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)
    - For more details, please refer to this guide
- A guided example with raylib (see /raylib-binding-python)

Creating Bindings from Native Code

Agenda 💭

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

Typically, the first two points are managed by the binding library, while the last one is up to the developer

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 (**idiomatic**)
- General guideline:
  - Native interface ➡️ language-specific binding ➡️ Idiomatic (language-based) interface
  - In Python, **Flow**: Native ➡️ Direct Python Binding ➡️ Pythonic Interface
  - Sometimes, the Python Binding is created automatically

What "Idiomatic" Means

- **Idiomatic Code**
  - Code that is natural to the target programming language
  - Follows design principles and community best practices
    - Namely, *idioms* (common patterns) and *conventions* (coding style)
- **Examples**
  - *Effective Java*: Effective Java
  - *The Zen of Python*: The Zen of Python
- **Different Languages, Different Styles**
  - **Python**: Readability, simplicity, elegance
  - **C**: Performance, control, efficiency
- **Creating an Interface** ➡️ Map between idiomatic styles

Managing Different (Platform) 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
- In the *binding* layer, you *may* need to handle these differences

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-library
```

- 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;
    int y;
} Point;
```

In python you can define the struct like this:

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

Pass structs to functions

You can pass structs to C functions

```
void move_point(Point p, int dx, int 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_int]
move_point.restype = None

p = Point(1, 2)
move_point(p, 3, 4)
```

Pass pointer to functions

You can also pass pointers to C functions

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

In Python you can call it like this:

```
move_point = lib.move_point
move_point.argtypes = [ctypes.POINTER(Point), ctypes.c_int, ctypes.c_float]
move_point.restype = None
point = Point(1, 2)
move_point(ctypes.byref(point), 3, 4)
```

❗ it is important to use `ctypes.byref` to pass a pointer to the struct ❗

On Memory Management

- The type definitions created via `ctypes` are **managed** by Python
- When the object is no longer needed, Python will automatically free the memory
- However, if a C function allocates memory, you need to free it manually
    - Otherwise, you will have a memory leak
- Typical solution: expose a function to free the memory

```
void free_point(Point *p) {
    free(p);
}
```

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,."]
)
```

Generating Module

```
ffi.compile()
```

- This will generate a shared library that can be imported in Python using the module name given in `set_source` 🔥
- You don't need to write any manual marshalling code, CFFI will handle it for you 💥
- Unfortunately, CFFI doesn't support C++ 😢
    - Typical workaround: Create a C wrapper around the C++ code

Python wrapper

- Starting from the CFFI module, you can create a Python wrapper

```python
class PointWrapper:
    def __init__(self, x=0, y=0):
        # Allocate memory for a C Point structure
        self._c_point = ffi.new("Point *")
        self.x = x
        self.y = y

    ## Utility methods

    # Functions from Point
    def move(self, dx, dy): #[...]

    def move_in_place(self, dx, dy): #[...]

    def __del__(self):
        ffi.release(self._c_point)  # Explicitly free memory
```

Alternatives?

- **Cython** 🚀: A language that makes writing C extensions for Python as easy as Python itself
  - ⚡ Static compiler that converts Python code to C
  - 📈 Excellent performance for numerical computations
  - 🤝 Can handle both Python and C code seamlessly
  - ➕ Direct support for C++ (unlike CFFI)
  - 🔬 Popular in scientific computing (NumPy, SciPy)
  - ⚠️ Steeper learning curve than ctypes/CFFI

How it works?

- Write a `.pyx` file with a Python-like syntax
- Compile it with Cython
- Import the compiled module in Python

Example

```
# point.pyx
cdef struct Point:
    int x
    float y

cdef class PyPoint:
    cdef Point p

    def __init__(self, x=0, y=0.0):
        self.p.x = x
        self.p.y = y

    def move(self, dx, dy):
        self.p.x += dx
        self.p.y += dy

    @property
    def x(self):
        return self.p.x

    @property
    def y(self):
        return self.p.y
```

And compile with:

```
# setup.py
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("point.pyx")
)
```

or do it manually

```
invoke.run("cython --cplus -3 library.pyx")
invoke.run("g++ -shared -std=c++11 -fPIC $(python3-config --includes) -o library.so library.cpp")
```

Cython Summary

- **Pros** 🔥:
  - Excellent performance 🚀
  - Direct support for C++ ➕➕
  - Seamless integration with Python 🐍
- **Cons** 😢:
  - Steeper learning curve ⛰️
    - Requires to learn ``another'' language 📖
  - Requires compilation step 🛠️
  - Not as easy as ctypes/CFFI 😕

SWIG

- Simplified Wrapper and Interface Generator:
- A code generator for creating bindings in different languages
  - It supports Python, Scala, Java, etc.
- **Pros** 🔥:
  - Supports multiple languages 🌐
  - Can generate bindings automatically ⚙️
  - Can handle C++ code ➕
- **Cons** 😢:
  - Complex to use 🚧
  - Not as popular as ctypes/CFFI/Cython 📉

How can decide which platform to use?

- **ctypes** ➡️ You need a quick and dirty solution
  - you don't want to wrap the entire library
- **cffi**: ➡️ You want an automatic solution with a clean API
  - you have a C header file
- **Cython**: ➡️ You need performance and C++ support
  - you are willing to learn a new language
- **SWIG**: ➡️ You need to support multiple languages

A Concrete Binding Example: Raylib 🎮 🕹️

- **Raylib** is a simple and easy-to-use library to learn videogame programming 🚀
- Written in **C** with focus on clean and efficient API ⚙️
- We will create a **Python binding** for Raylib using ctypes 🐍

Requirements

- Raylib installed on your system
  - follow the instructions here

First level: Native Interface 👷

- **First step**: Create Python binding for Raylib using ctypes 🐍
- Start by selecting core functions to expose 🎯:
  - 🪟 Window management
  - ✏️ Drawing simple text
  - 🧹 Clear the screen functionality
- Design principles in mind: **KISS** (Keep It Simple, Stupid) 💡
  - map the main functions and structure from Raylib to Python

How to?

- Look at the Raylib documentation:
- Load the shared library

```
try:
    lib = ctypes.CDLL(ctypes.util.find_library("raylib"))
except OSError:
    print("Error loading the shared library, try to install it!")
    sys.exit(1)
```

- Extract some main structures

```
class Color(ctypes.Structure):
    _fields_ = [
        ("r", ctypes.c_ubyte),
        ("g", ctypes.c_ubyte),
        ("b", ctypes.c_ubyte),
        ("a", ctypes.c_ubyte)
    ]
```

How to? 🔧

- Define the functions you want to expose with **ctypes**
  - It's **recommended** to explicitly define parameters and return types

```
init_window = lib.InitWindow
init_window.argtypes = [ctypes.c_int, ctypes.c_int, ctypes.c_char_p]
init_window.restype = None
```

- Why Types? 🤔
  - 🛡️ Prevents errors in marshalling/unmarshalling
  - ♻️ Helps avoid memory leaks
  - 📖 Makes code self-documenting and more readable
  - ⚠️ **Remember**: Clear type definitions are crucial for reliable native bindings!

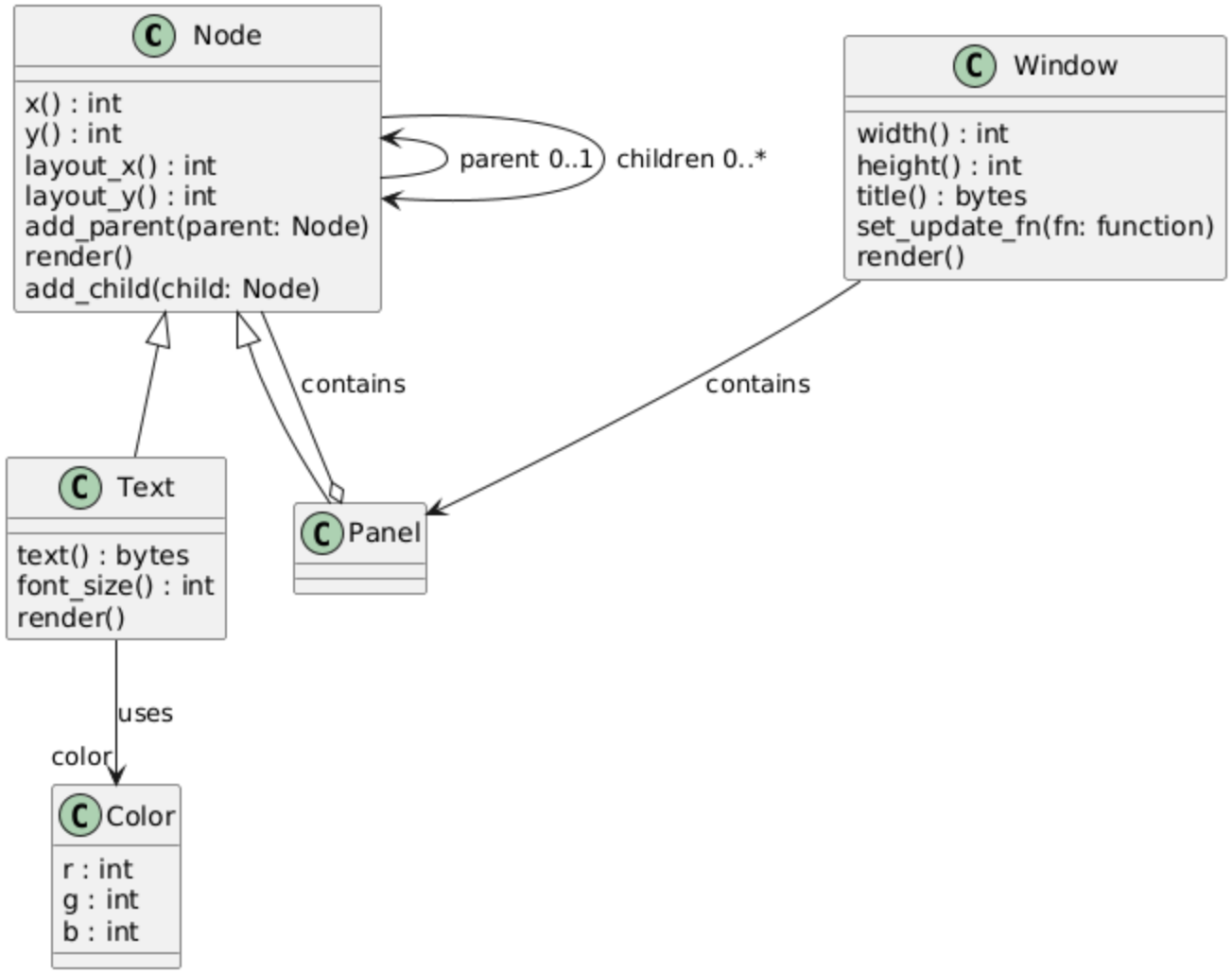
Simple example

```
width = 800
height = 450
fps = 60
speed = 10
font_size = 40
text_y = 10

init_window(width, height, b"Hello, World!")
set_target_fps(fps)

dx = 0
while not window_should_close():
    begin_drawing()
    clear_background(BLACK)
    draw_text(b"Hello, World!", dx, text_y, font_size, LIGHTGRAY)
    end_drawing()
    dx = (dx + speed) % width
close_window()
```

Second level: Pythonic Interface

- Once we have the basic bindings, we can design a more Pythonic interface leveraging
  - **Classes** to encapsulate data and behavior
  - First-class functions to simplify the API
- In this case, I create the following classes:
  - **Window**: to manage the window
    - Everything about window management
  - **Node**: to manage the drawing
    - Everything about drawing and structure

Overall Picture



Pythonic example

```
width = 800
height = 450
font_size = 40
text_y = 10
dx = 10

panel = dsl.Panel()
text = dsl.Text("Hello, World!", 0, 0, font_size, LIGHTGRAY)
text2 = dsl.Text("Hello, World!", 0, font_size + text_y, font_size, LIGHTGRAY)

def update():
    panel.x = (panel.x + random.randint(0, dx)) % width
    panel.y = (panel.y + random.randint(0, dx)) % height

panel.add_children([text, text2])
window = dsl.Window(width, height, "Hello, World!", panel)
window.update_fn = update
window.render()
```

Assignment

- **Goal**: Develop bindings between two different platforms, such as:
  - **Native** (C/C++) ➡️ **Python**, or **Native** ➡️ **Kotlin**, or **Python** ➡️ **Java**
- **Requirements**:
  - **Wrapped** code containing at least one function and one type definition (class/interface). Refer to awesome C libraries for examples.
  - Create an idiomatic interface in the target language.
- **Deliverables**:
  - A **GitHub repository** containing *the code* and a *report* (README.md) that includes (at least):
    - **Design choices** explaining what makes the interface idiomatic