ICHEC
Irish Centre for High-end Computing

SUMMER OF HPC

PRACE

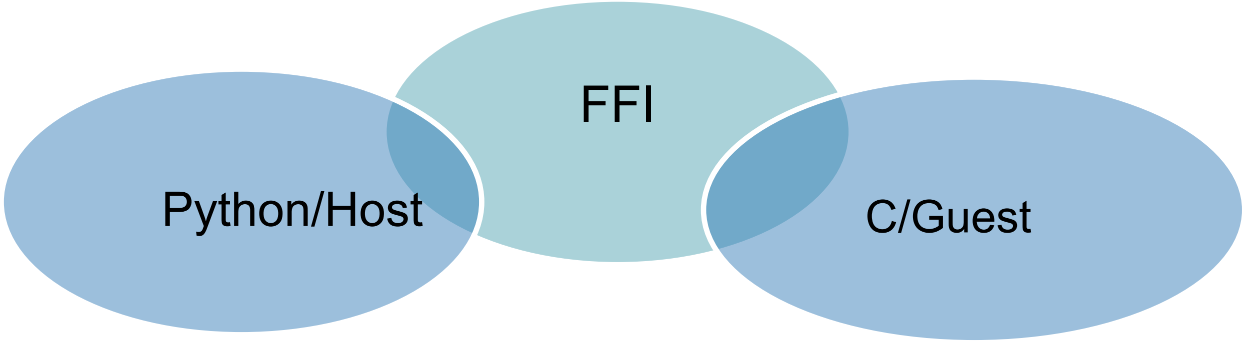
C Foreign Function Interface for Python - `cffi`

Instructor: Christopher Werner

Overview

- Some useful libraries are written in lower level languages like C, and can’t be directly used by higher level languages
- A few options:
  - Port all or part of the library to your language of choice
  - Write an extension in C to bridge the gap between the library and your language
  - Wrap the library using your language's **foreign function interface** (FFI) support

FFI wrappers are easier to write and maintain than C-extension



When to avoid FFI

- Implementation of your own low-level/optimised code, especially if you need to write custom code to directly process huge arrays, in order to get the performance you want
- Delicate call-backs from the guest language into the host. Some complex call-backs can be difficult.
- Library makes use of compile-time or pre-processor features (eg. C macros)

cffi

- External package that provides C Foreign Function Interface for Python
- Requires user to add C-like declarations (not static) to the Python code
- Main modes:
  - **ABI** (**A**pplication **B**inary **I**nterface) – “Easier” but slower
  - **API** (**A**pplication **P**rogrammer **I**nterface) – Harder but faster
- Goal is to call existing C libraries from Python **NOT** embedding C executable code
- Sub modes:
  - **In-line**: everything set up when code imported
  - **Out-of-line**: separate step of preparation

`cffi` - main utilities

- `ffi.cast("C-type", value)` - The value is casted between integers or pointers of any type
- `ffi.dlopen(libpath, [flags])` - opens and returns a handle to a dynamic library
- `ffi.cdef("C-type", value)` - creates a new ffi object with the declaration of function
- `ffi.from_buffer([cdecl,] python_buffer, …)` - return an array cdata that points to the data of the given Python object,
- `ffi.new("int *")` - allocate an instance according to the specified C type and return a pointer to it
- `ffi` vs `ffibuilder`
- Documentation for library can be found here.

Importing an existing library

- Library files have the extension `.so` for Windows, UNIX, `.dylib` for Macs

```
from cffi import FFI
ffi = FFI()

lib = ffi.dlopen("libm.so.6")
ffi.cdef("""float sqrtf(float x);""")
a = lib.sqrtf(4)
print(a)
```

```
2.0
```

Create and use your own library (ABI in-line)

- **Step 1**: Create `ABI_add.c` file

```
#include <stdio.h>

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

Create and use your own library (ABI in-line)

- **Step 2**: Create `Makefile`

```
CC=gcc
CFLAGS=-fPIC -O3
LDFLAGS=-shared

mylib_add.so: ABI_add.c
	$(CC) -o ABI_add.so $(LDFLAGS) $(CFLAGS)
ABI_add.c
```

- **Step 3**: Create your library by using `make` command

```
ABI_add.c
ABI_add.so
Makefile
```

Create and use your own library (ABI in-line)

- **Step 4**: Import library and cast variables

```
from cffi import FFI
import numpy as np
import time

ffi = FFI()
lib = ffi.dlopen('./ABI_add.so')
ffi.cdef("void add(double *, double *, int);")


t0=time.time()
a = np.arange(0,200000,0.1)
b = np.ones_like(a)

aptr = ffi.cast("double *", ffi.from_buffer(a))
bptr = ffi.cast("double *", ffi.from_buffer(b))

lib.add(aptr, bptr, len(a))
print("a + b = ", a)
t1=time.time()

print ("\ntime taken for ABI in line", t1-t0)
```

Create and use your own library (API out of line)

- **Step 1**: Create a python build file, `API_add_build.py`

```
import cffi
ffibuilder = cffi.FFI()
ffibuilder.cdef("""void API_add(double *, double *, int);""")
ffibuilder.set_source("out_of_line._API_add", r"""
  void API_add(double *a, double *b, int n)
  {
       int i;
       for (i=0; i<n; i++){
           a[i] += b[i];
           }
           /* or some algorithm that is seriously faster in C than in Python
*/
  }
  """)


if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

- **Step 2**: Run the build file

Create and use your own library (API out of line)

- **Step 3**: Import the library and cast the variables

```
import numpy as np
import time
from out_of_line._API_add import ffi, lib

t0=time.time()
a = np.arange(0,200000,0.1)
b = np.ones_like(a)

# "pointer" objects
aptr = ffi.cast("double *", ffi.from_buffer(a))
bptr = ffi.cast("double *", ffi.from_buffer(b))

lib.API_add(aptr, bptr, len(a))
print("a + b = ", a)
t1=time.time()
print("\ntime taken for API out of line", t1-t0)
```

Create and use your own library (API out of line)

- **Step 3**: Import the library and cast the variables

```
import numpy as np
import time
from out_of_line._API_add import ffi, lib

t0=time.time()
a = np.arange(0,200000,0.1)
b = np.ones_like(a)

# "pointer" objects
aptr = ffi.cast("double *", ffi.from_buffer(a))
bptr = ffi.cast("double *", ffi.from_buffer(b))

lib.API_add(aptr, bptr, len(a))
print("a + b = ", a)
t1=time.time()
print("\ntime taken for API out of line", t1-t0)
```

Summary of different methods

- **ABI - in line**
  - Create/Have a `.c` file containing your C code
  - Have a MAKEFILE with compiler flags to make the output and library files
  - Create a `.py` file which imports, implements the library and casts any necessary variables
- **API - out of line**
  - Create a build file in python. The set_source can be a block of C code or a `.h` header file
  - Run the build file
  - Import library

Hands-on

- In the repo, navigate to the folder `05-CFFI`

1. Use the slides to run the same code in the `demo` folder
2. Head to the `exercise` folder and implement the `fibonacci.c` and `evolve.c` codes (see instructions in `README.md`)

- If you finish early, try implementing a different mode (ABI – out of line, API – in line)