Linux i Python w Elektronicznej Sieci – ćwiczenia #11: Łączenie Pythona i C++

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2023-06-25

W ramach kursu omawialiśmy dość szeroko programowanie w Pythonie oraz poznaliśmy podstawy C++. Języki te mocno różnią się od siebie – zarówno składnią jak i sposobem działania oraz przeznaczeniem. Mogą jednak stanowić doskonałe uzupełnienie - łączące łatwość modyfikacji pewnych fragmentów programu zapewnioną przez Pythona i wydajność zapewnianą przez C++ (czy nawet samo C).

Poniżej znajdują się przykładowe kody demonstrujące użycie Pythona z poziomu programu w C++ oraz użycie kodu C++ z poziomu Pythona. Korzystają one z biblioteki boost-python (i pośrednio libpython). W związku z tym konieczne może być doinstalowanie tych bibliotek:

```
apt install libboost-python1.67-dev
```

1 Pythonowe API programu w C++

Zaczniemy od jednej z metod wystawienia dane oraz funkcje z programu C++ do Pythona. Czyli umożliwieniu Pythonowi odczyt i modyfikację zmiennych utworzonych (i używanych) w kodzie C++ oraz wywoływania przez interpreter Pythona funkcji stworzonych w C++.

Przykładowy kod C++[1]:

```
/*  PLIK: py_api.cpp      kompilacja:
        g++ --std=c++11 -shared -fPIC -I/usr/include/python3.7m/ py_api.cpp \
                -o MyPyAPI.so -lpython3.7m -lboost_python37
        uwaga: - kolejność argumentów może być istotna
               - plik wynikowy powinien mieć taką samą nazwę jak moduł
                 zadeklarowany przy pomocy BOOST_PYTHON_MODULE()
*/

#include <iostream>
#include <string>
#include <list>

std::string f1(int a) {
        for (int i=0; i<a; ++i)
                std::cout << "Ala ma kota\n";
        return "Kot ma Alę";
}

struct K1 {
```

1. Wszystkie kody zawarte w tym skrypcie znajdują się także na http://vip.opcode.eu.org/#API_skrypty_..., skąd kopiowanie może być łatwiejsze niż z pdf'a.

```
                int a;
                static K1* obj;
                int f1(int b) { return a + b; }
                static K1* get() { return obj; }
};

void f2(K1& k, int n) {
        std::cout << "run f2(): " << k.f1(2*n) << "\n";
        k.a += 1;
}

K1* K1::obj = NULL;

#include <boost/python.hpp>

BOOST_PYTHON_MODULE(MyPyAPI) {
        boost::python::def("f1", f1);

        boost::python::class_<K1>("Klasa")
                .def_readwrite("a", &K1::a)
                .def("f1", &K1::f1)
                // w pythonie w odróżnieniu od C++ referencja do klasy jest jawnym
                // argumentem metod nie statycznych zatem f2 które przyjmuje jako
                // pierwszy argument referencję do obiektu klasy K1 możemy użyć
                // jako metody klasy pythonowej
                .def("f2", f2)

                // typo z funkcji do pythona zwracana jest wartość zmiennej (a nie
                // referencja do zmiennej C++) jednak dla funkcji zwracających
                // wskaźnik lub referencję na ogół chcemy aby zwracana była właśnie
                // referencja do istniejącej zmiennej C++
                .def("get", &K1::get,
                        boost::python::return_value_policy<
                                boost::python::reference_existing_object
                        >()
                )
        ;

        // f2 możemy też użyć jako niezależnej funkcji
        boost::python::def("f22", f2);

        // podobnie jak f1
        boost::python::def("f11", &K1::f1);
}
```

oraz kod Pythona wykorzystujący nasz moduł stworzony w C++:

```
# skrypt wykorzystujący moduł
# MyPyAPI stworzony w C++

# dodajemy bierzący katalog do ścieżki w której python szuka bibliotek
import sys
sys.path.append('./')
```

```python
# importujemy naszą bibliotekę (z pliku MyPyAPI.so)
import MyPyAPI

# uruchomienie funkcji f1 i odebranie wyniku
ret = MyPyAPI.f1(2)
print(ret)

# utworzenie obiektu klasy K1 (Klasa)
kk = MyPyAPI.Klasa()

# oraz jego używanie ...
kk.a = 3
print(kk.f1(2))

kk.f2(1)
MyPyAPI.f22(kk, 1)
print("kk.a =", kk.a)

print(MyPyAPI.f11(kk, 1))
```

Zadanie 1.0.1

Stwórz w C++ moduł Pythonowy (bibliotekę) zawierającą funkcję obliczającą silnię dla podanej w argumencie wartości i zwracającą (do Pythona) wynik.

Zadanie 1.0.2

Stwórz w C++ moduł Pythonowy (bibliotekę) udostępniający klasę `Vector3`, zawierającą pola publiczne `x`, `y`, `z` i metodę `len()` obliczającą długość wektora przechowywanego w obiekcie tej klasy. *Wskazówka: do obliczenia pierwiastka w C++ możesz użyć funkcji* ***`sqrt`****, dostępnej po zainkludowaniu pliku nagłówkowego* ***`math.h`****.*

Biblioteki C w Pythonie

Python potrafi także użyć niedostosowanych (nie posiadających API pythonowego) bibliotek języka C. Poniższy przykład korzysta z biblioteki `libX11.so.6` (związanej z system obsługującym środowisko graficzne w Linuxie) do włączenia "NumLock" na klawiaturze.

```python
from ctypes import *
X11 = cdll.LoadLibrary("libX11.so.6")

X11.XOpenDisplay.restype = c_void_p
display = X11.XOpenDisplay(None);
X11.XkbLockModifiers(
        c_void_p(display), c_uint(0x0100),
        c_uint(16), c_uint(16)
)
X11.XCloseDisplay(c_void_p(display))
```

Generatory wrapperów

Przedstawiony sposób tworzenia interfejsu pythonowego do programu/biblioteki w C++ jest dość pracochłonny dla większych projektów – wymaga pisania odpowiedniego fragmentu kodu C++ dla każdej wystawianej funkcji czy klasy. Tworzenie interfejsów bibliotek dla innych językach programowania (w tym Pythona) jest jednak na tyle użyteczne i popularne, że istnieją programy ułatwiające to zadanie. Nazywane generatorami wrapperów/interfejsów. Pozwalają one na dość łatwe automatyczne wystawienie wszystkich (lub wskazanych) klas i funkcji z kodu C/C++ do innego języka. Przykładem tego typu rozwiązań jest SWIG, czyli *Simplified Wrapper and Interface Generator* – http://www.swig.org/.

2 Uruchamianie kodu pythonowego z C++

Możemy także uruchomić kod pythonowy z poziomu C++, przekazywać do niego jakieś parametry (np. argumenty do pythonowej funkcji) oraz odbierać jakieś wartości (np. wyniki zwracane przez taką funkcję).

Przykładowy kod C++:

```
/*  PLIK: py_run.cpp      kompilacja:
                g++ -I/usr/include/python3.7m/ py_run.cpp -lpython3.7m
 ↪  -lboost_python37
        uwaga: - kolejność argumentów może być istotna
               - plik używa "py_api.cpp" z poprzedniego przykładu
               - plik wywołuje skrypt "py_run.script.py" z bieżącego katalogu
*/

// korzystamy z wcześniej przygotowanego (w "API pythonowe biblioteki C++")
 ↪  interfejsu pythonowego
#include "py_api.cpp"

int main(int, char **) {
        K1 *o1 = new K1();
        o1->a = 1;

        K1 *o2 = new K1();
        o2->a = 2;

        K1::obj = o1;

        std::cout << "o1->a = " << o1->a << "   o2->a = " << o2->a << "\n";

        // initialise python
        Py_Initialize();

        try {
                // initialise and import MyPyAPI module
                PyObject* module = PyInit_MyPyAPI();
                PyDict_SetItemString(PyImport_GetModuleDict(), "MyPyAPI", module);
                Py_DECREF(module);
                PyRun_SimpleString("import MyPyAPI" );
                // poprzez PyRun_SimpleString można też uruchamiać inne fragmenty
                 ↪  kodu pythonowego

                // prepare to run scripts
```

```
                boost::python::object main = boost::python::import("__main__");
                boost::python::object global(main.attr("__dict__"));

                // export object to python
                global["ck1"] = boost::python::ptr(o1);

                // run file
                boost::python::object result = boost::python::exec_file(
                        "./py_run.script.py", global, global
                );

                // import object from python
                boost::python::object script = global["script1"];

                // run python function with args from C++
                if(!script.is_none()) {
                                // run scripts
                                std::cout << "RUN\n";
                                std::cout << "return = " <<
                                 ↪  boost::python::extract<int>(
                                        script(boost::python::ptr(o2))
                                ) << "\n";
                }
        } catch(boost::python::error_already_set &) {
                PyErr_Print();
                exit(-1);
        }

        std::cout << "o1->a = " << o1->a << "   o2->a = " << o2->a << "\n";

        delete o1;
        delete o2;

        return 0;
}
```

Zauważ że kod ten używa wcześniej pokazanego kodu (py_api.cpp) celem udostępnienia pythonowi typów danych, których używamy do komunikacji z funkcjami w skrypcie pythonowym.
Kod Pythona zawierający kod używany z poziomu C++:

```
# PLIK: py_run.script.py
# uruchamiany przez kod C++ z py_api.cpp

print("początek pliku .py")

ck0 = MyPyAPI.Klasa.get()
print("ck0:", ck0.f1(0))
print("ck1:", ck1.f1(0))

ck0.a=4
print("ck0:", ck0.f1(0))
print("ck1:", ck1.f1(0))
```

```
def script1(arg):
        print("run script1")

        print("ck0:", ck0.f1(0))
        print("ck1:", ck1.f1(0))
        print("arg:", arg.f1(0))

        arg.a=13
        print("ck0:", ck0.f1(0))
        print("ck1:", ck1.f1(0))
        print("arg:", arg.f1(0))

        print("end script1")

        return arg.a + ck0.a;

print("koniec pliku .py")
```

Zadanie 2.0.1

Napisz program w C++, który uruchomi wskazany plik pythonowy, przekazując do niego (globalną) zmienną typu całkowitego. Kod pythonowy powinien wypisywać otrzymaną wartość oraz jej kwadrat.

Zadanie 2.0.2

Napisz program w C++, który uruchomi wskazaną funkcję pythonową, przekazując do niej dwa argumenty typu całkowitego. Pythonowa funkcja powinna pomnożyć otrzymane liczby i zwrócić wynik do kodu C++, który powinien je wypisać.