Python : liaison dynamique avec une bibliothèque de fonctions C

1. Introduction

Ce document montre comment programmer des fonctions en langage C et obtenir une bibliothèque à chargement dynamique (*dynamic library*), appelée aussi bibliothèque partagée. On verra comment le module Python ctypes permet d'accéder au fonctions de cette bibliothèque dynamique.
La méthode exposée est la plus rapide (à programmer) pour accéder depuis Python à des fonctions écrites en C/C++. La programmation de telles fonctions peut être nécessaire pour le calcul numérique intensif ou le traitement d'image.
On désignera la bibliothèque dynamique par l'abbréviation DLL (Dynamic Load Library), qui est effectivement l'extension des fichiers de bibliothèque dynamique dans Windows. Une DLL est chargée en mémoire dès qu'une application demande son utilisation (chargement dynamique). Si plusieurs applications l'utilisent en même temps, il n'y a qu'une seule copie du code exécutable en mémoire (bibliothèque partagée) mais chaque application possède bien sûr ses propres données associées à la DLL.
Une autre manière d'écrire des fonctions en C et de les utiliser en Python est d'écrire une extension en C pour Python (Extending and Embedding the Python Interpreter). Cette méthode, que nous n'abordons pas ici, est plus complexe à mettre en œuvre que l'utilisation de ctypes. Elle nécessite de plus une compilation pour chaque version de Python, ce qui n'est pas le cas de la méthode de la DLL que nous exposons ici.

2. Code source en C

Nous présentons un exemple simple de code source en C contenant quelques fonctions dont le but est de montrer le fonctionnement de l'interface avec Python. Ce code source peut contenir des fonctions mais aussi des données déclarées sous la forme de variables globales (variables d'état du module). Dans les cas les plus simples, on écrira simplement des fonctions indépendantes les unes des autres, sans utilisation de données globales. Pour le compilateur Microsoft MSVC (Visual Studio), la déclaration d'une fonction exportée par la DLL doit porter en préfixe la directive suivante :

```
__declspec(dllexport)
```

Pour la compilation sous Linux avec GCC, aucune directive n'est requise : toutes les fonctions déclarées sont publiées dans la DLL.
Pour cet exemple, nous plaçons les déclarations et les définitions des fonctions dans le fichier `fonctions.c`.
Voici la déclaration des fonctions :
fonctions.c

```
#ifdef _WIN32
    #define EXPORT __declspec(dllexport)
#elif _WIN64
    #define EXPORT __declspec(dllexport)
#else
    #define EXPORT
```

```
#endif

extern EXPORT float add(float a, float b);
extern EXPORT void mul2(float *a);
extern EXPORT void mulArray(double *a, int n, double x);
extern EXPORT double meanArray(double *a, int n);
extern EXPORT int sumArray8bits(unsigned char *a, int n);
extern EXPORT void incCount(void);
extern EXPORT int getCount(void);
```

On ajoute au fichier une variable globale (un entier) qui est initialisée à 0 lorsque la DLL est ouverte par une application (la variable est propre à chaque application) :

```
int count=0;
```

La fonction `add` effectue l'addition de deux flottants 32 bits et renvoie le résultat. Voici sa définition :

```
float add(float a, float b) {
    return a+b;
}
```

C'est le type de fonction le plus simple, avec des arguments transmis par valeur et une valeur renvoyée. Il est fréquent que le résultat du calcul effectué par une fonction ne puisse pas se réduire à un type de base (entier ou flottant). Dans ce cas, on est amené à transmettre en argument de la fonction des pointeurs. La fonction `mul2` prend comme argument un pointeur vers un flottant 32 bits et effectue la multiplication par 2 de ce flottant :

```
void mul2(float *a) {
    *a *=2;
}
```

La variable `a` contient un pointeur vers un `float`. L'expression `*a` signifie *objet d'adresse a* (et `float *a` signifie donc que l'objet d'adresse `a` est un float). L'objet d'adresse `a` est donc multiplié par 2.
Remarque : en langage C et plus encore en C++, il est courant d'utiliser la transmission par référence pour obtenir le même résultat. La même fonction avec transmission par référence s'écrirait :

```
void mul2_ref(float &a) {
    a *=2;
}
```

Dans ce cas, l'appel de la fonction se ferait par `mul2_ref(b)` où `b` désigne un flottant (et non pas un pointeur). Dans le premier cas, l'appel se fait par `mul2(&b)` car il faut transmettre l'adresse de `b`. La différence entre les deux fonctions est simplement syntaxique car elles réalisent exactement la même opération, à savoir la multiplication par 2 d'un flottant stocké en mémoire. Il est d'ailleurs probable que le code exécutable soit le même pour les deux fonctions. Nous préférons la première syntaxe car, comme nous le verrons plus loin, son interfaçage avec Python est très simple.
La fonction `mulArray` multiplie par un flottant 64 bits tous les éléments d'un tableau de flottants 64 bits. Le premier argument est un pointeur sur le premier élément du tableau, le deuxième est le nombre d'éléments et le troisième et le multiplicateur.

```
int void mulArray(double *a, int n, double x) {
    int i;
    for (int i=0; i<n; i++) a[i] *=x;
}
```

Il est intéressant de remarquer la similitude avec la fonction `mul2` : dans les deux cas, on transmet un pointeur vers un flottant et on modifie les données en mémoire. Pour la fonction `mul2`, on aurait d'ailleurs pu utiliser la syntaxe `a[0]=2` pour modifier le flottant. Si le taille du tableau est fixe, le deuxième argument peut bien sûr être ommis.
La fonction `meanArray` calcule la moyenne des éléments d'un tableau de flottants 64 bits et renvoie le résultat en valeur de retour :

```
double meanArray(double *a, int n) {
    int i;
    double m;
    m = 0.0;
    for (int i=0; i<n; i++) m += a[i];
    return m/n;
}
```

La fonction `sumArray8bits` calcule la somme des éléments d'un tableau d'entiers 8 bits (ce type de tableau est courant en traitement d'image) :

```
int sumArray8bits(unsigned char *a, int n) {
    int i,s;
    s = 0;
    for (int i=0; i<n; i++) s += a[i];
    return s;
}
```

La fonction `incCount` incrémente la variable globale `count` :

```
void incCount() {
    count += 1;
}
```

r La fonction `getCount` envoit la valeur de cette variable :

```
int getCount() {
    return count;
}
```

3. Compilation

3.a. Windows

Pour la configuration de la compilation, nous utilisons CMake. CMake est un logiciel permettant d'écrire un fichier de configuration valable pout tout compilateur et pour toute plateforme (Windows, MacOS, Linux). Le fichier de configuration est nommé `CMakeLists.txt`. CMake génère à partir de ce fichier un fichier de configuration pour le compilateur choisi.
Voici le contenu du fichier `CMakeLists.txt` pour la compilation de `fonctions.c` :
CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(testPythonCtypes)
set(SRCS fonctions.c)
add_library(fonctions SHARED ${SRCS})
```

La commande `add_library` indique qu'il faut générer un exécutable sous la forme d'une bibliothèque (*library*) et que celle-ci est de type SHARED (bibliothèque partagée, c'est-à-dire une DLL).
Supposons que ce fichier de configuration et le fichier `fonctions.c` soit placés dans le même dossier. Dans une console et dans ce dossier, on exécute cmake de la manière suivante (dans Windows) :

```
cmake -A x64 -S . -B buildX64
```

L'option -A précise l'architecture : ici, nous voulons créer une DLL 64 bits. L'option -S précise le dossier où se trouve les sources (ici le dossier où l'on se trouve). L'option -B précise le dossier où CMake placera les fichiers de configuration pour la compilation.
Le *générateur* désigne le logiciel de génération des fichiers de compilation, qui cible un compilateur particulier. Sur windows, le générateur distribué par Microsoft est Visual Studio ; il s'agit en fait d'un IDE dont l'installation comporte le générateur et le compilateur MSVC. Pour connaître la liste des générateurs acceptés par CMake sur la platforme, il suffit d'exécuter `cmake --help` dans une console. Voici cette liste sur Windows :

```
* Visual Studio 17 2022        = Generates Visual Studio 2022 project files.
                                  Use -A option to specify architecture.
  Visual Studio 16 2019        = Generates Visual Studio 2019 project files.
                                  Use -A option to specify architecture.
  Visual Studio 15 2017 [arch] = Generates Visual Studio 2017 project files.
                                  Optional [arch] can be "Win64" or "ARM".
  Visual Studio 14 2015 [arch] = Generates Visual Studio 2015 project files.
                                  Optional [arch] can be "Win64" or "ARM".
```

```
  Visual Studio 12 2013 [arch] = Generates Visual Studio 2013 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 11 2012 [arch] = Generates Visual Studio 2012 project files.
                                 Optional [arch] can be "Win64" or "ARM".
  Visual Studio 10 2010 [arch] = Deprecated.  Generates Visual Studio 2010
                                 project files.  Optional [arch] can be
                                 "Win64" or "IA64".
  Visual Studio 9 2008 [arch]  = Generates Visual Studio 2008 project files.
                                 Optional [arch] can be "Win64" or "IA64".
  Borland Makefiles            = Generates Borland makefiles.
  NMake Makefiles              = Generates NMake makefiles.
  NMake Makefiles JOM          = Generates JOM makefiles.
  MSYS Makefiles               = Generates MSYS makefiles.
  MinGW Makefiles              = Generates a make file for use with
                                 mingw32-make.
  Green Hills MULTI            = Generates Green Hills MULTI files
                                 (experimental, work-in-progress).
  Unix Makefiles               = Generates standard UNIX makefiles.
  Ninja                        = Generates build.ninja files.
  Ninja Multi-Config           = Generates build-<Config>.ninja files.
  Watcom WMake                 = Generates Watcom WMake makefiles.
  CodeBlocks - MinGW Makefiles = Generates CodeBlocks project files.
  CodeBlocks - NMake Makefiles = Generates CodeBlocks project files.
  CodeBlocks - NMake Makefiles JOM
                               = Generates CodeBlocks project files.
  CodeBlocks - Ninja           = Generates CodeBlocks project files.
  CodeBlocks - Unix Makefiles  = Generates CodeBlocks project files.
  CodeLite - MinGW Makefiles   = Generates CodeLite project files.
  CodeLite - NMake Makefiles   = Generates CodeLite project files.
  CodeLite - Ninja             = Generates CodeLite project files.
  CodeLite - Unix Makefiles    = Generates CodeLite project files.
  Eclipse CDT4 - NMake Makefiles
                               = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - MinGW Makefiles
                               = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Ninja         = Generates Eclipse CDT 4.0 project files.
  Eclipse CDT4 - Unix Makefiles= Generates Eclipse CDT 4.0 project files.
  Kate - MinGW Makefiles       = Generates Kate project files.
  Kate - NMake Makefiles       = Generates Kate project files.
  Kate - Ninja                 = Generates Kate project files.
  Kate - Unix Makefiles        = Generates Kate project files.
  Sublime Text 2 - MinGW Makefiles
                               = Generates Sublime Text 2 project files.
  Sublime Text 2 - NMake Makefiles
                               = Generates Sublime Text 2 project files.
  Sublime Text 2 - Ninja       = Generates Sublime Text 2 project files.
  Sublime Text 2 - Unix Makefiles
                               = Generates Sublime Text 2 project files.
```

L'astérisque indique le générateur par défaut, qui est d'ailleurs le seul installé sur notre système. Il s'agit de Visual Studio 17. CMake génère donc, dans le dossier `buildX64`, les fichiers de configuration pour la compilation par ce générateur (avec une architecture 64 bits). Pour effectuer la compilation, il faut aller dans le dossier `buildX64` depuis une console Visual Studio et exécuter la commande suivante :

```
msbuild fonctions.vcxproj
```

Il y a aussi un fichier `testPythonCtypes.sln` pour l'IDE Visual Studio mais l'utilisation de ce dernier n'est pas nécessaire.
La DLL est générée dans le sous-dossier `Debug`.
Pour générer une DLL 32 bits, il faut exécuter CMake de la manière suivante :

```
cmake -A Win32 -S . -B buildWin32
```

Remarquons que l'architecture (32 ou 64 bits) de la DLL doit correspondre à celle de la version de Python utilisée.

3.b. Linux

Sous Linux, le générateur est `Unix Makefiles` et le compilateur est GCC.
Depuis le dossier où se trouve le fichier `CMakeLists.txt` et le code source `fonctions.c`, on exécute :

```
cmake -S . -B build
```

puis dans le dossier `build` :

```
make
```

La DLL est générée dans le dossier `build` sous la forme d'un fichier `libfonctions.so`. Sous Linux, on parle plutôt de bibliothèque partagée (*shared library*) mais il s'agit bien du même type d'exécutable que la DLL sous Windows (une bibliothèque à chargement dynamique, partagée par plusieurs applications).

3.c. MacOS

Le fichier de configuration de CMake (`CMakeLists.txt`) est identique et la commande cmake s'exécute de la même manière mais la déclaration des fonctions à exporter dans le code source peut nécessiter (en fonction du compilateur utilisé) une directive spécifique.

4. Accès à la DLL dans Python

Le module Python ctypes (module de la bibliothèque standard de Python) permet d'accéder directement aux fonctions d'une DLL. Numpy comporte des fonctions similaires à celles de ctypes : certaines fonctions sont directement accessibles depuis un tableau `numpy.ndarray`, d'autres sont dans le module numpy.ctypeslib.
Voici les modules à importer :
testFonctions.py

```
import ctypes as ct
import numpy as np
import sys
```

Le première étape est le chargement de la DLL. Lors du travail de débogage, la DLL se trouve dans le dossier `buildWin32/Debug` ou `buildX64/Debug` sous Windows et `build` sous Linux. Si le script Python est placé dans le même dossier, la DLL devrait être trouvée sans que son chemin complet soit spécifié mais il semble que cela ne fonctionne pas toujours. De plus, il faut charger la DLL correspondant à la platforme (Windows ou Linux) et à l'achitecture de Python (32 ou 64 bits). Voici comment charger la bonne DLL :

```
if sys.platform=="win32": # Windows 32 ou 64 bits
    try:
        fcts = ct.cdll.LoadLibrary("C:\\Users\\fredl\\Documents\\CPP\\testPythonCtypes\\
    except:
        fcts = ct.cdll.LoadLibrary("C:\\Users\\fredl\\Documents\\CPP\\testPythonCtypes\\
else: # linux
    ftcs = ct.cdll.loadLibrary("~/Documents/CPP/testPythonCtypes/build/libfonctions.so")
```

Nous n'avons pas mis le chargement de la DLL sous MacOS (dans ce cas `sys.platform=darwin`). Voici comment se fait l'appel de la fonction `float add(float a, float b)` :

```
fcts.add.restype = ct.c_float
y = fcts.add(ct.c_float(1.5),ct.c_float(2.0))
print(y)
```

La première ligne permet de préciser le type de retour de la fonction. Voir ctypes pour les différentes types de données. Pour appeler la fonction (deuxième ligne), il faut convertir chaque donnée transmise en argument (ici les flottants 1.5 et 2.0) en le type qui convient, ici `ctypes.c_float`, qui correspond bien sûr au type `float` du langage C/C++ (flottant 32 bits). ctypes ne fait jamais de conversion de type implicite : oublier la conversion déclenche une erreur. D'ailleurs, dans le cas présent, il y a aurait une ambiguïté sur le type de flottant à transmettre (32 ou 64 bits). Remarquons que `ctypes.c_float(1.5)` créé en fait un objet (voir ci-dessous).
L'appel de la fonction `void mul2(float *a)` nécessite la transmission d'un pointeur sur un flottant 32 bits :

```
x = ct.c_float(2.2)
fcts.mul2(ct.pointer(x))
print(x.value)
```

La première ligne crée un objet de type `ctypes.c_float` contenant la valeur 2.2. La fonction `ctypes.pointer` crée un objet qui contient le pointeur qui doit être transmis à la fonction de la DLL. Après l'appel, il suffit de consulter l'attribut `value` de l'objet `x` pour voir le résultat.
Pour l'appel de la fonction `void mulArray(double *a, int n, double x)`, nous voulons transmettre un pointeur sur les données d'un tableau `numpy.ndarray`. Cela ne pose pas de problème car ce tableau est implémenté par un tableau C, c'est-à-dire un espace mémoire contenant des flottants 64 bits placés les uns à la suite des autres. Il faut transmettre le pointeur sur le premier élément du tableau et le nombre d'éléments. Voici comment se fait l'appel :

```
N = 10
A = np.arange(N)*0.1
fcts.mulArray(A.ctypes.data_as(ct.POINTER(ct.c_double)),ct.c_int(N),ct.c_double(2.0))
print(A)
```

L'appel `A.ctypes.data_as(ct.POINTER(ct.c_double))` fournit le pointeur sur le tableau en mémoire associé au tableau `numpy.ndarray` nommé `A`. Il faut remarquer que l'appel et l'exécution de `mulArray` se fait sans aucune copie des données du tableau. Le contenu du tableau `numpy.ndarray` est donc modifié directement (chaque élément est mutliplié par `x`).
L'appel de la fonction `double meanArray(double *a, int n)` est similaire au précédent mais il faut préciser le type de retour de la fonction :

```
B = np.arange(N)*0.1
fcts.meanArray.restype = ct.c_double
print(fcts.meanArray(B.ctypes.data_as(ct.POINTER(ct.c_double)),ct.c_int(N)))
print(B.mean())
```

L'appel de la fonction `int sumArray8bits(unsigned char *a, int n)` est similaire au précédent. La différence est qu'il faut créer un tableau d'octets non signés et faire la bonne conversion pour générer le pointeur :

```
C = np.array(np.arange(10),dtype=np.ubyte)
fcts.sumArray8bits.restype = ct.c_int
print(fcts.sumArray8bits(C.ctypes.data_as(ct.POINTER(ct.c_ubyte)),ct.c_int(N)))
print(C.sum())
```

Il faut noter que la création d'un tableau de flottants (`C=np.array(np.arange(10))` ne provoquerait pas de message d'erreur mais le résultat du calcul serait faux car le tableau de flottants serait interprété comme un tableau d'octets (ce qui ne provoquerait pas de violation d'espace mémoire).
Voici pour finir l'utilisation des fonctions `void incCount()` et `int getCount` :

```
fcts.incCount()
print(fcts.getCount())
fcts.incCount()
print(fcts.getCount())
```

Ces deux fonctions permettent de vérifier qu'il est possible d'associer des variables globales à la DLL et que le contenu de ces variables est bien propre à chaque application utilisant la DLL.
Voici la sortie console du script Python précédent, qui permet de vérifier son bon fonctionnement :

```
3.5
4.400000095367432
[0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8]
0.45000000000000007
0.45000000000000007
45
45
1
2
```