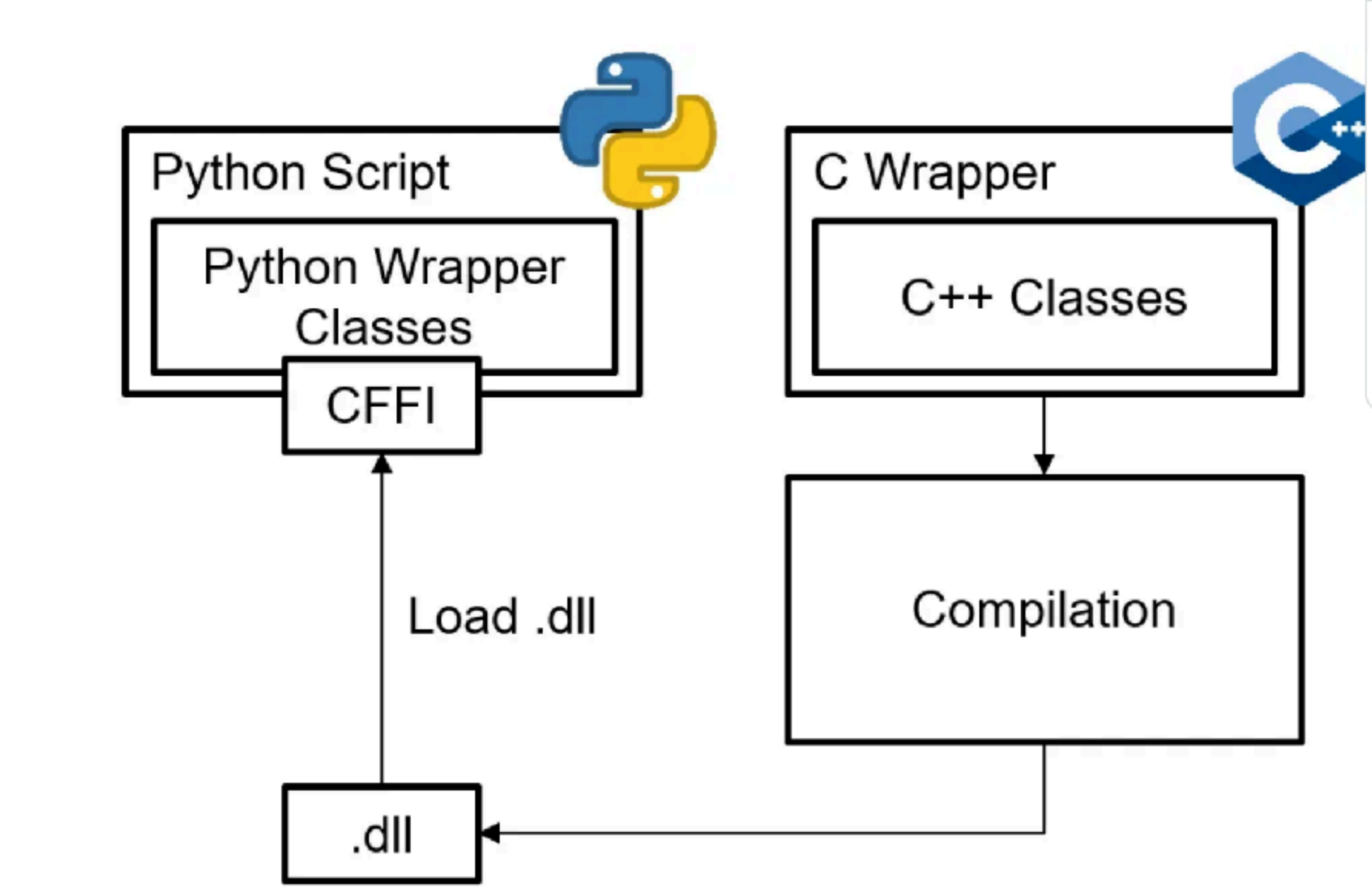
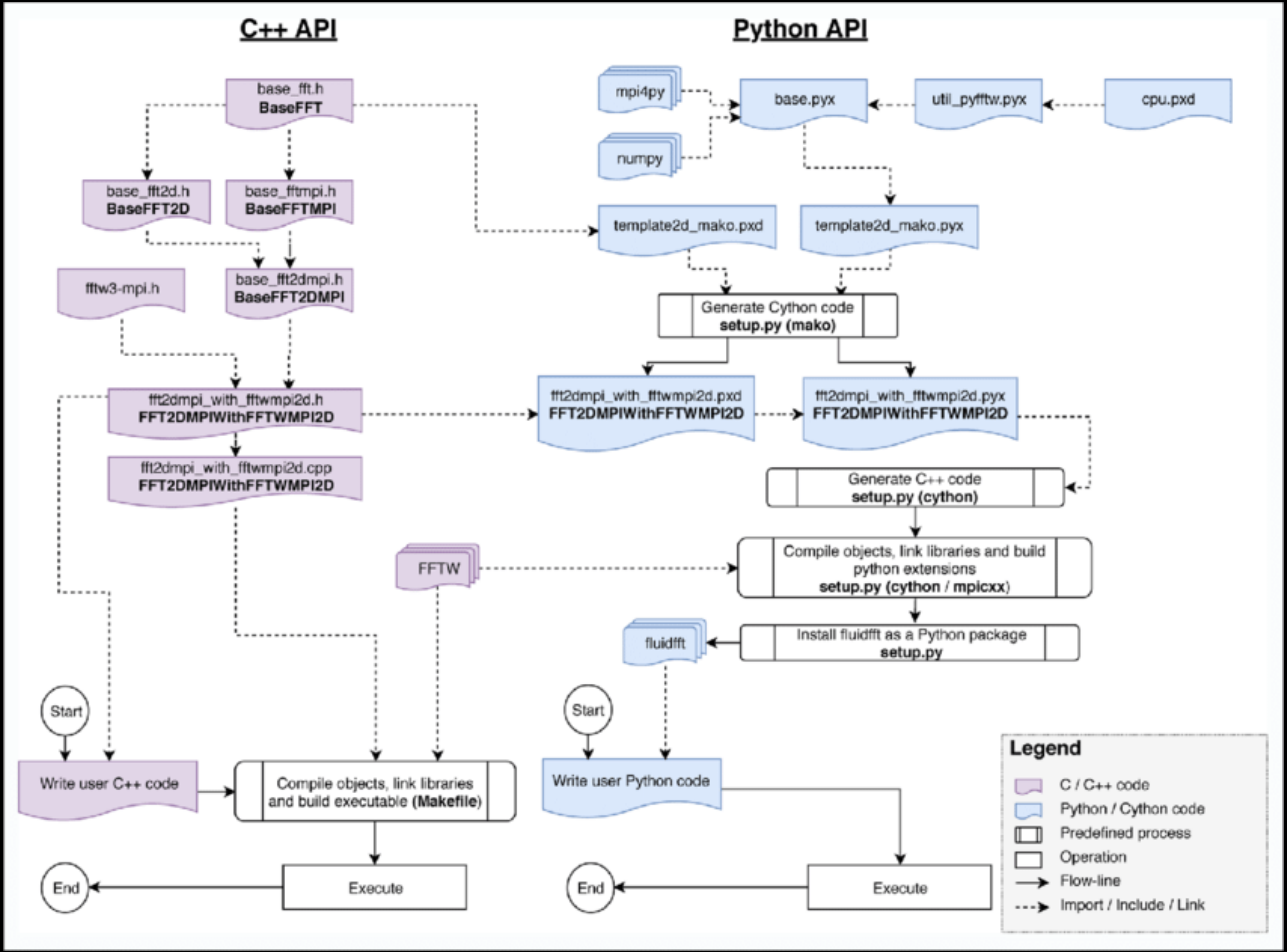
Advanced Programming

Python-C Interface

Ramaseshan Ramachandran



Principle of Interfacing C/C++ and Python


C++ API
Python API
base_fft.h
BaseFFT
base_fft2d.h
BaseFFT2D
base_fftmpi.h
BaseFFTMPI
fftw3-mpi.h
base_fft2dmpi.h
BaseFFT2DMPI
fft2dmpi_with_fftwmpi2d.h
FFT2DMPIWithFFTWMPI2D
fft2dmpi_with_fftwmpi2d.cpp
FFT2DMPIWithFFTWMPI2D
FFTW
Start
Write user C++ code
Compile objects, link libraries and build executable (Makefile)
Execute
End
mpi4py
numpy
base.pyx
util_pyfftw.pyx
cpu.pxd
template2d_mako.pxd
template2d_mako.pyx
Generate Cython code
setup.py (mako)
fft2dmpi_with_fftwmpi2d.pxd
FFT2DMPIWithFFTWMPI2D
fft2dmpi_with_fftwmpi2d.pyx
FFT2DMPIWithFFTWMPI2D
Generate C++ code
setup.py (cython)
Compile objects, link libraries and build python extensions
setup.py (cython / mpicxx)
fluidfft
Install fluidfft as a Python package
setup.py
Start
Write user Python code
Execute
End
Legend
C / C++ code
Python / Cython code
Predefined process
Operation
Flow-line
Import / Include / Link

PYTHON HIGHLIGHTS

- In order to understand Python-C interface it is important understand basic types of Python
- Dynamic typing – type of variable can change during the execution of the program
- Strongly typed – Enforces type-checking rules to prevent operations between incompatible types
  - To promote predictable behaviour
- Object-oriented types – all basic types and built-in types are object orientedi
  - Operations are performed through methods
  - Allows for powerful abstraction and encapsulation
  - Easy to work with complex data structures and behaviours
- Type Annotations
  - To specify the expected types of variables, function parameters, and return values
  - Enhances code readability

WHY C?

- Unmatched Performance
  - Excels in computationally intensive tasks.
  - Achieves significant speed improvements for numerical computations, image processing, or other intensive workloads
- Fine-grained Control
  - Grants direct access to hardware resources and system functionalities
  - Ability to interact with device drivers
  - Manages memory with greater precision
- Well Established Libraries
  - Leverage a vast treasure trove of functionalities in C libraries
  - Seamless access to these libraries without the need for redundant development efforts.

THE NEED FOR AN INTERFACE

- Integration
  - Take advantage of best of both platforms
    - C/C++ to develop Performance-critical library
    - Python for rapid development and ease of use
- Reuse
    - Reuse this code without having to rewrite it entirely in Python
- Performance
- Wrapping
  - Leverage this performance of lower-level language while working in Python
- Ecosystem
  - Bridge the gap between Python and C ecosystems, enabling interoperability and extending the capabilities

ADVANTAGES

- Performance
- Leverage existing libraries of C/C++
- Extend Python capabilities
- System-level interfaces from Python
- Cross-platform compatibility
- Rapid prototyping
- Extending low-level libraries by adding functionalities

How do we integrate?

C WRAPPER

- Bridge the gap between their different levels of abstraction
- C offers fine-grained control over memory management and hardware interaction
  - Closer to the machine, but requires significant effort
- Python abstracts away many of these complexities of C
  - Memory management and garbage collection is done automatically
  - Less efficient for specific hardware interactions
- A wrapper interfaces exposes performance-critical functionalities in C to Python as Python modules

INTERFACE BETWEEN TWO LANGUAGES

- Foreign Function Interface (FFI)
  - Allows code written in one programming language to call functions or use data structures defined in another language
- Data Representation
  - Python and C have different data representations for fundamental types and built-in types
- Calling Conventions – To establish a good communication mechanism between languages
  - Data types in Python and C do not have a one-to-one correspondence, necessitating careful handling during communication.
  - Marshal function calls are made, parameters are passed and return values are handled

INTERFACE BETWEEN TWO LANGUAGES

- Memory Management
  - Python manages memory automatically while C requires manual memory management
- Error Handling
  - Python and C++ uses exceptions to handle errors
  - C often uses error codes through return values.
  - Graceful error handling is required
- Bindings
  - To facilitate communication between Python and C code and abstract the details of interfacing between the two languages
- Performance
  - Improve performance and develop rapidly

CYTHON

- An open-source programming language
  - It is a compiled language that translates Python-like code into C
- Cython is a superset of Python
  - Combines the ease of Python syntax with the speed of C
  - Allows developers to write C extensions for Python using a syntax that is very similar to Python

FEATURES OF CYTHON

- Static Typing
  - Static type annotations - enabling better type inference and optimization
- C-Level Integration
  - Cython provides seamless integration with existing C code,
    - Allows developers to call C functions and work with C data types directly from Cython code
- Automatic Conversion
  - Cython automatically converts Python code to C
- Easy Python Integration
  - Can be easily integrated with existing Python codebases to allow reuse and optimisation

CYTHON CODE

```
# distutils: language=c++


from libcpp.vector cimport vector

def primes(unsigned int nb_primes):
    cdef int n, i
    cdef vector[int] p
    p.reserve(nb_primes)  # allocate memory for 'nb_primes' elements.

    n = 2
    while p.size() < nb_primes:  # size() for vectors is similar to len()
        for i in p:
            if n % i == 0:
                break
        else:
            p.push_back(n)  # push_back is similar to append()
        n += 1

    # If possible, C values and C++ objects are automatically
    # converted to Python objects at need.
    return p  # so here, the vector will be copied into a Python list.
```

Source: https://cython.readthedocs.io/en/stable/src/tutorial/cython_tutorial.html#primes-with-c

CALLING C FUNCTIONS

```
from libc.math cimport sin


cdef double f(double x):
    return sin(x * x)
```

```
from cython.cimports.libc.math import sin

@cython.cfunc
def f(x: cython.double) -> cython.double:
    return sin(x * x)
```

FOREIGN FUNCTION INTERFACE (FFI)

- This is a popular method that allows calling C functions directly from Python code.
- A Translation of types and function calls between the languages
- Python code can access C functions as if they were native Python functions
  - Maintains underlying efficiency of C

CTYPES

Interoperability Challenge

- Difficulties in direct interaction between Python and C
- High-level vs. low-level paradigms
- Data type discrepancies
- Memory management differences

Type - Bridging C Libraries in Python

- **Enables function calls from Python to C libraries**
- **Handles data type conversions automatically**
- Provides memory management mechanisms

ADVANTAGES OF TYPES

- Key advantages for developers:
- Leverage existing C libraries without rewriting them in Python
- Enhance performance for computationally intensive tasks
- Integrate with hardware-related functionality exposed by C libraries

HOW DOES IT WORK?

- Load C library using CTypes functions (e.g., cdll)
- Define function prototypes matching C functions
- Call C functions from Python with appropriate data types
- Handle returned data and potential errors
- There is no need to install additional libraries or compilers – built into Python

C LIBRARY - READ_FILE

```c
char* read_file(const char* filename) {
  size_t bytes_read = 0;

  FILE* fp = fopen(filename, "r");
  if (fp == NULL) {
    printf("Couldn't open file %s\n", filename);
    return NULL;
  }

   /* Code not shown */

  fclose(fp);

  return content;
}

void free_file_content(char* content) {
  if (content != NULL) {
    free(content);
  }
}
```

```python
from ctypes import *

# Load the shared library
lib = CDLL("./libread_file.so")

# Define the function signatures
read_file_func = lib.read_file
read_file_func.restype = c_char_p
read_file_func.argtypes = [c_char_p]

filename = "./read_fle.c".encode('utf-8')

content_ptr = read_file_func(filename)

if content_ptr:
  content = content_ptr.decode('utf-8')
  print(f"File content:\n{content}")
else:
  print("Error reading file")

#Free the allocated memory (crucial in Python)
```

BUILD SCRIPT

```bash
#!/bin/bash

source_file="$1"

if [ -z "$source_file" ]; then
  echo "Error: Please provide a source file name as an argument."
  exit 1
fi

extension="${source_file##*.}"

if [[ "$extension" == "cpp" ]]; then
  clang++ -c -std=c++20 -fPIC "$source_file" -o "${source_file%.cpp}.o"
  clang++ -shared -o "lib${source_file%.cpp}.so" "${source_file%.cpp}.o"
elif [[ "$extension" == "c" ]]; then
  clang -c -fPIC "$source_file" -o "${source_file%.c}.o"
  clang -shared -o "lib${source_file%.c}.so" "${source_file%.c}.o"
else
  echo "Error: Unsupported file extension"
  exit 1
fi

echo "Built library for ${source_file}"
```

POSITION INDEPENDENT CODE

- The -fPIC option in the clang/gcc compiler stands for Position-Independent Code
  - Instruction to the compiler to at generate code that can be loaded and executed at any memory address in the program's address space.
  - This is crucial for creating shared libraries/DLLs
  - Requires the smallest amount of page modification at runtime