Astroinformatics II (Semester 2 2024)

Integrating C++ and Python

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

December 3, 2024

Motivation

We have seen that both Python and C/C++ come with their own **benefits and drawbacks**.

Idea: Using both of them together for their appropriate tasks.

Motivation

We have seen that both Python and C/C++ come with their own **benefits and drawbacks**.

Idea: Using both of them together for their appropriate tasks.

? How to do that?

Motivation

In a larger software project, the usage of Shell scripts (`bash`), Python and C/C++ have their distinctive roles:

`bash`

- preparing folders
- preparing files
- starting our code

Python

- carrying out data I/O
- using libraries for complex algorithms (scientific computing, machine learning)
- plotting

C/C++

- computationally expensive but algorithmically simple(r) tasks

Using bash

Similarly to executing a Python script from the terminal, you can use a `bash` script to call and execute your Python scripts.
For **example**:

```
#!/bin/bash

python3 myscript.py
```

Using bash

Similarly to executing a Python script from the terminal, you can use a `bash` script to call and execute your Python scripts.
For **example**:

```
#!/bin/bash

python3 myscript.py
```

This can be, of course, part of a more complex script.

Using bash

You can pass command-line arguments to your Python script from `bash`.

In Python, you can then access these arguments using the `sys.argv` list from the `sys` module. For `example`:

```
#!/usr/bin/env python3
import sys

if len(sys.argv) > 1:
    print("Hello, " + sys.argv[1] + "!")
else:
    print("Hello, World!")
```

When calling this script from `bash`, you can provide a name as an argument:

```
./myscript.py Mars
```

will print

```
Hello, Mars!
```

Using `bash`

With the above, we saw how to start a Python program from `bash`.

In addition to this, it is sometimes useful to use `bash` for certain operations it can carry out fast, but which would be slow in Python.

`bash` can be fast for e.g.:

- searching (and replacing) in large files
- working with tables: removing columns, changing order of columns

Using bash

With the above, we saw how to start a Python program from `bash`.

In addition to this, it is sometimes useful to use `bash` for certain operations it can carry out fast, but which would be slow in Python.

`bash` can be fast for e.g.:

- searching (and replacing) in large files
- working with tables: removing columns, changing order of columns

Carrying out such tasks with `bash` is much faster than using Python as with `bash` there is only minimal overhead.

Using bash

In the following more complete **example**, we create a folder, cd into that folder and download a file using curl.

We then remove columns 2 and 4 from it, creating a new file lc_short.cl.

We then call Python code that will process this file.

```
#!/bin/bash

mkdir /lab/username/spaceproject/lightcurves
cd /lab/username/spaceproject/lightcurves

curl https://archive.observatory.edu/missions/space/lightcurvetable.lc

cut -d\  -f2,4 --complement lightcurvetable.lc > lc_short.lc

python3 process_lc('lc_short.lc')
```

Program Architecture

Conclusion on **program architecture**:

- Use `bash` for simple, quick automation tasks directly involving the OS.
- Use Python if you need more complexity, readability, or if you're already using Python for other tasks.
- Use C primarily for performance-critical applications, not for simple scripting tasks.

Program Architecture

Conclusion on **program architecture**:

- Use `bash` for simple, quick automation tasks directly involving the OS.
- Use Python if you need more complexity, readability, or if you're already using Python for other tasks.
- Use C primarily for performance-critical applications, not for simple scripting tasks.

What we can archieve with this is a **sequence of tasks**.

Program Architecture

Conclusion on **program architecture**:

- Use `bash` for simple, quick automation tasks directly involving the OS.
- Use Python if you need more complexity, readability, or if you're already using Python for other tasks.
- Use C primarily for performance-critical applications, not for simple scripting tasks.

What we can archieve with this is a **sequence of tasks**.

But sometimes we want code written in Python and C++ **interact** in the same application.

Calling External Program from Python

We can call code written in C from within a Python program.
In the simplest form, we call a complete external program, and retrieve the output/return code with the `subprocess` package.

Calling External Program from Python

We can call code written in C from within a Python program.
In the simplest form, we call a complete external program, and retrieve the output/return code with the `subprocess` package.

The basic **syntax** is:

```
import subprocess
subprocess.call("command-name-here")
subprocess.call(["/path/to/command", "arg1", "-arg2"])
```

Calling External Program from Python

We can call code written in C from within a Python program.
In the simplest form, we call a complete external program, and retrieve the output/return code with the `subprocess` package.

The basic **syntax** is:

```
import subprocess
subprocess.call("command-name-here")
subprocess.call(["/path/to/command", "arg1", "-arg2"])
```

example:
Run the `ping` command to send `ICMP ECHO_REQUEST` packets to `www.nasa.gov`:

```
#!/usr/bin/python
import subprocess
subprocess.call(["ping", "-c 2", "www.nasa.gov"])
```

Calling Shared Library

The default implementation of Python is called CPython and is, as the name suggests, written in C.

For this reason, C functions can be used rather easily in a Python program. We will see here how we can call C functions from Python code using a **shared library** with a `ctypes` wrapper.

Calling Shared Library

The default implementation of Python is called CPython and is, as the name suggests, written in C.

For this reason, C functions can be used rather easily in a Python program. We will see here how we can call C functions from Python code using a **shared library** with a `ctypes` wrapper.

A shared library or a shared object can be called from other programs. It has the file endin `.so` for Linux and OS X, or `.dll` (which stands for dynamically loaded library) for Windows.

Calling Shared Library

The default implementation of Python is called CPython and is, as the name suggests, written in C.

For this reason, C functions can be used rather easily in a Python program. We will see here how we can call C functions from Python code using a **shared library** with a `ctypes` wrapper.

A shared library or a shared object can be called from other programs. It has the file endin `.so` for Linux and OS X, or `.dll` (which stands for dynamically loaded library) for Windows.

The Python `ctypes` library is a robust resource that empowers us to generate C-compatible data types and directly invoke functions in dynamic link libraries or shared libraries using Python.

Calling Shared Library

Calling a shared libary, written in C, from Python code using a `ctypes` wrapper involves the following steps:

- Creating a C file (`.c` extension) with the required functions.
- Creating a shared library file (`.so` extension) using the C compiler.
- In the Python program, import the `ctypes` module. Create a `ctypes.CDLL` instance from the shared file.
- Finally, call the C function using the format `{CDLL_instance}.{function_name}({function_parameters})`.

Calling Shared Library

Step 1: Creating a C File with the required function(s)

```
#include <stdio.h>

int my_function(int i, int j) {
        return i * i + j;
}
```

We have a simple C function that will return the square of an integer. We save this function code in the file named `my_functions.c`.

Calling Shared Library

Step 2: Creating the Shared Library File

We can use the following command to create the shared library file from the C source file.

```
$ gcc -fPIC -shared -o my_functions.so my_functions.c
```

Calling Shared Library

Step 3: Calling C Function from Python Program

To call the function from within a Python program, we have to import the `ctypes` library.
Then, with `ctypes.CDLL()`, we load the shared library file into memory, making the functions within it accessible.
We then declare the function prototype, including the arguments.
Finally, we can call the C function just like any other Python function.

```python
import ctypes

lib = "./my_functions.so"

# Declare the function prototype
my_function = lib.my_function
my_function.argtypes = [ctypes.c_int, ctypes.c_int]
my_function.restype = ctypes.c_int

# Call the C function from Python
result = my_function(5, 10)
print(result) # returns 35
```

Calling Shared Library

Don't forget:

If you **change** the code in the C program file, you will have to recompile to regenerate the shared library file.

Using the Cython Language

Another option for using C code inside of a Python program is using **Cython**.

Cython extends Python's syntax, enabling the creation of C extensions in a Python-like manner. It serves as a bridge between Python and C, allowing developers to combine Python's simplicity with C's performance.

With Cython, you can leverage Python's high-level features while seamlessly incorporating low-level C functionality into your code.

Calling Shared Library

Step 1: Create a `.pyx` file

We create a file with extenson `.pyx`, e.g. `mymodule.pyx`, with the following content:

```
cdef extern from "./myheader.h":
    int my_function(int a, int b)

def call_c_function(int a, int b):
    return my_function(a, b)
```

Calling Shared Library

Step 2: Create a `setup.py` file

We create a file `setup.py` with the following content that refers to the previously created `.pyx` file:

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

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

Calling Shared Library

Step 3: Compile the Cython module

To compile the Cython module, navigate to the directory containing the setup.py file in the terminal and execute the following command:

```
python setup.py build_ext --inplace
```

This command will generate the necessary C files and compile them into a shared library.

Calling Shared Library

Step 3: Compile the Cython module

To compile the Cython module, navigate to the directory containing the setup.py file in the terminal and execute the following command:

```
python setup.py build_ext --inplace
```

This command will generate the necessary C files and compile them into a shared library.

Step 4: Import the module and call the C function

```python
import mymodule

result = mymodule.call_c_function(5, 10)
print(result)
```

Calling Shared Library

To summarize this:

The ability to call C functions in Python provides a robust way to integrate high-performance capabilities and direct system access into Python applications.

Through the utilization of libraries like ctypes and Cython, developers can effortlessly connect Python and C, effectively closing the gap between the two languages.

This approach is especially valuable for optimizing computationally demanding tasks, interact with hardware devices, or using existing C libraries.

Calling Shared Library

To summarize this:

The ability to call C functions in Python provides a robust way to integrate high-performance capabilities and direct system access into Python applications.

Through the utilization of libraries like ctypes and Cython, developers can effortlessly connect Python and C, effectively closing the gap between the two languages.

This approach is especially valuable for optimizing computationally demanding tasks, interact with hardware devices, or using existing C libraries.

more:
`https://docs.python.org/2/extending/`

Numba

Numba is an open source JIT (just in time) compiler that translates a subset of Python and NumPy code into fast machine code.

Numba translates Python functions to optimized machine code at runtime using the industry-standard LLVM compiler library. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

You don't need to replace the Python interpreter, run a separate compilation step, or even have a C/C++ compiler installed. Just apply one of the Numba decorators to your Python function, and Numba does the rest.

We will just see a short introduction here. More on Numba can be found at `https://numba.pydata.org/`

Numba

Numba is designed to be used with NumPy arrays and functions. Numba generates specialized code for different array data types and layouts to optimize performance. Special decorators can create universal functions that broadcast over NumPy arrays just like NumPy functions do.

Numba

Numba is designed to be used with NumPy arrays and functions. Numba generates specialized code for different array data types and layouts to optimize performance. Special decorators can create universal functions that broadcast over NumPy arrays just like NumPy functions do.

In addition to writing traditional Python scripts, Numba also works great with Jupyter notebooks for interactive computing, and with distributed execution frameworks, like Dask and Spark.

Numba

Numba is designed to be used with NumPy arrays and functions. Numba generates specialized code for different array data types and layouts to optimize performance. Special decorators can create universal functions that broadcast over NumPy arrays just like NumPy functions do.

In addition to writing traditional Python scripts, Numba also works great with Jupyter notebooks for interactive computing, and with distributed execution frameworks, like Dask and Spark.

Numba offers a range of options for **parallelizing** your code for CPUs and GPUs, often with only minor code changes:
Threading, Vectorization, GPU Acceleration.

Numba

Using the `numba.jit()` **decorator**, you can mark a function for optimization by Numba's JIT compiler.

The basic usage in so-called **lazy compliation**, were we let Numba decide when and how to optimize, is the following:

Numba

```
from numba import jit

@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

In **lazy compilation**, compilation will be deferred until the first function execution. Numba will infer the argument types at call time, and generate optimized code based on this information.

Numba will also be able to compile separate specializations depending on the input types. For example, calling the `f()` function above with integer or complex numbers will generate different code paths:

```
>>>f(1, 2)
3

>>>f(1j, 2)
(2+1j)
```

Numba

We can also tell Numba the function signature (including argument types) you are expecting. This is called **eager compilation**.
The function `f()` now looks like:

```
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

Numba

We can also tell Numba the function signature (including argument types) you are expecting. This is called **eager compilation**.
The function `f()` now looks like:

```
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

`int32(int32, int32)` is the function's signature. In this case, the corresponding specialization will be compiled by the `@jit` decorator, and no other specialization will be allowed. This is useful if you want fine-grained control over types chosen by the compiler (for example, to use single-precision floats).

Numba

We can also tell Numba the function signature (including argument types) you are expecting. This is called **eager compilation**.
The function `f()` now looks like:

```
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

`int32(int32, int32)` is the function's signature. In this case, the corresponding specialization will be compiled by the `@jit` decorator, and no other specialization will be allowed. This is useful if you want fine-grained control over types chosen by the compiler (for example, to use single-precision floats).

The compiled function gives the expected result:

```
>>>f(1, 2)
3
```

Calling Python Code from C

We have seen various ways on how to call C code from within Python code.

In some cases, we want to do the opposite.

Calling Python Code from C

We have seen various ways on how to call C code from within Python code.

In some cases, we want to do the opposite.

Use case: We have a program written in C/C++ for which we want to provide a scripting language to extend the program.

Developing a scripting language for this is a nontrivial task that easily can become demanding. A better solution is to embed the interpreter of an existing interpreted language, like Python.

With Python, you can **embed the interpreter directly into your application** and expose the full power and flexibility of Python without adding very much code at all to your application.

Including the Python Interpreter into a C Application

Including the Python interpreter in your program is simple. Python provides a single **C header file** for including all of the definitions you need when embedding the interpreter into your application: `Python.h`.

`Python.h` includes already several of the standard headers.

Including the Python Interpreter into a C Application

Including the Python interpreter in your program is simple. Python provides a single **C header file** for including all of the definitions you need when embedding the interpreter into your application: `Python.h`.

`Python.h` includes already several of the standard headers.

For **linking** your application to the Python interpreter at compile time, you run the `python-config` program to get a list of the linking options that should be passed to the compiler:

```
-lpython2.3 -lm -L/usr/lib/python2.3/config
```

Including the Python Interpreter into a C Application

How much code does it take in the C program to onclude the Python interpreter?

It can be done in as little as three lines of code, which initialize the interpreter, send it a string of Python code to execute and then shut the interpreter back down.

example: Embedding Python into C code

```
include Python.h

void exec_pycode(const char* code)
{
  Py_Initialize();
  PyRun_SimpleString(code);
  Py_Finalize();
}
```

Including the Python Interpreter into a C Application

Alternatively, you can embed an interactive Python terminal in your program by calling `Py_Main()`.

This will bring up the interpreter just as if you'd run Python directly from the command line. Control is returned to your application after the user exits from the interpreter shell.

example: Embedding an interactive Python interpreter

```
void exec_interactive_interpreter(int arg, char** argv)
{
  Py_Initialize();
  Py_Main(argc, argv);
  Py_Finalize();
}
```

Including the Python Environment into a C Application

So far, we have included a Python interpreter.
We can, however, do much more complex things like including the Python environment that we are using.
Fist let's take a look at initializing the environment that Python executes within.

When you run the Python interpreter, the main environment context is stored in the `__main__` module's namespace dictionary. All functions, classes and variables that are defined globally can be found in this dictionary.

Calling the `PyImport_AddModule()` function looks up the module name you supply and returns a `PyObject` pointer to that object. All Python data types derive from `PyObject`, which makes it a handy lowest-common denominator. Therefore, almost all of the functions that you'll deal with when interacting with the Python interpreter will take or return pointers to `PyObjects` rather than another more specific Python data type.
Once you have the `__main__` module referenced by a `PyObject`, you can use the `PyModule_GetDict()` function to get a reference to the main module's dictionary, which again is returned as a `PyObject` pointer. You can then pass the dictionary reference when you execute other Python commands.

Including the Python Environment into a C Application

The following C code **example** shows how to duplicate the Python global environment and execute two different Python files in separate environments.

```c
// Get a reference to the main module.
PyObject* main_module = PyImport_AddModule("__main__");

// Get the main module's dictionary and make a copy of it.
PyObject* main_dict = PyModule_GetDict(main_module);
PyObject* main_dict_copy = PyDict_Copy(main_dict);

// Execute two different files of Python code in separate environments
FILE* file_1 = fopen("file1.py", "r");
PyRun_File(file_1, "file1.py", Py_file_input, main_dict, main_dict);

FILE* file_2 = fopen("file2.py", "r");
PyRun_File(file_2, "file2.py", Py_file_input, main_dict_copy, main_dict_copy);
```

Including the Python Environment into a C Application

Now let's take a look at a slightly more useful **example** to see how Python can be embedded into a real program.

In this **example** we build a simple calculator:

Including the Python Environment into a C Application


```c
#include <python2.3/Python.h>

void process_expression(char* filename, int num, char** exp)
{
    FILE* exp_file;

    // Initialize a global variable for display of expression results
    PyRun_SimpleString("x = 0");

    // Open and execute the file of functions to be made available to user expressions
    exp_file = fopen(filename, "r");
    PyRun_SimpleFile(exp_file, exp);

    // Iterate through the expressions and execute them
    while(num--) {
        PyRun_SimpleString(*exp++);
        PyRun_SimpleString("print x");
    }
}

int main(int argc, char** argv)
{
    Py_Initialize();
    if(argc != 3) {
        printf("Usage: %s FILENAME EXPRESSION+\n");
        return 1;
    }
    process_expression(argv[1], argc - 1, argv + 2);
    return 0;
}
```

Summary: Astroinformatics II

During this course Astroinformatics II, we have seen:

- advanced usage of Python
- an introduction in C/C++ and object oriented programming
- how to combine `bash`, Python and C/C++

Usually, a software project in astronomy consists of various tasks suited best to specific programming languages and concepts.

At this knowledge stage, it is especially important to **identify the best tools** for individual tasks in developing scientific sofware.