Python kód beágyazása C programokba

C, ismerd meg Pythont.
Python, ő C.

Meglepően kis erőfeszítéssel beépíthetjük a ***Python*** értelmezőt programjainkba, így rendkívül gyorsan valósíthatunk meg olyan funkciókat, amelyek *C* nyelven hónapokat vennének igénybe.
A nagyméretű, nagy teljesítményű alkalmazások nyelve ***Linuxon*** szinte mindig a *C*, esetleg a *C++*. Mindkettő igen hatékony nyelv, segítségükkel gyors, natív, lefordított programokat hozhatunk létre. Futásidejű rugalmasságuk viszont elég csekély. A lefordított kód statikus, ez pedig komoly akadály is lehet. Ha például bővítőmodulokkal akarjuk bővíteni a program funkcionalitását, dinamikus kapcsolási és egyéb problémák sorával kell szembenéznünk. Nem beszélve arról hogy a bővítéshez elengedhetetlen a *C*, vagy a *C++* ismerete, ami nagymértékben szűkíti azon felhasználók körét, akik bővítményeket tudnak írni.
Érdemes tehát valamilyen parancsnyelvet biztosítani, amellyel a bővítmények létrehozhatók.
A parancsnyelvek nagyobb futásidejű rugalmasságot és rövidebb fejlesztési időt biztosítanak, továbbá a tanulási idejük is rövidebb – mindezek biztosítják, hogy több felhasználó tudjon bővítményeket írni.
Egy parancsnyelv létrehozása azonban nem egyszerű feladat, és akár a program fejlesztésének központi eleme is lehet. Szerencsére nem kell saját parancsnyelvet létrehoznunk. A ***Python*** értelmezőt közvetlenül beépíthetjük alkalmazásainkba, így viszonylag kevés többletmunkával a ***Python*** teljes hatékonysága és rugalmassága elérhetővé válik.

A Python beépítése saját alkalmazásunkba

A címben említett feladat nagyon egyszerű. A ***Python Python.h*** nevű fejállományában minden meghatározás megtalálható, ami az értelmező beépítéséhez szükséges. Hatékonyabb lehetne a fordítás, ha a felületnek csak azokat az elemeit kellene beépítenünk, amelyekre szükségünk van, a ***Python*** azonban ezt nem teszi lehetővé. Ha megnézzük a ***Python.h*** fájlt, láthatjuk, hogy fontos makrókat határoz meg és számos fejállományt beemel, amelyekre később az egyes összetevőknek szükségük lesz.
Hogy fordításkor hozzákapcsoljuk a ***Python*** értelmezőt alkalmazásunkhoz, futtatnunk kell a `python-config` programot. Ez kilistázza a kapcsolási opciókat, melyeket át kell adnunk a fordítónak. Nálam ezek az alábbiak:

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

Egy egyszerű beágyazott alkalmazás

Mennyi kód kell tehát, hogy a ***Python*** értelmezőt *C* alkalmazásunkból futtathassuk? Igazán kevés. Mint az *1. Lista* mutatja, három sor kód is elegendő – beállítjuk az értelmezőt, átadjuk neki a végrehajtandó Python kódot egy karakterlánc formájában, majd lezárjuk az értelmezőt.
A `Py_Main()` függvénnyel beépíthetünk programunkba egy interaktív ***Python*** terminált is, ahogy az a *2. Listában* látható. Ez ugyanúgy nyit meg egy értelmezőt, mintha a ***Pythont*** közvetlenül a parancssorból futtattuk volna. Az értelmező bezárásakor a vezérlés visszatér az alkalmazásunkhoz.

1. lista A Python beágyazása, mindössze három sor kóddal

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

2. lista Interaktív Python értelmező beépítése

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

A Python környezet

Három sor kód elég az értelmező beágyazásához, de valljuk be, ***Python***

3. Lista A környezet másolása

```
// Hivatkozást szerzünk a
// main modulra.
PyObject* main_module =
    PyImport_AddModule
    ➥("__main__");
// Hozzáférünk a main modul
// szótárához és másolatot
// készítünk róla.
PyObject* main_dict =
    PyModule_GetDict
    ➥(main_module);
PyObject* main_dict_copy =
    PyDict_Copy(main_dict);
// Két különböző fájlt
// külön környezetben hajtunk
// végre.
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);
```

parancsok végrehajtása a programon belül nem túl szórakoztató és nem is igazán hasznos. Ám ez még közel sem minden, amit a ***Python*** kínál. Mielőtt azonban végignéznénk a lehetőségeket, vegyük szemügyre a ***Python*** környezet beállításának folyamatát.

A ***Python*** értelmező futtatásakor a környezet tulajdonságait a `__main__` modul névtér szótára tartalmazza. Minden globális függvény, osztály és változó megtalálható ebben a szótárban. Az interaktív értelmező, vagy egy parancsfájl futtatásakor többnyire nem kell foglalkoznunk ezzel a globális névtérrel. A beágyazott értelmező futtatásakor azonban gyakran kell ehhez a könyvtárhoz fordulnunk végrehajtandó függvényekre való hivatkozásokért, vagy létrehozandó osztályokért. Néha az egész globális könyvtárat másolnunk kell, hogy különböző kódok másmás környezetben futhassanak, például ha minden betöltött bővítménynek új környezetet akarunk biztosítani.

A `__main__` modul szótárának eléréséhez először egy hivatkozásra lesz szükségünk. Ehhez a `PyImport_AddModule()` függvénysegítségével juthatunk hozzá, amely megkeresi a modult a megadott név alapján és visszaad egy `PyObject` mutatót az objektum címével. Miért `PyObject`? Minden ***Python*** adattípus a `PyObject`-től származik, ami tehát a közös nevező. A ***Python*** értelmező legtöbb függvénye tehát nem valamely speciális ***Python*** adattípussal, hanem a `PyObject` objektumokkal dolgozik.

Ha megvan a `__main__` modulra mutató `PyObject`-ünk, a `PyModule_GetDict()` függvény által visszaadott `PyObject` mutatóval férhetünk hozzá a main modul szótárához. Ezután már más ***Python*** parancsok futtatásakor átadhatjuk ezt a hivatkozást.

A *3. Lista* mutatja, hogyan másolhatjuk le a globális környezetet két különböző ***Python*** fájl külön környezetben való futtatásához.

Hamarosan részletesen is bemutatjuk, hogyan működik a `PyRun_File()`, ám előbb nézzük meg a *3. Listát*, ami tartalmaz némi érdekességet. A `PyRun_File()` függvénynek, amellyel a fájlokat futtatjuk, kétszer adjuk át a környe zetet. Az első a globális környezet, amelyről beszéltünk már, a második pedig a helyi, amely a helyileg meghatározott változókat és függvényeket tartalmazza. Ez esetben a kettő ugyanaz, mivel a végrehajtandó kód felsőszintű. Ha egy függvényt dinamikusan akarnánk végrehajtani több ***C*** hívással, valószínűleg érdemesebb lenne egy helyi környezetet létrehozni és használni a globális szótár helyett. Többnyire persze nyugodtan átadhatjuk a globális környezetet globális és helyi paraméterként egyaránt.

Python adatszerkezetek elérése C-ből/C++-ból

Érdemes megfigyelni a *3. Lista* `PyDECREF()` függvényhívásait is. Ezek a memóriakezeléshez nyújtanak segítséget. A ***Python*** értelmező automatikusan gazdálkodik a memóriával, a hivatkozásokat pedig a programozó számára láthatatlanul kezeli. Amint észleli, hogy egy adott memóriaterületre való összes hivatkozás megszűnt, felszabadítja a feleslegessé vált területet. Ez pedig probléma forrása is lehet a ***C*** oldalon, mivel a ***C*** nem kezeli helyettünk a memóriát, ezért ha ***C***-ből hivatkozunk egy adatszerkezetre, a ***Python*** nem lesz képes automatikusan nyilvántartani a hivatkozásokat. A ***C*** alkalmazás bármennyi hivatkozást létrehozhat és tárolhat, anélkül, hogy a ***Python*** tudna róla.

A megoldás az, hogy a ***C*** kód, amely egy ***Python*** adatszerkezetre hivatkozik, maga számolja a hivatkozásokat. Amikor a ***Python*** visszaad egy objektumot a ***C*** programnak, az növeli a hivatkozásszámlálót eggyel. A ***C*** kód így azt tehet az objektummal, amit akar és nem áll fenn a veszélye, hogy az váratlanul törlődik. Ha pedig nincs már vele több dolga, törölheti a hivatkozást a `Py_DECREF()` függvény meghívásával.

Fontos, hogyha a ***C*** programban másolunk egy mutatót, a másolat élettartama pedig hosszabb, mint az eredetié, a hivatkozásszámlálót magunknak kell növelnünk a `Py_INCREF()` meghívásával. Ha például másolatot készítünk egy `PyObject` mutatóról, hogy egy tömbben tároljuk, valószínűleg meg kell hívnunk a `Py_INCREF()`-et, hogy megakadályozzuk, hogy a mutatott objektum a szemétgyűjtő áldozatává váljon az eredeti hivatkozás megszűnése után.

Kód végrehajtása fájlból

Lássunk egy hasznosabb példát arról, hogy hogyan építhetjük be a ***Pythont*** egy igazi alkalmazásba. A *4. Listában* egy kisebb program látható, amely parancssorból vesz át matematikai kifejezéseket, kiszámolja az eredményt, majd megjeleníti azt a kimeneten. Hogy még érdekesebb legyen a dolog, megadhatunk egy fájlt is, benne ***Python*** kódokkal, amelyet a program beolvas, mielőtt a kifejezéseket kiértékelné. Így meghatározhatunk függvé-

4. Lista Egyszerű kifejezéskalkulátor

```
#include <python2.3/Python.h>
void process_expression(char*
➥filename,
                        int num,
                        char**
                        ➥exp)
{
    FILE*        exp_file;
    // Létrehozunk egy
    // globális változót
    // a kifejezés
    // eredményének
    // megjelenítéséhez.
    PyRun_SimpleString("x =
    ➥0");
    // Megnyitjuk és
    // végrehajtjuk a
    // függvényeket
    // tartalmazó fájlt, hogy
    // a függvények
    // elérhetőek legyenek a
    // felhasználói
    // kifejezések számára.
    exp_file = fopen
    ➥(filename, "r");
    PyRun_SimpleFile
    ➥(exp_file, exp);
    // Végighaladunk a
    // kifejezéseken és
    // végrehajtjuk őket.
    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;
}
```

5. Lista Meghívható függvényhivatkozások

```
#include <python2.3/Python.h>
void process_expression(int
➥num, char* func_name)
{
    FILE*        exp_file;
    PyObject*    main_module,
    ➥* global_dict, *
    ➥expression;
    // Beállítunk egy
    // globális változót
    // a kifejezés
    // eredményének
    // megjelenítéséhez.
    PyRun_SimpleString("x =
    ➥0");
    // Megnyitjuk és
    // végrehajtjuk a Python
    // fájlt.
    exp_file = fopen(exp,
    ➥"r");
    PyRun_SimpleFile
    ➥(exp_file, exp);
    // Hivatkozást szerzünk
    // a main modulra és
    // a globális szótárra.
    main_module =
    ➥PyImport_AddModule
    ➥("__main__");
    global_dict =
    ➥PyModule_GetDict
    ➥(main_module);
    // Kinyerjük a
    // "func_name" függvényre
    // való hivatkozást
    // a globális szótárból.
    expression =
        PyDict_GetItemString
        ➥(global_dict,
        ➥func_name);
    while(num-) {
        // Meghívjuk az
        // "expression"
        // által hivatkozott
        // függvényt.
        PyObject_CallObject
        ➥(expression,
        ➥NULL);
    }
    PyRun_SimpleString("print
    ➥x");
}
```

nyeket, amelyek a parancssorból átadott kifejezésekből is elérhetők. Két alapvető ***Python API*** függvényt használtunk ebben a programban, a `PyRun_SimpleString()`-et és a `PyRun_AnyFile()`-t.
A `PyRun_SimpleString()`-gel már találkoztunk. Mindössze annyit csinál, hogy végrehajtja a megadott ***Python*** kifejezést a globális környezetben.
A `PyRun_SimpleFile()` hasonlít a korábban bemutatott `PyRun_File()` függvényhez, kivéve, hogy alapból a globális környezetben fut, így minden általa végrehajtott kifejezés, vagy kifejezéscsoport elérhető a később végrehajtandók számára is.

Meghívható függvényobjektumok

Kifejezés-kalkulátorunkkal nem csak kifejezések sorozatát értékelhetjük ki, de akár betölthetünk mondjuk egy `f()` függvényt is a megadott ***Python*** fájlból, és azt a parancssorban átadott számnak megfelelően többször végrehajtva a kifejezések összegét is kiszámolhatjuk. A függvényt a `PyRun_SimpleString("f()")` függvénnyel hajthatjuk végre, bár ez nem túl hatékony, mivel az értelmezőnek így be kell olvasnia és ki kell értékelnie a karakterláncot a függvény minden meghívásakor. Jobb lenne hivatkozni a függvényre, hogy közvetlenül meghívhassuk.
A ***Python*** az összes globálisan meghatározott függvényt a globális szótárban tárolja. Ha van egy hivatkozásunk erre a szótárra, onnan hivatkozhatunk bármely függvényre is.
A ***Python API*** kínál is függvényeket erre a célra. Működésük a *5. Listában* látható.
Hogy megkapjuk a függvényhivatkozást, a program először létrehoz egy hivatkozást a main modulra, „importálva” azt a `PyImport_AddModule("__main__")` függvénnyel. Ha ez megvan, a `PyModule_GetDict()` függvényhívással megszerzi a szótárát. Innentől csak a `PyDict_GetItemString(global_dict, "f")` hívásra van szükségünk, hogy a szótárból megkapjuk magát a függvényt.
Most, hogy megvan a hivatkozás a függvényre, meghívhatjuk a `PyObject_CallObject()` függvénnyel. Mint látható, egy mutatót

vesz át a meghívandó függvényre. A függvény már létezik a *Python* környezetben, így le is van fordítva. Meghívásakor tehát nincs beolvasás, vagy fordítás, így a függvény elég gyorsan végrehajtható.

Adatok átadása a függvényeknek

Most nyilván arra gondolunk, hogy ***„jaj, de jó, de mennyivel jobb lenne, ha ezeknek a függvényeknek adatokat is át tudnánk adni”***. És tudunk is. Egyik megoldás a titokzatos `NULL` érték használata - ezt adtuk át az *5. Listában* a `PyObject_CallObject()` függvénynek. Röviden bemutatjuk, hogyan is működik, de előbb lássunk egy sokkal egyszerűbb módot a függvények paraméterezésére *C*/*C++* adattípusok és a `PyObject_CallFunction()` segítségével. Ennek a függvénynek, ahelyett, hogy a *C* és *Python* közötti átalakítással kellene törődnünk, átadhatunk egy formátum karakterláncot és egy változó hosszúságú paraméterlistát, hasonlóan a `printf()` függvénycsaládhoz.
Visszatérve a kifejezés-kalkulátorunkhoz, tegyük fel, hogy egy nem folytonos értékek tartományból álló kifejezést akarunk kiértékelni.
Ha a kifejezést egy, a betöltött *Python* fájlban levő függvényben határoztuk meg, szerezhetünk rá hivatkozást a szokásos módon, majd végighaladhatunk a tartományon.
Minden értéknél hívjuk meg a `PyObject_CallFunction(expression, "i", num)`-ot. Az `"i"` karakterlánc azt jelzi, hogy egyetlen egész számot akarunk átadni. Ha a függvényünk két egész számot és egy karakterláncot venne át, ily módon hívnánk meg:
`PyObject_CallFunction (expression, "iis", num1, num2, string)`. Az esetleges visszatérési értéket a `PyObject_CallFunction()` függvény visszatérési értékeként egy `PyObject` mutató formájában kapjuk meg.
Python függvényeknek legegyszerűbben így adhatunk át argumentumokat, ám a legrugalmasabb megoldás nem ez. Gondoljunk csak bele, mi van, ha dinamikusan választjuk ki a meghívandó függvényt.
A nehézséget ekkor az okozza, hogy a különböző függvények más-más paramétereket várnak.
A `PyObject_CallFunction()`-nek viszont már a fordításkor meg kell adnunk az argumentumokat, ez pedig nem az a rugalmasság, amit egy parancsnyelvtől elvárnánk.
A megoldás kulcsa a `PyObject_CallObject()`. Ez a függvény beéri egyetlen *Python* objektumokból álló *tuple*-lel (tetszőleges, nem módosítható elemeket tartalmazó szekvencia) a változó hosszúságú *C* értékekből álló paraméterlista helyett. Hátránya viszont, hogy a natív *C* értékeket *Python* objektumokká kell alakítanunk, ám amennyit veszítünk sebességben, annyit nyerünk rugalmasságban. Persze mielőtt *Python* objektumok *tuple*-jeként átadnánk az értékeket, tudnunk kell, hogyan hozzuk létre ezt a *tuple*-t. Erről szólunk a következő részben.

Átalakítás Python és C adattípusok között

A *Python* értelmező `PyObject` objektumok formájában vesz át és ad vissza adatszerkezeteket. Ahhoz, hogy a megfelelő típust kapjuk, típusátalakítást kell végrehajtanunk. Átalakíthatunk például egy `PyObject` objektumot `PyIntObject()` objektummá. Ha nem ismerjük azonban a változó típusát, az átalakításnak katasztrofális következményei is lehetnek. Ilyen esetben meghívhatjuk valamelyik `Check()` függvényt, hogy megállapítsuk, az objektum típusa megfelelő-e. A `PyFloat_Check()` például igaz (true) értékkel tér vissza, ha az objektum átalakítható `float` (lebegőpontos) típusúvá. Más szóval igazat ad, ha az objektum `float`, vagy annak altípusa. Ha a pontos egyezést akarjuk vizsgálni, a `PyFloat_CheckExact()` függvényt használhatjuk.
A homályos `PyObject` adatszerkezet nem túl hasznos egy *C* program számára. Ahhoz, hogy a *Python* adatait elérhessük a programunkban, különböző átalakító függvények állnak rendelkezésre, amelyek natív *C* adatokat adnak vissza. Ha egy `PyObject` objektumot `long int`-té (duplapontos egész) akarunk alakítani, csak meg kell hívnunk a `PyInt_AsLong()` függvényt. A `PyInt_AsLong()` biztonságos függvény, ellenőrzött típusátalakítást fog végezni `PyIntObject`-té, mielőtt visszaadná a `long int` értéket.
Ha biztosan tudjuk, hogy az átalakítandó érték valóban int, felesleges lehet az ellenőrzés, különösen egy ciklus belsejében.
A *Python* függvények gyakran vesznek át *Python* objektumokat *tuple*-k, vagy listák formájában. Ezeknek a típusoknak nincsen *C* megfelelőjük, de a *Python* biztosít függvényeket, amelyekkel létrehozhatók *C* adattípusokból. Lássuk most egy *tuple* létrehozását - erre szükségünk lesz, ha a `PyObject_CallObject()`-tel akarunk függvényeket meghívni.
Az első lépés egy új *tuple* létrehozása a `PyTuple_New()` függvénnyel, ami átveszi a *tuple* hosszát és visszaad egy, az új *tuple*-t jelölő `PyObject` mutatót. Ezután a `PyTuple_SetItem()`-mel értéket adhatunk az egyes elemeknek, minden elemet egy `PyObject` mutatóként megadva.

Összefoglalás

Mostanra elegendő alapismeretre tettünk szert, hogy elkezdhessünk *Python* szkripteket használni saját programjainkban. Lásd még a *Python* dokumentációjának ***„Extending and Embedding the Python Interpreter”*** (***A Python értelmező bővítése és beágyazása***) című részét, ami további információval szolgál, de bemutatja a másik irányt is, vagyis a *C* függvények beépítését Pythonba. Hasznos forrás még a ***„Python/C API Reference Manual”*** (***Python/C API Referencia Kézikönyv***), amely részletes leírást tartalmaz a *Python* beágyazásához használható függvényekhez. A *LinuxJournal* archívumában megtalálható *Ivan Pulleyn* kitűnő cikke a *Pythont* beágyazó többszálú programokról.

Linux Journal 2006., 142. szám

William Nagel

Vezető szoftvertervező egy kis szoftverfejlesztő cégnél, ahol Linux alapú real-time rendszereket fejleszt. Ő a szerzője a „SubversionVersion Control: Using the Subversion Version Control System in Development Projects” című könyvnek.

KAPCSOLÓDÓ CÍMEK

➲ www.linuxjournal.com/article/8714