Computer Architecture

Programming in C

Extending Python using C

Agenda

- The Python C API
  - define a function
  - parse the args
  - return values to Python
  - create a method table
- Compile the C code
- Import and use the module
- Appendix

Python C API

Python C API is a set of functions, macros, and structures that allow you to extend Python by writing C code.

- You can define your own Python functions, manipulate Python objects, and return values in C using this API.

Why we want to extend Python with C?

- Speed up bottlenecks in Python code, particularly in loops and numerical computations
- Creating Python modules
- Using an existing C or C++ library with Python
- ……

Including Python.h

- To use the C API, firstly, you should include the <Python.h> in your script.

```
2   #include <Python.h>
3   //if the Python.h is not found in the default system path in you IDE,
4   //you can inlcude it by specify the path manually, like the following example:
5   //(replace the path with that in your laptop)
6   #include </Library/Frameworks/Python.framework/Versions/3.12/Headers/Python.h>
```

Defining a Python function in C

To define a Python function in C, it should follow a specific signature that the Python interpreter can recognize:

```
static PyObject* function_name(PyObject* self, PyObject* args);
```

- `self`: for module-level functions, this argument is usually ignored;
- `args`: this is a tuple containing the arguments passed to the function from Python.

```c
// Function to add two numbers
static PyObject* add(PyObject* self, PyObject* args) {
    int a, b;
    // Parse the input from Python arguments (two integers)
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    int result = a+b;
    // Return the sum of the two numbers
    return Py_BuildValue("i", result);
}
```

In the example, it defines a function `add`, which sums the input numbers.

In this function, the input numbers are expected to be two integers, obtained from the args by `PyArg_ParseTuple`.

Its return value is converted into a `PyObject` by `Py_BuildValue`.

Parsing Arguments: PyArg_ParseTuple

`PyArg_ParseTuple` is the main function used to extract arguments from the `args` tuple. It takes a format string that specifies the types of arguments expected, followed by pointers to store the extracted values.

```
6     int a, b;
7     // Parse the input from Python arguments (two integers)
8     if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
9         return NULL; //return NULL to signal an error
10    }
```

In the example, it parses the `args` into two integers `a` and `b`.

If any error occurs in parsing, `PyArg_ParseTuple` returns 0, and the function will return `NULL`.

Format specifiers:

- `"i"`: `int`
- `"f"`: `float`
- `"s"`: string `(char *)`
- `"O"`: Python object

Returning values to Python

The return value of a function must always be a `PyObject*` because it will be handled by the Python interpreter as a Python data type (e.g., integer, list, dictionary, etc.).

- To convert a C type value to Python object, we use `Py_BuildValue`. It's often used for returning simple C data types to Python objects like integers, floats, tuples, and lists.

```
38  return Py_BuildValue("i", result) //returning a C integer type data
39
40  return Py_BuildValue("ii", a, b) //returning a tuple of two integers
41
42  return Py_BuildValue("s", "Hello from C!") //returning a string
```

Note:
You can define `PyObject*` in the function and return it directly to Python for complex C data types.

Format specifiers:

- "i": int
- "l": long
- "f": float
- "d": double
- "s": string (char *)
- "O": Python object
- "()": tuple
- "[]": list

Returning complex data types

We can create a Python list manually using `PyList_New` and `PyList_SetItem`.

- `PyList_New(array_size)`: creates an new empty Python list.
- `PyLong_Fromlong(c_array[i])`: converts the int from the C array to Python integer object.
- `PyList_SetItem(py_list, i, num)`: set the i-th element of the Python list to the Python integer `num`. (Note: `PyList_SetItem` steals the reference to `num`, meaning you don't need to `Py_DECREF num` after setting it in the list.)

Note: To return a C array to Python as a Python list, we **cannot** directly use `Py_BuildValue` for an array.

```c
// Function to return a C array as a Python list
static PyObject* array_to_list(PyObject* self, PyObject* args) {
    // Example array of integers
    int c_array[] = {1, 2, 3, 4, 5};
    int array_size = 5;
    // Create a new Python list of the same size as the C array
    PyObject* py_list = PyList_New(array_size);
    if (!py_list) {
        return NULL;  // Return NULL on failure
    }
    // Fill the Python list with elements from the C array
    for (int i = 0; i < array_size; i++) {
        // Convert C int to Python int
        PyObject* num = PyLong_FromLong(c_array[i]);
        if (!num) {
            // Clean up the list if memory allocation fails
            Py_DECREF(py_list);
            return NULL;
        }
        // Set the Python list item
        //(PyList_SetItem steals the reference to `num`)
        PyList_SetItem(py_list, i, num);
    }
    // Return the Python list
    return py_list;
}
```

`Py_DECREF` is like the `free`, which deallocates the `PyObject`.

Return a 2D array (a list of list)

```c
// Function to return a 2D C array as a Python list of lists
static PyObject* array_2d_to_list(PyObject* self, PyObject* args) {
    int rows = 2;
    int cols = 3;
    int c_array[2][3] = {{1, 2, 3}, {4, 5, 6}};
    // Create a Python list to hold the rows
    PyObject* py_list = PyList_New(rows);
    if (!py_list) {
        return NULL;
    }
    // Loop over rows
    for (int i = 0; i < rows; i++) {
        // Create a Python list for each row
        PyObject* row_list = PyList_New(cols);
        if (!row_list) {
            Py_DECREF(py_list);
            return NULL;
        }
        // Loop over columns and fill the row
        for (int j = 0; j < cols; j++) {
            PyObject* num = PyLong_FromLong(c_array[i][j]);  // Convert C int to Python int
            if (!num) {
                Py_DECREF(row_list);
                Py_DECREF(py_list);
                return NULL;
            }
            PyList_SetItem(row_list, j, num);  // Set the item in the row list
        }
        // Set the row list in the outer list
        PyList_SetItem(py_list, i, row_list);  // Steals reference to row_list
    }
    return py_list;  // Return the list of lists
}
```

To return a 2D array in C, we should create a list of list.

Each inner list represents a row of the array.

Note:

- The C API also provides functions for returning C structures as PyObjects, and returning objects as complex Python object like Python dictionary. You may refer to the Python C API manual: https://docs.python.org/3/c-api/index.html (also, tools like ChatGPT can generate comprehensive introductions of them)

Register your functions with Python

We should create a method table to register the functions we defined.

```
52  // Method definitions
53  static PyMethodDef MyMethods[] = {
54      {"add", add, METH_VARARGS, "Add two numbers"},
55      {"ones", ones, METH_VARARGS, "Create a matrix with specificed rows and columns"},
56      {NULL, NULL, 0, NULL}  // Sentinel to mark the end of the table
57  };
```

- `PyMethodDef`: the table that maps function names in module and the actual functions
- `METH_VARAGS`: indicates the function accepts a variable number of arguments, passed in a tuple.
- `{NULL, NULL, 0, NULL}` : indicates the end of the table.

Module definition and initialization

After defining the functions and method table, we need to define the module and provide an initialization function.

```
60  // Module definition
61  static struct PyModuleDef mymodule = {
62      PyModuleDef_HEAD_INIT,
63      "mymodule",  // Module name
64      NULL,        // Optional documentation
65      -1,          // The module keeps state in global variables
66      MyMethods
67  };
68
69  // Module initialization function
70  PyMODINIT_FUNC PyInit_mymodule(void) {
71      return PyModule_Create(&mymodule);
72  }
```

These operations are the routines.

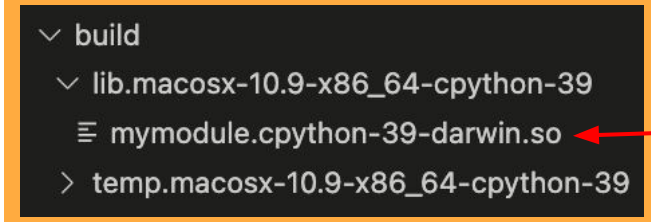
We use the structure and functions in the API to complete the definition and initialization of the module.

Compiling and Extension

You can use Python to compile the module.

1. Create a setup.py in the folder you save your C files.
2. Then, run the setup.py in the terminal to build the module:
   `Python setup.py build`

This will create a folder build. The file ends with `.so` (or `.pyd`) is the module built.



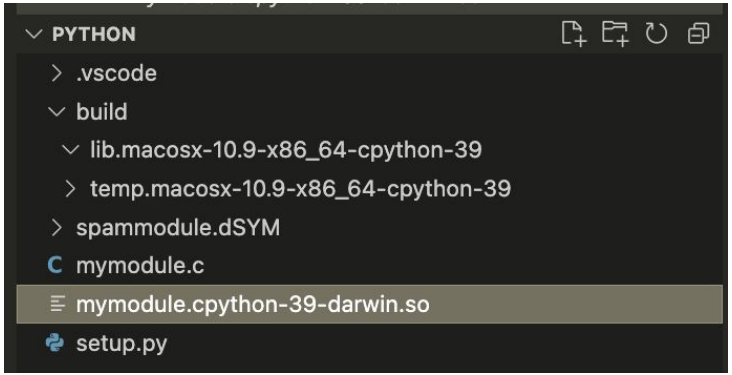

```python
from setuptools import setup, Extension

module = Extension("mymodule", sources=["mymodule.c"])

setup(
    name="mymodule",
    version="1.0",
    description="A simple C extension for Python",
    ext_modules=[module],
)
```

You may need to install the setuptools: `pip3 install setuptools`

Import and use the module

Once compiled, there are two ways to use your module:

- (recommend) You can copy and paste the `.so` file (e.g., `mymodule.cpython-39-dawin.so`) to the your working folder, and then, you can import `mymodule` to your script.
- You can run `python setup.py install` to install the module to your Python environment.




```
(anaconda3) bing@Xianbins-MacBook-Pro Python % python
Python 3.9.13 (main, Aug 25 2022, 18:29:29)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import mymodule
>>> mymodule.add(1, 2)
3
>>>
```

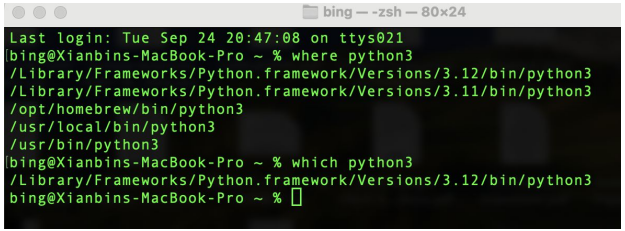
Exercise:

Compile the `mymodule.c` shared in Brightspace by the `setup.py`.

- Check if the path of `Python.h`, modify it if needed
  - For Unix-like systems, you can use the terminal commands like

    `where python3`

    `Which python 3`

    to find the location of your Python installation.

- Once the compilation is done, import the module in a Python script, make sure it works properly.

Appendix

Cython: https://cython.org/

Cython is a programming language that makes it easier to write C extensions for Python. It allows you to write Python-like code that gets compiled to highly efficient C or C++ code, which can significantly speed up the performance of your Python programs.