Creating Python Bindings for Native Code

🎤 **Speaker**: *Gianluca Aguzzi*, University of Bologna

📧 Email: gianluca.aguzzi@unibo.it

🌐 Personal site: https://cric96.github.io/

PDF slides @ https://cric96.github.io/phd-course-python-binding/index.pdf

Outline

- How to handle (conceptually) Python-native interaction
- Main alternatives in the current landscape (see /base-binding-python)
- A guided example with raylib (see /raylib-binding-python)

Creating Bindings from Native Code

Agenda 💭

- What do you want to **expose**?
  - Low level or Pythonic?
- How to **manage** different types?
  - Marshalling?
- How to handle **memory**?
  - GC vs Manual Management

What to Expose?

- It's important to define what you want to expose to the Python side
- Typically, native code **isn't** Pythonic, so you need to create a Pythonic interface
- **Flow**: Native ➡️ Direct Python Binding ➡️ Pythonic Interface

Managing Different Types

- **Marshalling**: the process of transforming data to pass between the two platforms
- Two mindsets:
  - C ➡️ Focused on performance
  - Python ➡️ Focused on simplicity
- Examples:
  - Integers: C has int, short, long, long long; Python has int
  - Floats: C has float, double; Python has float

Managing Memory

- Different paradigms:
  - C ➡️ Manual memory management
  - Python ➡️ Garbage collection
- Key challenges:
  - Memory ownership tracking
  - Cross-language memory management
  - Object lifetime synchronization
- Important considerations:
  - Memory allocation origin
  - Immutability concerns

Main Alternatives

Python offers several ways to create bindings with native code, from completely manual to automatic:

- **ctypes**: Built-in Python library for calling C functions directly
- **cffi**: Modern alternative to ctypes with cleaner API and better performance
- **Cython**: A language that makes writing C extensions for Python as easy as Python itself
- **SWIG**: A code generator for creating bindings in different languages (including Python)

Ctypes

- Built-in Python library for calling C functions directly
  - No need to write C code
  - No need to compile anything
  - Part of the Python standard library
- How it works:
  - Load a shared library
  - Wrap input for C functions (marshalling)
  - Wrap output from C functions (unmarshalling)

How to run

- Create a virtual environment

```
python -m venv venv
source venv/bin/activate
```

- Install the dependencies

```
pip install -r requirements.txt
```

- Build the shared library

```
invoke build-libray
```

- Run the Python script

```
python ctypes_test.py
```

On Invoke

- **Invoke** is a Python library for managing tasks
- It's a simple way to define and run tasks
- It's similar to Makefiles, but in Python
- It's a good way to automate tasks in Python projects
- There are other alternatives like **Ninja**

How to use Invoke

- Install Invoke

```
pip install invoke
```

- Create a file called `tasks.py` with the following content:

```
from invoke import task

@task
def hello(c):
    print("Hello, world!")
```

- Run the task

```
invoke hello
```

Load a Shared Library

Ctypes needs to load a shared library to access C functions

```
import ctypes

# Load the shared library (local)
lib = ctypes.CDLL('path/to/shared/library.so')

# Find a library by name

lib = ctypes.CDLL(find_library("library"))
```

Wrap Input for C Functions

Giving this simple C function:

```
float cmult(int int_param, float float_param)
```

You can call it from Python like this:

```
# Define the function signature
cmult = lib.cmult
cmult.argtypes = [ctypes.c_int, ctypes.c_float]
cmult.restype = ctypes.c_float

# Call the function
result = cmult(2, 3.14)
```

Wrap structs

You can also wrap C structs in Python

```
typedef struct {
    int x;
    float y;
} Point;
```

In python you can define the struct like this:

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_int), ('y', ctypes.c_float)]
```

Pass structs to functions

You can pass structs to C functions

```
void move_point(Point p, int dx, float dy) {
    p-x += dx;
    p.y += dy;
}
```

In Python you can call it like this:

```
move_point = lib.move_point
move_point.argtypes = [Point, ctypes.c_int, ctypes.c_float]
move_point.restype = None

p = Point(1, 2.0)
move_point(p, 3, 4.0)
```

Pass pointer to functions

You can also pass pointers to C functions

```
void move_point(Point *p, int dx, float dy) {
    p->x += dx;
    p->y += dy;
}
```

In Python you can call it like this:

```
move_point = lib.move_point
```

Ctypes Summary

- Pros:
  - Part of the Python standard library
  - No need to write C code
  - No need to compile anything
- Cons:
  - Low level API
  - Limited functionality (Class? Templates?)

CFFI

- Modern alternative to ctypes with an auto-generated API
- Two main modes for creating bindings:
  - ABI mode: Call C functions directly
  - API mode: Use a C header file to generate a Python API

CFFI need to be installed with pip:

```
pip install cffi
```

CFFI Module Creation

- CFFI creates a **full Python module**
- Steps to create CFFI bindings:
    i. Write Python code for bindings
    ii. Generate loadable module
    iii. Import and use the module

Writing Bindings

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

# Process header file
ffi.cdef(header_content)

# Configure source
ffi.set_source(
    "module_name",
    '#include "library.h"',
    libraries=["library"],
    library_dirs=[dir_path],
    extra_link_args=["-Wl,-rpath,."]
)
```