# 11-Native-Code-Binding

January 7, 2026

## 1 Binary Interfacing

Motivations of binary interfacing:

1. Use of binary libraries within the script language
2. From a C, C++ code, use a python interpreter and use flexibility of script languages.

- AutoCAD vs List
- Gimp vs Scheme and Python
- Blender vs Python
- Excell vs Visual Basic
- … Script language provides a powerful interface to Human to interact with a binary program, debuging, automating, macros etc.

### 1.1 ctypes

It loads dynamic libraries built in C/C++ in python run-time and let all C/C++ types encapsulated in classes so that data can be interchanged. It calls C/C++ functions within python.

class wrappers around C values: * `c_int, c_double, c_float, c_char` * `c_int * n` for an integer array of size n. `int a[10]; -> (c_int * 10)` * `c_void_p` and `c_char_p` for `void *` and `char *` * use `POINTER(type)` for pointer to type * `byref(value)` gives a pointer to value `POINTER(typeof(value))`

For structures. Inherit `ctypes.Structure` for creating a special wrapper:

```
struct Complex {
    double x;
    double y;
}
```

```
class Complex(ctypes.Structure):
    _fields_ = [("x", c_double), ("y", c_double)]
```

You need to set `func.restype` to set result type of Functions since python does not have any idea about the function prototypes. Also you can use `argtypes = (type1, type2, ...)` to set arguments type

```
lib = ctypes.CDLL("binaryfile.so")
lib.func.restype = c_double

darray = c_double * 10
```

```
dval = darray( * [ 1, 2, 4, 5, 6, 7, 8, 9, 10, 3])  # * unwraps the list in set of arguments
lib.func( c_double(arg1), c_int(arg2), dval)
# or
lib.func.argtypes = c_double, c_int, darray
lib.func(arg1, arg2, dval )
```

[2]:
```
from ctypes import *

# use system math library
mlib = CDLL('libm.so.6')

#this will return an integer since result type unknown
print(mlib.sin(c_double(3.1415926536)))

# try again by setting return type
mlib.sin.restype = c_double
print(mlib.sin(c_double(3.1415926536)))
```

```
0
-1.0206823934513925e-11
```

Following is a comprehensive example containing different parameter passing mechanisms in *ctest.c* compile as > `gcc -shared -o libctest.so ctest.c`

to create `libctest.so`

[3]:
```
# ctest.c contains following functions in C
# compile as 'gcc -shared -o libctest.so ctest.c' to create libctest.so
# struct Complex add(struct Complex , struct Complex );
# void swap(struct Complex *, struct Complex *);
# double sum(double f[], int n);
# void titlecase(char *);
# int tokenize(char sep, char *str, char t[][20]);
# void mult(double a[][100], double b[][100], double c[][100], int n, int r ,␣
 ↪int n);

# get the python library. searched in system path, specify full or
# relative path if not in system library
lib = CDLL('./libctest.so')

class Complex(Structure):
    _fields_ = [("x", c_double), ("y", c_double)]

# struct Complex add(struct Complex , struct Complex );
lib.add.restype = Complex
r = lib.add(Complex(3.1, 4.2), Complex(1.9, 2.8))
print('1-add\n',r, r.x, r.y)
```

```
a = Complex(3, 4)
b = Complex(2, 7)

# void swap(struct Complex *, struct Complex *);
# send pointers to values to pass by pointer
lib.swap(byref(a), byref(b))
print('2-swap\n',a.x, a.y, b.x, b.y)

# double sum(double f[], int n);
# Array type. You can pass pointers as in C. BE CAREFULL ABOUT STORAGE
# YOU CAN GET A SEGFAULT in PYTHON!!!
lib.sum.restype = c_double
lib.sum.argtypes = POINTER(c_double), c_int
myarr = (c_double * 10)( * [1, 2, 1, 6, 4, 2, 7, 3, 1, 5])
print('3-sum\n',lib.sum(myarr, 10))

# void titlecase(char *);
# c_char_p can be use to 0 terminated C strings. Make sure the storage is␣
 ↪sufficient
# use value field to get the content
lib.titlecase.argtypes = (c_char_p,)
name = c_char_p(b'the advantages of script languages over OTHER languages')
lib.titlecase(name)
print('4-titlecase\n', name.value)

# int tokenize(char sep, char *str, char t[][20]);
# this is slightly tricky t needs to store resulting tokens, so need sufficient␣
 ↪storage
lib.tokenize.restype = c_int
tokenstype = (c_char * 20) * 20
lib.tokenize.argtypes = c_char, c_char_p, tokenstype

# create an array for the tokenize result array of 20 strings
res = tokenstype ( )
for i in range(20):
    res[i] = create_string_buffer(20)
n = lib.tokenize(b' ', b'the advantages of script languages over OTHER␣
 ↪languages', res )
tokens = [tok.value for tok in res[:n]]
print('5-tokenize\n',n,tokens)

# void mult(double a[][100], double b[][100], double c[][100], int n, int r ,␣
 ↪int n);
a = [[ 1, 2, 3, 4, 5], [11, 12, 13, 14, 15]]
b = [[1, 2], [2, 3],[4, 5], [6, 7],[8,9]]
arrtype = (c_double * 100) * 100
```

```
# initialize matrix a
matA = arrtype()
i = 0
for row in a:
    matA[i] = (c_double * 100)( * row)
    i += 1

# initialize matrix B
matB = arrtype()
i = 0
for row in b:
    matB[i] = (c_double * 100)( * row)
    i += 1
matC = arrtype()

# multiply
lib.mult.argtypes = arrtype, arrtype, arrtype, c_int, c_int, c_int
lib.mult(matA, matB, matC, 2, 5, 2)

# result
print('6-mult\n',[[matC[i][j] for i in range(2)] for j in range(2)])
```

```
1-add
 <__main__.Complex object at 0x7fa9342e7950> 5.0 7.0
2-swap
 2.0 7.0 3.0 4.0
3-sum
 32.0
4-titlecase
 b'The Advantages Of Script Languages Over OTHER Languages'
5-tokenize
 8 [b'the', b'advantages', b'of', b'script', b'languages', b'over', b'OTHER',
b'languages']
6-mult
 [[81.0, 291.0], [96.0, 356.0]]
```

## 1.2 SWIG

https://github.com/swig/swig is a tool for generation of bindings and interfaces of C/C++ files in various platforms.

It compiles interface description files and generates wrapper code that glues script language with C/C++ libraries.

For example, a C++ class library:

```
/* File : example.h */

class Shape {
public:
```

```
  Shape() {
    nshapes++;
  }
  virtual ~Shape() {
    nshapes--;
  }
  double  x, y;
  void    move(double dx, double dy);
  virtual double area() = 0;
  virtual double perimeter() = 0;
  static  int nshapes;
};

class Circle : public Shape {
private:
  double radius;
public:
  Circle(double r) : radius(r) { }
  virtual double area();
  virtual double perimeter();
};

class Square : public Shape {
private:
  double width;
public:
  Square(double w) : width(w) { }
  virtual double area();
  virtual double perimeter();
};
```

and interface description:

```
/* File : example.i */
%module example

%{
#include "example.h"
%}

/* Let's just grab the original header file here */
%include "example.h"
```

SWIG compiler generates the language bindings for a given language when called as:

```
swig -c++ -python example.i
```

Creates `example_wrap.cxx` file which creates necessary bindings for the class library. When wrapper and the original code is compiled into a shared libray.

```
g++ example_wrap.cxx example.cpp -o _example.so -I/usr/include/python3
```

Target shared library can be used in python as a module:

```
import example
c = example.Circle(10)
c.move(4,10)
print(c.area())
```

It create type wrappers for default types in the interface. If further data types are needed like STL data types. The interface description language can be used to load libraries or create data type mappings between language and C/C++.

```
...
%include "example.h"
%template(ShapeVector) std::vector<Shape>;
```

For example if one of the methods return a `vector` of `Shape` values. The last line includes a SWIG template library to make `ShapeVector` class available in Python or the target language.

See:

https://github.com/onursehitoglu/python-445/tree/master/examples/binding/swigpy

for binding example of a tree library

Almost same library and interface description can be used to generate a Javascript/Node library:

https://github.com/onursehitoglu/python-445/tree/master/examples/binding/swigjs

### 1.3 WASM

Another way of mixing the languages is to share the same target architecture. WASM (WebAssembly) is a special case for Javascript. It is a virtual architecture that compilers can generate target binaries. Most of the modern browsers and interpreters execute WASM code with the minimum overhead.

This way, many language platforms and compilers can produce cross-platform binaries that can be used even in a browser. Javascript scripts and any other language programs can co-exist in the same application.

The following technologies are used with WASM and make it possible:

- **emscripten** A compiler framework for generating WASM binaries
- **embind** Binding framework for emscripten that makes Javascript/language bindings. Makes C/C++ call Javascript and vice versa.
- **WASI** Webassembly System Interface to generate system library interface for WASM compiled code. Like POSIX for WASM.

The same example for SWIG is compiled with emscripten and embedded in a browser in the example:

https://github.com/onursehitoglu/python-445/tree/master/examples/binding/emscripten

[ ]: