINTER(c_double),
        c_char_p,
    ]
    lib.c_funcpdf.restype = None

    # The input [byte] string - which we leave unchanged.
    in_ptr = c_char_p(b'inner')
    in_ptr = None

    # Now we should process the data somehow...
    lib.c_funcpdf(byref(c_int(1)),
         byref(c_int(len(xrec))),
         byref(c_int(len(x))),
         byref(c_int(len(xrec))),
         byref(c_double(values[-4])),
         byref(c_double(values[-3])),
         byref(c_double(values[-2])),
         byref(c_double(values[-1])),
         z.ctypes.data_as(POINTER(c_double)),
         x.ctypes.data_as(POINTER(c_double)),
         y.ctypes.data_as(POINTER(c_double)),
         xrec.ctypes.data_as(POINTER(c_double)),
         yrec.ctypes.data_as(POINTER(c_double)),
         f.ctypes.data_as(POINTER(c_double)),
         in_ptr)
```

*ASCII Printable Characters (128 characters, see https://www.w3schools.com/charsets/ref_html_ascii.asp)

Cffi

cffi is a C Foreign Function Interface for Python. It is more flexible than ctypes when writing more complex interfaces. Here is an example using the mod_sum.so shared library built before:



```
import cffi
import numpy as np

ffi = cffi.FFI()
ffi.cdef("""
    void csum_columns(double* a, int n_r_a, int n_c_a, double* sum_col);
""")
lib = ffi.dlopen("./mod_sum.so")

a = np.array([[1, 2, 3], [1, 2, 3]], dtype=np.float64)
sum_col = np.empty(a.shape[1], dtype=np.float64)

c_a = ffi.cast("double*", a.ctypes.data)
c_n_r_a = ffi.cast("int", a.shape[0])
c_n_c_a = ffi.cast("int", a.shape[1])
c_sum_col = ffi.cast("double*", sum_col.ctypes.data)

lib.csum_columns(c_a, c_n_r_a, c_n_c_a, c_sum_col)
print(sum_col)
```

For each function/subroutine you should provide a C-signature to CFFI

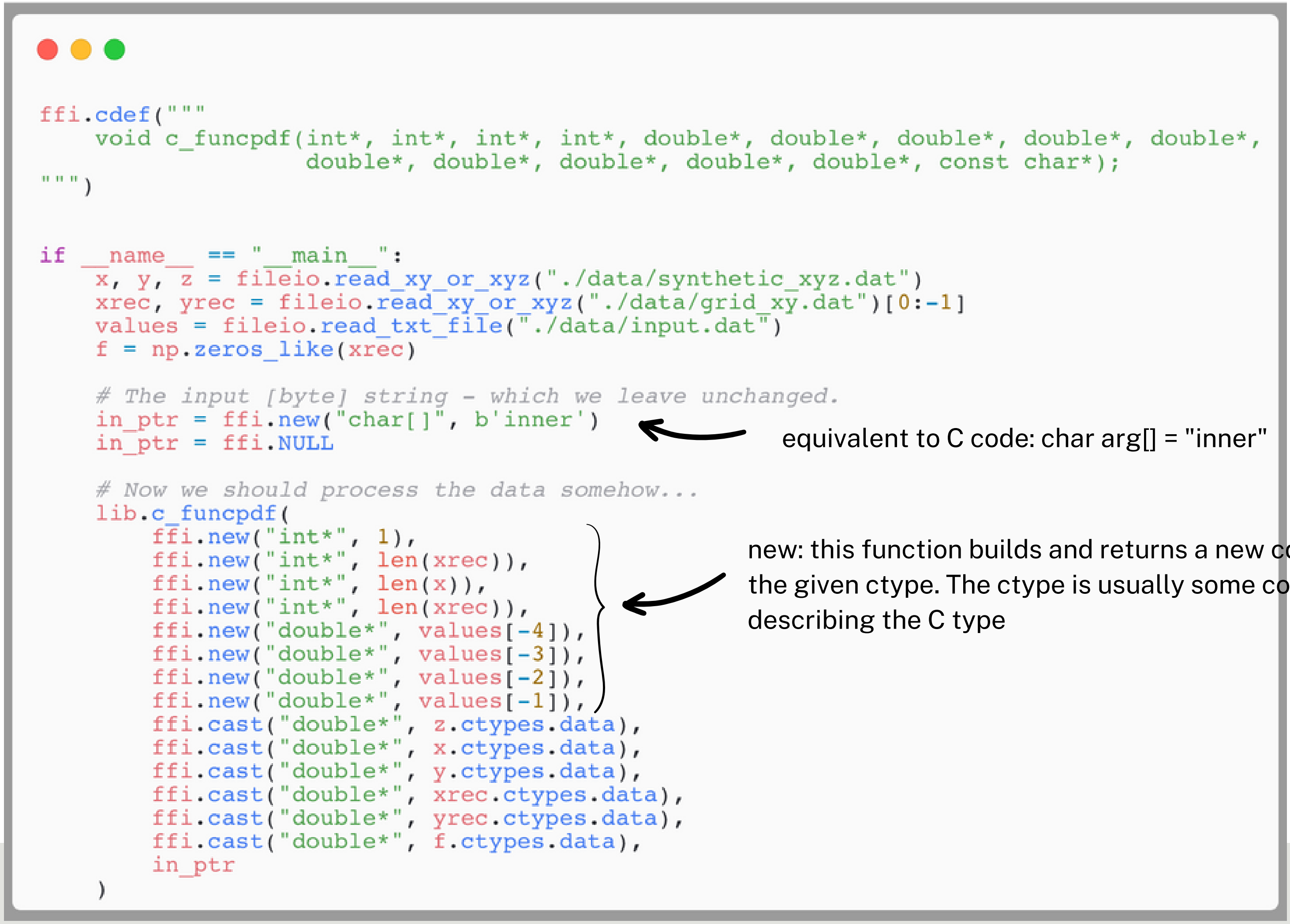
It could also have seen sum_col.__array_interface__['data'][0]

First, a FFI object is created to handle all the interactions with the library. The library is opened with ffi.dlopen. One key difference between ctypes and cffi concerns the need to include C declarations using cdef. This method takes a C code string as it would be given in a C header file.

Another difference is that FFI.cast is used to cast Python types as C types.The first argument to cast is a C code string describing the type. Pointers are described by using the <type> * syntax, and there is therefore no need for a separate byref function as in ctypes.

A Geophysical plain example

Gravity field caused by a sediment-basement interface



```python
ffi.cdef("""
    void c_funcpdf(int*, int*, int*, int*, double*, double*, double*, double*, double*,
                   double*, double*, double*, double*, double*, const char*);
""")


if __name__ == "__main__":
    x, y, z = fileio.read_xy_or_xyz("./data/synthetic_xyz.dat")
    xrec, yrec = fileio.read_xy_or_xyz("./data/grid_xy.dat")[0:-1]
    values = fileio.read_txt_file("./data/input.dat")
    f = np.zeros_like(xrec)

    # The input [byte] string - which we leave unchanged.
    in_ptr = ffi.new("char[]", b'inner')
    in_ptr = ffi.NULL

    # Now we should process the data somehow...
    lib.c_funcpdf(
        ffi.new("int*", 1),
        ffi.new("int*", len(xrec)),
        ffi.new("int*", len(x)),
        ffi.new("int*", len(xrec)),
        ffi.new("double*", values[-4]),
        ffi.new("double*", values[-3]),
        ffi.new("double*", values[-2]),
        ffi.new("double*", values[-1]),
        ffi.cast("double*", z.ctypes.data),
        ffi.cast("double*", x.ctypes.data),
        ffi.cast("double*", y.ctypes.data),
        ffi.cast("double*", xrec.ctypes.data),
        ffi.cast("double*", yrec.ctypes.data),
        ffi.cast("double*", f.ctypes.data),
        in_ptr
    )
```

equivalent to C code: char arg[] = "inner"

new: this function builds and returns a new cdata object of the given ctype. The ctype is usually some constant string describing the C type

Cython -- C → Python and Python → C

Cython is an optimising static compiler for Python that also provides its own programming language as a superset for standard Python. Cython is designed to provide C-like performance for a code that is mostly written in Python by adding only a few C-like declarations to an existing Python code.

Cython comes as an installable library using the “pip” package manager. Install it using this command:

> pip install Cython

Using Cython consists of these steps:

1. Write a .pyx source file
2. Run the Cython compiler to generate a C file
3. Run a C compiler to generate a compiled library
4. Run the Python interpreter and ask it to import the module

cython command

- The cython command takes a .py or .pyx file and compiles it into a C/C++ file (i.e., cython yourmod.pyx -3).
  - How to compile your .c files will vary depending on your operating system and compiler
  - On a Linux system, for example, it might look similar to this:
    - $ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing -I/usr/include/python3.5 -o yourmod.so yourmod.c

cythonize command

- The cythonize command takes a .py or .pyx file and compiles it into a C/C++ file. It then compiles the C/C++ file into an extension module which is directly importable from Python.
  - cythonize -I yourmod.pyx (-i builds “in place”)

setuptools

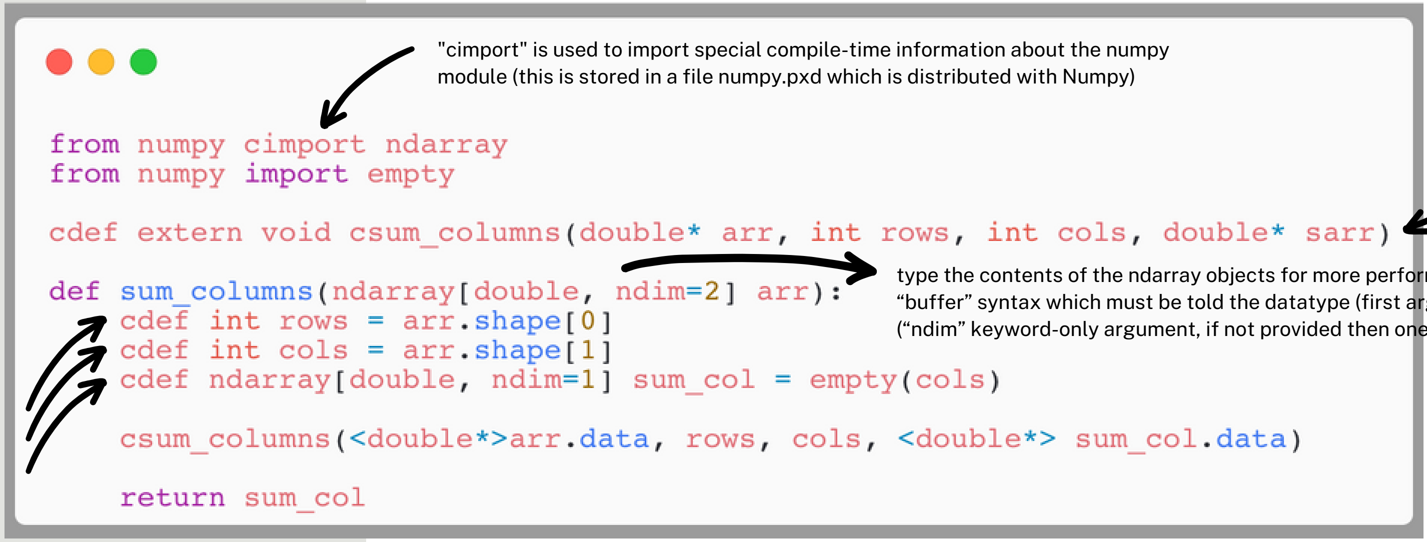
- Cython supports setuptools so that you can very easily create build scripts which automate the process, this is the preferred method for Cython implemented libraries and packages
  - Add a setup.py file in the same directory as the Cython file:

Cython

In brief, the workflow described here is to:

1. Create a fortran wrapper function that defines c_iso_bindings for a fortran subroutine
2. Create a cython .pyx function
3. After that, run the setup.py file using the following command: python setup.py build_ext --inplace
4. To verify that it works, run the test script.

CYTHON HAS SUPPORT FOR FAST ACCESS TO NUMPY ARRAYS.



"cimport" is used to import special compile-time information about the numpy module (this is stored in a file numpy.pxd which is distributed with Numpy)

```
from numpy cimport ndarray
from numpy import empty

cdef extern void csum_columns(double* arr, int rows, int cols, double* sarr)

def sum_columns(ndarray[double, ndim=2] arr):
    cdef int rows = arr.shape[0]
    cdef int cols = arr.shape[1]
    cdef ndarray[double, ndim=1] sum_col = empty(cols)

    csum_columns(<double*>arr.data, rows, cols, <double*> sum_col.data)

    return sum_col
```

type the contents of the ndarray objects for more performance. We do this with a special “buffer” syntax which must be told the datatype (first argument) and number of dimensions (“ndim” keyword-only argument, if not provided then one-dimensional is assumed).

- The "cdef" keyword defines functions that are implemented primarily in C for additional speed, although those functions can only be called by other Cython functions and not by Python scripts. The "cdef" keyword is also used within functions to type variables.
- Explicitly declare the variable types, both for the function parameters and the variables used in the body of the function (double, int, and so on), Cython will translate all of this into C

Cython

Setuptools is a package development process library designed to facilitate packaging Python projects by enhancing the Python standard library distutils (distribution utilities). Setuptools. (2023, March 27). In *Wikipedia.https://en.wikipedia.org/wiki/Setuptools*

All the features of what can go into a setup.py file is beyond the scope of this simple tutorial. The basic usage of setup.py is:

> python setup.py <some_command> <options>

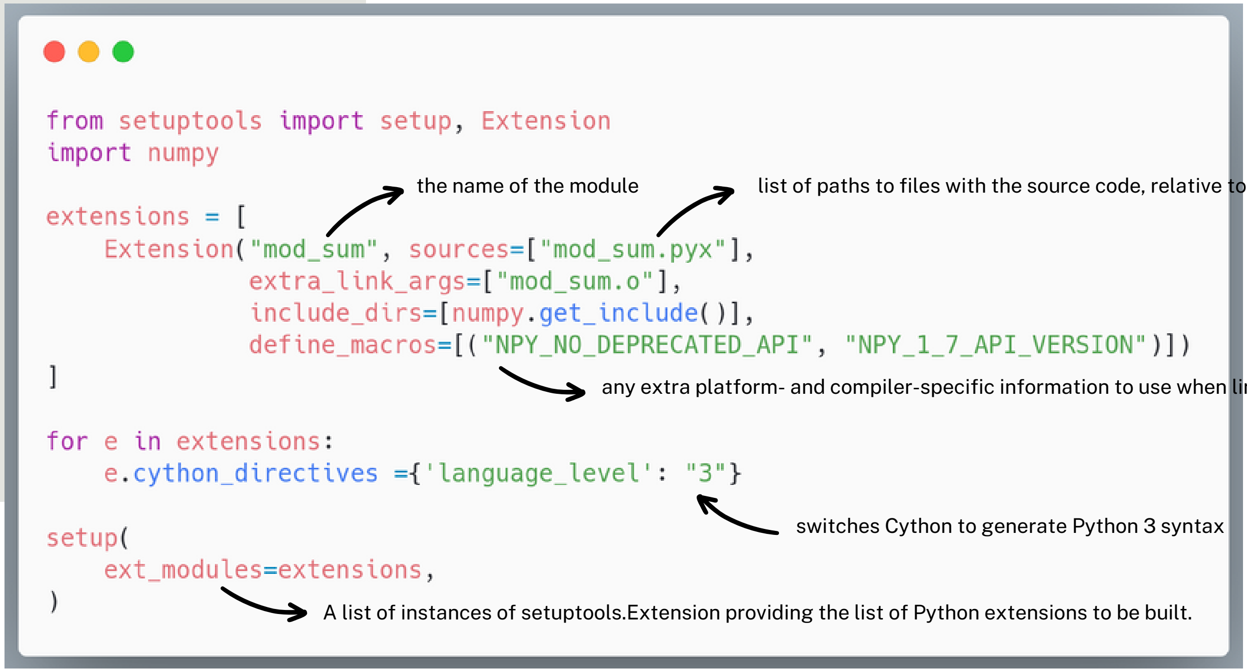
To see all commands type:

> python setup.py --help-commands

What is an Extension Module?
A module which can be imported and used from within Python which is written in another language

Extension API Reference: https://setuptools.pypa.io/en/latest/userguide/ext_modules.html#setuptools.Extension

Setup keywords: https://setuptools.pypa.io/en/latest/references/keywords.html



```python
from setuptools import setup, Extension
import numpy

extensions = [
    Extension("mod_sum", sources=["mod_sum.pyx"],
              extra_link_args=["mod_sum.o"],
              include_dirs=[numpy.get_include()],
              define_macros=[("NPY_NO_DEPRECATED_API", "NPY_1_7_API_VERSION")])
]

for e in extensions:
    e.cython_directives ={'language_level': "3"}

setup(
    ext_modules=extensions,
)
```

the name of the module

list of paths to files with the source code, relative to the setup script

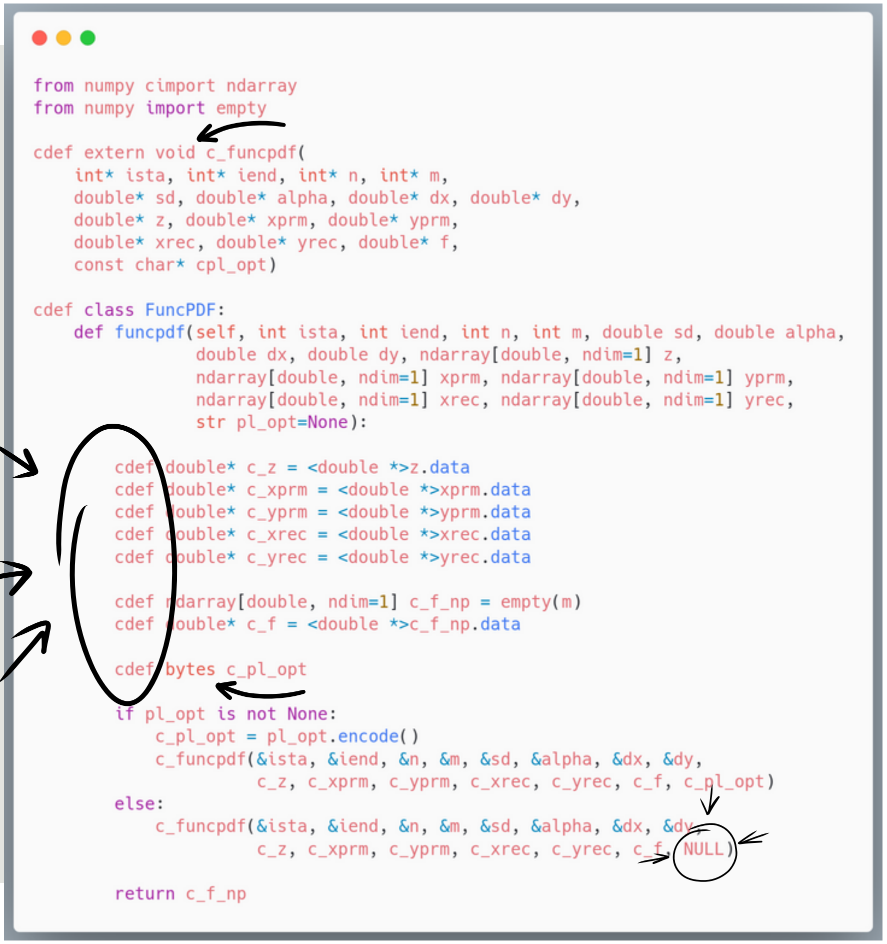
any extra platform- and compiler-specific information to use when linking object files together to create the extension

switches Cython to generate Python 3 syntax

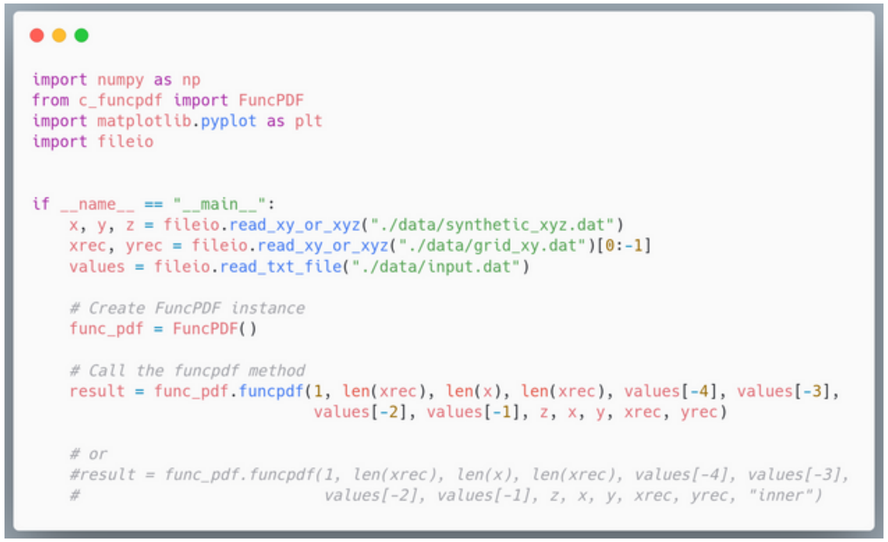
A list of instances of setuptools.Extension providing the list of Python extensions to be built.

A Geophysical plain example

Gravity field caused by a sediment-basement interface



```
from numpy cimport ndarray
from numpy import empty

cdef extern void c_funcpdf(
    int* ista, int* iend, int* n, int* m,
    double* sd, double* alpha, double* dx, double* dy,
    double* z, double* xprm, double* yprm,
    double* xrec, double* yrec, double* f,
    const char* cpl_opt)

cdef class FuncPDF:
    def funcpdf(self, int ista, int iend, int n, int m, double sd, double alpha,
                double dx, double dy, ndarray[double, ndim=1] z,
                ndarray[double, ndim=1] xprm, ndarray[double, ndim=1] yprm,
                ndarray[double, ndim=1] xrec, ndarray[double, ndim=1] yrec,
                str pl_opt=None):

        cdef double* c_z = <double *>z.data
        cdef double* c_xprm = <double *>xprm.data
        cdef double* c_yprm = <double *>yprm.data
        cdef double* c_xrec = <double *>xrec.data
        cdef double* c_yrec = <double *>yrec.data

        cdef ndarray[double, ndim=1] c_f_np = empty(m)
        cdef double* c_f = <double *>c_f_np.data

        cdef bytes c_pl_opt

        if pl_opt is not None:
            c_pl_opt = pl_opt.encode()
            c_funcpdf(&ista, &iend, &n, &m, &sd, &alpha, &dx, &dy,
                      c_z, c_xprm, c_yprm, c_xrec, c_yrec, c_f, c_pl_opt)
        else:
            c_funcpdf(&ista, &iend, &n, &m, &sd, &alpha, &dx, &dy,
                      c_z, c_xprm, c_yprm, c_xrec, c_yrec, c_f, NULL)

        return c_f_np
```



```
import numpy as np
from c_funcpdf import FuncPDF
import matplotlib.pyplot as plt
import fileio


if __name__ == "__main__":
    x, y, z = fileio.read_xy_or_xyz("./data/synthetic_xyz.dat")
    xrec, yrec = fileio.read_xy_or_xyz("./data/grid_xy.dat")[0:-1]
    values = fileio.read_txt_file("./data/input.dat")

    # Create FuncPDF instance
    func_pdf = FuncPDF()

    # Call the funcpdf method
    result = func_pdf.funcpdf(1, len(xrec), len(x), len(xrec), values[-4], values[-3],
                              values[-2], values[-1], z, x, y, xrec, yrec)

    # or
    #result = func_pdf.funcpdf(1, len(xrec), len(x), len(xrec), values[-4], values[-3],
    #                           values[-2], values[-1], z, x, y, xrec, yrec, "inner")
```

Unicode and passing strings: https://cython.readthedocs.io/en/latest/src/tutorial/strings.html

Useful links

Python Fortran Rosetta Stone: https://fortran-lang.org/learn/rosetta_stone/#

Python Bindings: Calling C or C++ From Python: https://realpython.com/python-bindings-overview/#how-its-installed_3

Building a Python C Extension Module: https://realpython.com/build-python-c-extension-module/