Interfacing Native Code (C/C++) with Python

By Jayaprakash Nevara (JP Shivakavi)

About Me

Jayaprakash Nevara (Linked In)

- BE in CSE from NITK, Surathkal, Karnataka.
- 19+ years of Experience in Software & Firmware Development,
- Prior to joining Intel associated with – Cisco, Philips, Zilog and TCS

Areas of Work

- Platform Debug / Validation Tools
- Ported & open sourced Py3 Interpreter for UEFI to Tianocore
- Layer 2 N/W Protocol development
- Embedded Systems Design and Development
- Device Drivers and Linux Kernel Modules
- Mainframes’ Applications development

Hobbies / Interests

- Learning Languages (Natural as well as Computer)
- Reading Novels – Mostly Kannada
- Poetry writing in Kannada.
- Translations
- http://bayalasiriballari.blogspot.in/

Objectives

- Methods to create basic **extension modules** to **interface native code (C/C++) with python**
- Build a working sample with **Python C APIs** and **Ctypes**
- Pros and cons of using one method over the other
- When to use which method for interfacing native code

Agenda

- Introduction
- Methods of using/interfacing native with Python
  - Python C API
  - Ctypes – foreign function library
- Python C API - samples
  - HelloWorld extension module (Hands on)
  - HWAPILib extension module (Code walk through from git repo / self study)
- Ctypes – Samples
  - HelloWorld example (Hands on)
  - Invoking APIs from HWAPILib (walk through of code from git repo / self study)
- Q&A

Why does Python live on land?

Introduction

Python Vs Native Code (Read it as C/C++)

Python	Native code
• Lots of standard library, Highly flexible, concise programs, rapid application development • Code execution is slow - Interpreted • Doesn't have mechanism to directly interact with the hardware	• Bigger code size, development may be slower compared to Python • Execution is fast - compiled and optimized to a specific m/c architecture • Better control on system/device level interaction

Why interface native code with Python?

To accelerate execution

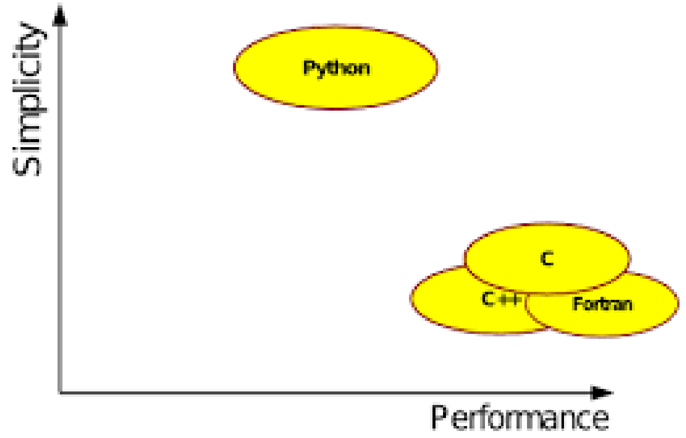
- Runs faster
- Provides better performance

To wrap existing functionality in C

- Expose existing C interfaces to Python
- Can expose C interfaces or a polished “Pythonic” interface

Low level system access

- To have access to low level features of CPython runtime, the Operating System
- To access underlying hardware (network device, video/audio device and etc)



Take advantage of simplicity of Python and Performance of C by putting them to work together

Python Modules

A Python module is a file containing Python definitions and statements.

- Can define functions, classes, and variables.
- Can include runnable code.

Grouping related code into a module makes the code easier to understand and use.
It also makes the code logically organized & reusable

Types of Python Modules

- **Extension Module** **(Will be discussed in this training)**
  - Written in C / C++
  - Built as .PYD file
  - Loaded dynamically on module import
- Built-in Module
  - Written in C / C++
  - Built into Python interpreter – Static Linking
- Pure Python Module
  - Written in Python
  - File with .py extension

The way of working with the modules remains same irrespective of the type of module

import - All types of modules can be used in python applications with the help of import statements

Methods of Interfacing C/C++ (Native) code with Python

Python C API

- Built in module
- Extension module

C-types

- Extension module

Python C APIs

APIs to interface C/C++ code to Python and vice versa

- Used for writing an ***extension module*** – widely used for this purpose
  - Interface native code – such as C/C++
  - Extend python interpreter capabilities
  - Define new functions, object types and methods which work on those objects
  - Compile it as python DLLs (.pyd) or Compile into Python interpreter as built-in modules
  - Extension modules can be hand coded or auto generated
  - Third party tools for auto generation: SWIG, Cython, Numba, cffi and etc **(not discussed in this session)**
- Defines C APIs to use python as a component in a larger application – referred to as ***embedding python*** in applications (C/C++)

https://docs.python.org/3/c-api/intro.html

Extension Modules: Build Environment

- Windows OS : Official C-python Build tool **Visual C++ compiler from Microsoft**
- Extension modules should be compiled with the same tool

Python version	Visual Studio Compiler tool set	Comments
Python 3.5 and 3.6	Visual Studio 2015	
Python 3.9 & 3.10	Visual Studio 2019	
Python 3.11+	Visual Studio 2022	

Windows Tool Chain for Python 3.11 and 3.12

```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\njayapra>set path=c:\Python311;%path%

C:\Users\njayapra>python
Python 3.11.6 (tags/v3.11.6:8b6ee5b, Oct  2 2023, 14:57:12) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

C:\Users\njayapra>set path=c:\python312;%path%

C:\Users\njayapra>python
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

MSV version and Visual studio tool chain mapping

For this version of Visual C++	Use this compiler version
Visual C++ .NET 2003	MSC_VER=1310
Visual C++ 2005 (8.0)	MSC_VER=1400
Visual C++ 2008 (9.0)	MSC_VER=1500
Visual C++ 2010 (10.0)	MSC_VER=1600
Visual C++ 2012 (11.0)	MSC_VER=1700
Visual C++ 2013 (12.0)	MSC_VER=1800
Visual C++ 2015 (14.0)	MSC_VER=1900
Visual C++ 2017 (15.0)	MSC_VER=1910
Visual C++ 2017 (15.3)	MSC_VER=1911
Visual C++ 2019 (16.9)	MSC_VER=1929
Visual C++ 2019 (16.10)	MSC_VER=1929
Visual C++ 2019 (16.11)	MSC_VER=1929
Visual C++ 2022 (17.0.1)	MSC_VER =1930

Extension Modules: Build Environment Setup Windows

Install visual studio 2022 – Download, choose "Desktop development with C++" workload while installing VS2022

Linux you will need to use the GCC tool chain which comes by default installed on most of the Linux based OSes

Install Python 3.12.x from https://www.python.org/downloads/release/python-3121/

Ubuntu Linux you may follow the instructions provided in this link

https://phoenixnap.com/kb/how-to-install-python-3-ubuntu

Install build, wheel modules by using the commands

python –m pip install build # for Linux replace python with python3

Python –m pip install wheel # for Linux replace python with python3

Python –m pip install setuptools # for Linux replace python with python3

Extension Modules – General Structure

The code in an extension module is organized in following sections:

- **Header section** – python.h should be included
  - Header file gives access to internal python API used to hook your module into the interpreter
  - This should be the first header file in your extension module
- **Definition of C functions**
  - All the C functions that you want to implement
- **Method mapping table**
  - A table mapping the names of functions between C and Python
- **Module definition Structure**
  - Defines the module with name, documentation and the set of functions exposed to Python world
- **An initialization function**
  - Stitches all the pieces together and hooks the module to Python interpreter at runtime

Definition of C Functions

The C Functions takes one of the following forms:

```
static PyObject *MyModule_MyFunction( PyObject *self, PyObject *args );
static PyObject *MyModule_MyFunctionWithKeywords(PyObject *self, PyObject *args,
PyObject *kw);
static PyObject *MyModule_MyFunctionWithNoArgs( PyObject *self );
```

Example:

```
static PyObject *MyModule_func(PyObject *self, PyObject *args)
{
    #MyModule is the name of module here
     /* Do your stuff here. */
     Py_RETURN_NONE;
}
```

MyModule -> Name of the module; func -> name of the function in the module

Method Mapping Table

It’s simply an array of PyMethodDef structure

```
  static struct PyMethodDef {
  char *ml_name;
  PyCFunction ml_meth;
  int ml_flags;
  char *ml_doc;
}
```

Member	Description
ml_name	This is the name of function as the python interpreter presents when it used in python programs
ml_meth	Address to a function that has any one of the signatures described previously
ml_flags	Tells the interpreter which one of the three signatures ml_meth is using METH_VARARGS, METH_KEYWORDS, METH_NOARGS
ml_doc	Documentation string for the function, it can be NULL

This table needs to be terminated with a sentinel that consists of NULL and 0 values for the appropriate members Example

Extension Modules: Method Mapping Table

Sample Method Table:

```
static PyMethodDef MyModule_methods[] = {
 { "func", MyModule_func, METH_NOARGS, "sample module function" },
 { NULL, NULL, 0, NULL }
 };
```

Extension Modules: Module Definition

Module Definition

```
static struct PyModuleDef MyModulemodule = {
    //MyModule is the name of module
    PyModuleDef_HEAD_INIT,
    "MyModule",    /* name of module */
    NULL, /* module documentation, may be NULL */
    -1,       /* size of per-interpreter state of the module,
                 or -1 if the module keeps state in global
                variables. */
    MyModule_methods   /* Module methods exposed to Python */
};
```

Initialization Function

- Called by python interpreter when the module is loaded (done as part of import operation)
- Named as ***PyInit_MyModule() –*** *where* ***MyModule*** *is the name of the module*
- Needs to be exported from the library you will be building
- Use PyMODINIT_FUNC defined in python headers to export the initialization function to Python Interpreter
- General structure of module initialization function

```
PyMODINIT_FUNC PyInit_MyModule() {
    // Module initialization code here
}
```

PyMODINIT_FUNC declares the function as PyObject * return type

Module Initialization

Create a python module using ***PyModule_Create()*** function

Create an Exception object using ***PyErr_NewException()*** function **(Optional)**

Add the module object to the list of modules maintained by Python Interpreter using ***PyModule_AddObject()***

Example:

```
PyMODINIT_FUNC PyInit_MyModule(void)
{
   PyObject *m;
   m = PyModule_Create(&MyModulemodule);    # Create a python module
   if (m == NULL)
      return NULL;
   MyModuleError = PyErr_NewException("MyModule.error", NULL, NULL);   # Create an exception object
   Py_INCREF(MyModuleError );
   PyModule_AddObject(m, "error", MyModuleError );    # register Python module along with Exception obj
   return m;
}
```

Writing hello module

hello – a python extension module

Let’s write a simple “hello” extension module –

- This module makes a call to a C routine hello() from python application.
- The C function takes no arguments and returns nothing
- It prints the following string “Hello from Python Extension Module!!!” when called from python.

hello – Header Section

```c
#include <Python.h>
#include <stdio.h>
// Exception from hello module
static PyObject *HelloError;
```

// Function which prints “Hello from Python module!!!”

```c
static PyObject * hello_sayhello(PyObject *self) {
	printf("Hello From Python Extension Module!!!");
	Py_RETURN_NONE;
}
```

There is no void function in Python – it should return None type hence using Py_RETURN_NONE

hello – Method Definition Table

```
static PyMethodDef HelloMethods[] = {
   {"sayhello" , (PyCFunction)hello_sayhello, METH_NOARGS,
     "Prints hello message from extension module"},
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

hello – Module Definition

```
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT,
    "hello",      /* name of module */
    "Sample hello module", /* module documentation, may be NULL */
    -1,           /* size of per-interpreter state of
                     the module, or -1 if the module
                     keeps state in global variables.
                  */
    HelloMethods  // Methods table of the module
};
```

hello - Module Initialization

```
PyMODINIT_FUNC PyInit_hello(void)
{
    PyObject *m;
    m = PyModule_Create(&hellomodule);
    if (m == NULL)
        return NULL;
    HelloError = PyErr_NewException("hello.error", NULL, NULL);
    Py_INCREF(HelloError);
    PyModule_AddObject(m, "error", HelloError);
      return m;
}
```

Building and Installing Extension modules

- **setuptools** – Provides a standard mechanism to build and distribute pure python as well as extension modules
- setup.py – is required for providing information about the C source files from which the extension module will be built, any other libraries that needs to be linked along with all compiler and linker flags

setup.py:

```
from setuptools import setup, Extension
setup( ext_modules=[Extension(name = 'hello’, sources = ['hello.c'])]
       )
```

[setuptools documentation]

Building and Installing Extensions …

Launch the Visual Studio Commandline Tool(x64 as our interpreter is of type x64), **Linux you can launch a terminal**

Add python installer path to the path variable

Go to the directory where your module source code and setup.py scripts are stored

Run the following command

python –m build -- no-isolation # this command builds the python extension module

***python –m pip install* <name_of_module>** # this command installs the python module

Command Prompt

Building log from hello module

```
C:\native-code-with-python-samples\python_c_api_samples\Hello>python -m build --no-isolation
* Getting build dependencies for sdist...
running egg_info
writing hello.egg-info\PKG-INFO
writing dependency_links to hello.egg-info\dependency_links.txt
writing top-level names to hello.egg-info\top_level.txt
reading manifest file 'hello.egg-info\SOURCES.txt'
writing manifest file 'hello.egg-info\SOURCES.txt'
* Building sdist...
running sdist
running egg_info
writing hello.egg-info\PKG-INFO
writing dependency_links to hello.egg-info\dependency_links.txt
writing top-level names to hello.egg-info\top_level.txt
reading manifest file 'hello.egg-info\SOURCES.txt'
writing manifest file 'hello.egg-info\SOURCES.txt'
warning: sdist: standard file not found: should have one of README, README.rst, README.txt, README.md

running check
creating hello-0.42
creating hello-0.42\hello.egg-info
copying files to hello-0.42...
copying hello.c -> hello-0.42
copying pyproject.toml -> hello-0.42
copying setup.py -> hello-0.42
copying hello.egg-info\PKG-INFO -> hello-0.42\hello.egg-info
copying hello.egg-info\SOURCES.txt -> hello-0.42\hello.egg-info
copying hello.egg-info\dependency_links.txt -> hello-0.42\hello.egg-info
copying hello.egg-info\top_level.txt -> hello-0.42\hello.egg-info
```

Command Prompt

Build log from hello module cont..

```
Creating tar archive
removing 'hello-0.42' (and everything under it)
* Building wheel from sdist
* Getting build dependencies for wheel...
running egg_info
writing hello.egg-info\PKG-INFO
writing dependency_links to hello.egg-info\dependency_links.txt
writing top-level names to hello.egg-info\top_level.txt
reading manifest file 'hello.egg-info\SOURCES.txt'
writing manifest file 'hello.egg-info\SOURCES.txt'
* Building wheel...
running bdist_wheel
running build
running build_ext
building 'hello' extension
creating build
creating build\temp.win-amd64-cpython-312
creating build\temp.win-amd64-cpython-312\Release
"C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\Tools\MSVC\14.40.33807\bin\HostX86\x64\cl
.exe" /c /nologo /O2 /W3 /GL /DNDEBUG /MD -Ic:\Python312\include -Ic:\Python312\Include "-IC:\Program Fi
les\Microsoft Visual Studio\2022\Professional\VC\Tools\MSVC\14.40.33807\include" "-IC:\Program Files\Mic
rosoft Visual Studio\2022\Professional\VC\Tools\MSVC\14.40.33807\ATLMFC\include" "-IC:\Program Files\Mic
rosoft Visual Studio\2022\Professional\VC\Auxiliary\VS\include" "-IC:\Program Files (x86)\Windows Kits\1
0\include\10.0.26100.0\ucrt" "-IC:\Program Files (x86)\Windows Kits\10\\include\10.0.26100.0\\um" "-IC:\
Program Files (x86)\Windows Kits\10\\include\10.0.26100.0\\shared" "-IC:\Program Files (x86)\Windows Kit
s\10\\include\10.0.26100.0\\winrt" "-IC:\Program Files (x86)\Windows Kits\10\\include\10.0.26100.0\\cppw
inrt" "-IC:\Program Files (x86)\Windows Kits\NETFXSDK\4.8\include\um" /Tchello.c /Fobuild\temp.win-amd64
-cpython-312\Release\hello.obj
hello.c
creating C:\Users\njayapra\AppData\Local\Temp\build-via-sdist-fgc59osm\hello-0.42\build\lib.win-amd64-cp
```

Command Prompt

```
ython-312
"C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\Tools\MSVC\14.40.33807\bin\HostX86\x64\li
nk.exe" /nologo /INCREMENTAL:NO /LTCG /DLL /MANIFEST:EMBED,ID=2 /MANIFESTUAC:NO /LIBPATH:c:\Python312\li
bs /LIBPATH:c:\Python312 /LIBPATH:c:\Python312\PCbuild\amd64 "/LIBPATH:C:\Program Files\Microsoft Visual
 Studio\2022\Professional\VC\Tools\MSVC\14.40.33807\ATLMFC\lib\x64" "/LIBPATH:C:\Program Files\Microsoft
 Visual Studio\2022\Professional\VC\Tools\MSVC\14.40.33807\lib\x64" "/LIBPATH:C:\Program Files (x86)\Win
dows Kits\NETFXSDK\4.8\lib\um\x64" "/LIBPATH:C:\Program Files (x86)\Windows Kits\10\lib\10.0.26100.0\ucr
t\x64" "/LIBPATH:C:\Program Files (x86)\Windows Kits\10\\lib\10.0.26100.0\\um\x64" /EXPORT:PyInit_hello
build\temp.win-amd64-cpython-312\Release\hello.obj /OUT:build\lib.win-amd64-cpython-312\hello.cp312-win_
amd64.pyd /IMPLIB:build\temp.win-amd64-cpython-312\Release\hello.cp312-win_amd64.lib
   Creating library build\temp.win-amd64-cpython-312\Release\hello.cp312-win_amd64.lib and object build\
temp.win-amd64-cpython-312\Release\hello.cp312-win_amd64.exp
Generating code
Finished generating code
installing to build\bdist.win-amd64\wheel
running install
running install_lib
creating build\bdist.win-amd64
creating build\bdist.win-amd64\wheel
copying build\lib.win-amd64-cpython-312\hello.cp312-win_amd64.pyd -> build\bdist.win-amd64\wheel\.
running install_egg_info
running egg_info
writing hello.egg-info\PKG-INFO
writing dependency_links to hello.egg-info\dependency_links.txt
writing top-level names to hello.egg-info\top_level.txt
reading manifest file 'hello.egg-info\SOURCES.txt'
writing manifest file 'hello.egg-info\SOURCES.txt'
Copying hello.egg-info to build\bdist.win-amd64\wheel\.\hello-0.42-py3.12.egg-info
running install_scripts
c:\Python312\Lib\site-packages\wheel\bdist_wheel.py:108: RuntimeWarning: Config variable 'Py_DEBUG' is u
```

Build log from hello module cont..

Build log from hello module cont..

```
c:\Python312\Lib\site-packages\wheel\bdist_wheel.py:108: RuntimeWarning: Config variable 'Py_DEBUG' is u
nset, Python ABI tag may be incorrect
  if get_flag("Py_DEBUG", hasattr(sys, "gettotalrefcount"), warn=(impl == "cp")):
creating build\bdist.win-amd64\wheel\hello-0.42.dist-info\WHEEL
creating 'C:\native-code-with-python-samples\python_c_api_samples\Hello\dist\.tmp-e10qje35\hello-0.42-cp
312-cp312-win_amd64.whl' and adding 'build\bdist.win-amd64\wheel' to it
adding 'hello.cp312-win_amd64.pyd'
adding 'hello-0.42.dist-info/METADATA'
adding 'hello-0.42.dist-info/WHEEL'
adding 'hello-0.42.dist-info/top_level.txt'
adding 'hello-0.42.dist-info/RECORD'
removing build\bdist.win-amd64\wheel
Successfully built hello-0.42.tar.gz and hello-0.42-cp312-cp312-win_amd64.whl

C:\native-code-with-python-samples\python_c_api_samples\Hello>
```

hello module : installation log

```
C:\native-code-with-python-samples\python_c_api_samples\Hello\dist>python -m pip install hello-0.42-cp31
2-cp312-win_amd64.whl
Defaulting to user installation because normal site-packages is not writeable
Processing c:\native-code-with-python-samples\python_c_api_samples\hello\dist\hello-0.42-cp312-cp312-win
_amd64.whl
Installing collected packages: hello
Successfully installed hello-0.42
```

Using the extension module

import and use this module in any python scripts

Example:

import hello

help(hello) # displays the help information for hello module

x=hello.sayhello() # prints “Hello From Python Extension Module!!!”

```
Python 3.12.6 (tags/v3.12.6:a4a2d2b, Sep  6 2024, 20:11:23) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello
>>> hello.sayhello()
Hello from hello DLL
>>>
```

Sample Extension Module to interface CPP code

HelloCpp module

A sample example extension module to interface CPP code with Python

Let’s do the following:

Code walk through

build, install and exercise the module

Python/C API

Advantages	Disadvantages
• Requires no additional libraries • Lots of low-level control • Build support using setuptools through setup.py • Most of the standard extension modules are written using Python/C APIs	• Wrapping code is written in C • May require a substantial amount of effort • Requires regular maintenance • No forward compatibility across Python versions as Python C-APIs changes • Reference count bugs are easy to create and very hard to track down.

For further details use this Python/C API reference documentation

https://docs.python.org/3/c-api/index.html

Part -2

ctypes

Ctypes - Introduction

Foreign function interface library for Python

Provides C Compatible data types

Can invoke APIs from DLLs / shared libraries

Simple and Quick way to integrate DLLs with Python

Ctypes - introduction

Exposes **cdll** and **windll** (for Windows OS) objects

Use these objects to load DLLs by calling **LoadLibrary**()

cdll -> for DLLs using calling convention **cdecl (C Declaration)**

windll -> for DLLs using calling convention **stdcall**

Functions from loaded DLLs can be accessed as **attributes** of dll objects

Calling Conventions

Calling conventions specify

- how arguments are passed to a function,
- how return values are passed back out of a function,
- how the function is called, and
- how the function manages the stack and its stack frame.

In short, the calling convention specifies how a function call in C or C++ is converted into assembly language.

* For further study on calling conventions : https://en.wikipedia.org/wiki/Calling_convention

Ctypes – fundamental data types

C type	ctypes type	Python type
char	**c_char**	1-character bytes object
wchar_t	**c_wchar**	1- character string
char	**c_byte**	int
unsigned char	**c_ubyte**	Int
short	**c_short**	Int
unsigned short	**c_ushort**	Int
int	**c_int**	Int
unsigned int	**c_uint**	Int
long	**c_long**	Int
unsigned long	**c_ulong**	Int
__int64 or long long	**c_longlong**	Int
unsigned __int64 or unsigned long long	**c_ulonglong**	Int
float	**c_float**	Float
double	**c_double**	Float
char * (NUL terminated)	**c_char_p**	bytes object or None
wchar_t * (NUL terminated)	**c_wchar_p**	string or None
void *	**c_void_p**	int or None

https://docs.python.org/3/library/ctypes.html

Working with Ctypes module

Calling functions

```
>>> from ctypes import *
>>> libc=windll.msvcrt
>>> printf=libc.printf
>>> printf(b"Hello World!")
Hello World!12
>>> printf(b"Hello %s\n", b"World!")
Hello World!
13
>>> printf(b"Value of PI = %f\n", 3.412)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: <class 'TypeError'>: Don't know how to convert parameter 2
>>>
>>> printf(b"Value of PI = %f\n", c_double(3.412))
Value of PI = 3.412000
```

Calling functions with custom data types

```
>>> class Bottles:
...     def __init__(self,number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(20)
>>> printf(b"%d bottles of water\n", bottles)
20 bottles of water
```

Specifying argument types and return types

argtypes – use this attribute to set the type of arguments of a function exported from DLL

```
>>> printf.argtypes = [c_char_p, c_char_p,
c_int, c_double]
```

```
>>> printf(b"String '%s', Int %d, Double %f\n",
b"Hi", 10, 2.2)
```

```
String 'Hi', Int 10, Double 2.200000
```

restype – assign a ctype to specify the type of return value from the foreign function

Pass by reference

Use byref() function

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s", byref(i),
byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049041 75 b'Hello'
>>>
```

Structures and Unions

- Base classes **Structure** and **Union**
- Create subclasses of the above base classes
- Define _fields_ attribute
- _fields_ is a list of 2 item tuples – field name and field type

Structures and Unions

```
>>> from ctypes import *
>>> class Student(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("age", c_double)]
...
>>> s1=Student(b"JP", c_double(20.0))
>>> s1.name
b'JP'
>>> s1.age
20.0
>>>
```

Bit fields in Structures and Unions

```
>>> from ctypes import *
>>> class GpioDw0(Structure):
...     _fields_ = [("PadMode", c_int, 3),
...                 ("Termination", c_int, 3),
...                 ("Reserved", c_int, 26)]
>>> print(GpioDw0.PadMode)
<Field type=c_long, ofs=0:0, bits=3>
>>>
```

Arrays

It’s a sequence containing a fixed number of instances of the same data type

Create array by multiplying a data type with a positive integer

```
>>> from ctypes import *
>>> class Student(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("class_of_student", c_int),
...                 ("marks", c_int * 6)]
...
>>> print(len(Student().marks))
6
>>>
```

Pointers

Create instances of Pointers using pointer() function on a ctypes type

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

Use **contents** attribute to get the object to which pointer is pointing to

```
>>> pi.contents
c_long(42)
>>>
```

Hello Module Using Ctypes

hello module

- Create a DLL for hello module with sayhello function
- Load hello DLL using ctypes
- Invoke sayhello() function on the dll object

```
HELLODLL_API void sayhello()
{
    printf("Hello from DLL");
    return;
}
```

Hands on writing hello module (Cont...)

```
#hello.py python module

from ctypes import *
import sys
import os.path

dll_path = os.path.join(os.path.dirname(__file__), "hello.dll")    # on Linux replace DLL with SO file name
handle = windll.LoadLibrary(dll_path) # use cdll on Linux instead of windll

handle.sayhello()
```

Passing structure pointer to DLL APIs

Create a sample DLL with an API getStudent

Prototype of getStudent in C:

void getStudent(struct Student **pStudent)

Structure type definition in C:

```
typedef struct {
        char *name;
        int age;
}Student;
```

Passing structure pointer to DLL APIs (Cont...)

C Function definition:

```
STUDENTDLL_API void getStudent(Student **pStudent)
{
        Student *s1 = malloc(sizeof(Student));
        s1->name = malloc(64);
        s1->age = 20;
        strcpy_s(s1->name, 3,  "JP");
        *pStudent = s1;
        return;
}
```

Error handling code has been purposefully omitted for simplicity

Passing structure pointer to DLL APIs (Cont…)

- For memory allocated inside the DLL, DLL should provide an interface to free that memory.
- For example, delStudent is a function to delete the memory allocated to Student structure within the DLL

```
STUDENTDLL_API void delStudent(Student **pStudent)
{
	if (pStudent && *pStudent)
	{
		free(*pStudent);
		*pStudent = NULL;
	}
}
```

Passing structure pointer to DLL APIs (Cont...)

```
#Invoking the getStudent API from Python

from ctypes import *
import sys
import os.path

dll_path = os.path.join(os.path.dirname(__file__), "student.dll")
handle = windll.LoadLibrary(dll_path)

# Python equivalent Student structure
class Student(Structure):
    _fields_ = [('Name', c_char_p), ('Age', c_int)]
```

```
handle.getStudent.argtype = [POINTER(POINTER(Student))]
pst = POINTER(Student)()
handle.getStudent(byref(pst))
pst[0].Age
pst[0].Name
```

Freeing memory allocated in DLL using a DLL provided API

- DLL should provide an API to delete/free the memory allocated to any object with it.
- For example, the sample DLL has provided an API called delStudent using this we can delete / free the memory allocated to student object
- **handle.delStudent(pst)**

Ctypes

- **Advantages**
  - Part of the Python standard library
  - No additional compiling is required
  - **Wrapping code is entirely in Python**
  - **Binary (DLL) independent of Python interpreter**
    - **Can be used on any python version**
    - **Can be used in other languages**
- **Disadvantages**
  - Only shared libraries (DLLs/SO) can be wrapped
  - No direct support for C++
  - Syntax is quite complex and is not very natural

Further Learning (Cont...)

- Python C APIs documentation
  - https://docs.python.org/3/c-api/intro.html
- Ctypes documentation
  - https://docs.python.org/3/library/ctypes.html

Conclusion

- Learnt about two different ways of interfacing native code with Python.
- **Use Python C APIs** approach if you want your **module to operate in environments with no support for dynamic loading of libraries for example (UEFI),** or you want to build a built-in python module
- **Use ctypes** approach if you want **to interface an existing native code DLL/Shared Object with Python** (can be used on OS environments such as Windows, Linux, Mac OSx and etc)

Thank You