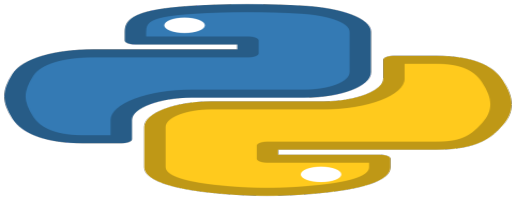
Interfaçage Python/C

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel



Plan

Question

Pourquoi interfacer **Python** avec du **C** ?

Introduction

Question

Pourquoi interfacer **Python** avec du **C** ?

Pou répondre à cette question, il faut comprendre la philosophie de chacun.

Python

- Langage de haut niveau, expressif et très productif.
- Gestion automatique de la mémoire.
- Écosystème très riche pour le web, la data, l'IA, l'automatisation...

Introduction

Python

- Langage de haut niveau, expressif et très productif.
- Gestion automatique de la mémoire.
- Écosystème très riche pour le web, la data, l'IA, l'automatisation...

C

- Langage compilé, proche de la machine.
- Très performant pour les traitements bas niveau.
- Très utilisé dans les systèmes, bibliothèques natives et logiciels embarqués.

Conclusion

Python et **C** : deux philosophies différentes.

Pourquoi mélanger **Python** et **C** ?

Python est très pratique pour écrire rapidement des programmes, mais certaines parties peuvent nécessiter plus de performance ou un accès direct à du code natif.

Pourquoi mélanger **Python** et **C** ?

Python est très pratique pour écrire rapidement des programmes, mais certaines parties peuvent nécessiter plus de performance ou un accès direct à du code natif.

Cas fréquents

- accélérer une fonction critique
- réutiliser une bibliothèque **C** existante
- accéder à des fonctionnalités système
- communiquer avec du matériel
- écrire des extensions **Python** performantes

De nombreuses bibliothèques **Python** utilisent du **C**

- `NumPy` utilise du code natif pour accélérer les calculs numériques.
- `SciPy` s'appuie sur des bibliothèques scientifiques performantes.
- `OpenCV` fournit des bindings **Python** vers une bibliothèque **C**/**C++**.
- `TensorFlow` et `PyTorch` exposent une **API Python** au-dessus de moteurs natifs optimisés.

Introduction

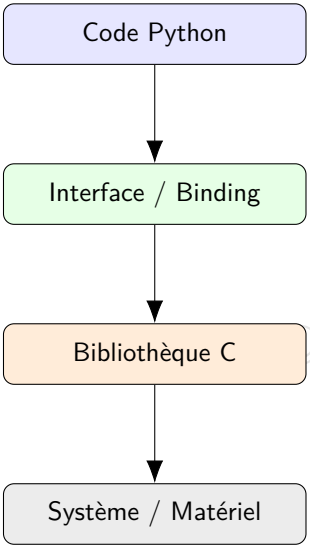
De nombreuses bibliothèques **Python** utilisent du **C**

- `NumPy` utilise du code natif pour accélérer les calculs numériques.
- `SciPy` s'appuie sur des bibliothèques scientifiques performantes.
- `OpenCV` fournit des bindings **Python** vers une bibliothèque **C**/**C++**.
- `TensorFlow` et `PyTorch` exposent une **API Python** au-dessus de moteurs natifs optimisés.

Remarque

Python est souvent utilisé comme langage d'orchestration, tandis que les calculs lourds sont délégués à du code natif.

Introduction



Architecture générale

L'interfaçage permet à **Python** d'appeler du code compilé, souvent plus rapide ou déjà disponible.

Solutions courantes

- `ctypes` : appeler une bibliothèque **C** sans écrire d'extension **Python**.
- **API Python/C** : écrire directement une extension **C** pour **Python**.
- **Cython** / **Pyrex** : écrire un code proche de **Python** générant du **C**.
- Embedding : intégrer l'interpréteur **Python** dans une application **C**.
- Profilage : mesurer avant d'optimiser.

Présentation du module ctypes

ctypes

Module standard de **Python** qui permet d'appeler des fonctions écrites en **C** depuis du code **Python**.

Présentation du module ctypes

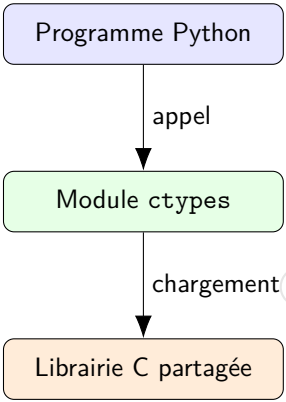
ctypes

Module standard de **Python** qui permet d'appeler des fonctions écrites en **C** depuis du code **Python**.

Avantages

- Disponible dans la bibliothèque standard.
- Ne nécessite pas forcément de compiler une extension **Python**.
- Permet de charger des bibliothèques partagées : `.so`, `.dll`, `.dylib`.

Présentation du module ctypes



Principe de fonctionnement de ctypes

Python ne compile pas le code **C** : il charge une bibliothèque déjà compilée et appelle ses fonctions.

Présentation du module ctypes

Import recommandé

```
import ctypes
```

Présentation du module ctypes

Import recommandé

```
import ctypes
```

Ou

```
from ctypes import CDLL, c_int, c_double
```

Présentation du module ctypes

Sous Linux

```
import ctypes

lib = ctypes.CDLL("./libcalcul.so")
```

Présentation du module ctypes

Sous Linux

```
import ctypes

lib = ctypes.CDLL("./libcalcul.so")
```

Sous Windows

```
import ctypes

lib = ctypes.CDLL("./calcul.dll")
```

Présentation du module ctypes

Sous Linux

```
import ctypes

lib = ctypes.CDLL("./libcalcul.so")
```

Sous Windows

```
import ctypes

lib = ctypes.CDLL("./calcul.dll")
```


Important

Le fichier chargé doit être une bibliothèque partagée compilée pour le même système et la même architecture que **Python**.

Extensions des bibliothèques partagées selon le système d'exploitation

Système	Extension	Exemple
Linux	`.so`	`libcalcul.so`
macOS	`.dylib`	`libcalcul.dylib`
Windows	`.dll`	`calcul.dll`

Présentation du module ctypes

Extensions des bibliothèques partagées selon le système d'exploitation

Système	Extension	Exemple
Linux	`.so`	`libcalcul.so`
macOS	`.dylib`	`libcalcul.dylib`
Windows	`.dll`	`calcul.dll`

Attention

Une bibliothèque compilée pour Linux ne peut pas être utilisée directement sous **Windows**, et inversement.

Correspondance des types fréquents **C** et ctypes

Type C	Type ctypes
`int`	`ctypes.c_int`
`long`	`ctypes.c_long`
`float`	`ctypes.c_float`
`double`	`ctypes.c_double`
`char *`	`ctypes.c_char_p`
`void`	`None`

Présentation du module ctypes

Correspondance des types fréquents **C** et ctypes

Type C	Type ctypes
`int`	`ctypes.c_int`
`long`	`ctypes.c_long`
`float`	`ctypes.c_float`
`double`	`ctypes.c_double`
`char *`	`ctypes.c_char_p`
`void`	`None`

Pourquoi préciser les types ?

Cela permet à `ctypes` de convertir correctement les valeurs entre **Python** et **C**.

Présentation du module ctypes

Déclarer les paramètres d'une fonction avec `argtypes`

```
lib.add.argtypes = [ctypes.c_int, ctypes.c_int]
```

Présentation du module ctypes

Déclarer les paramètres d'une fonction avec `argtypes`

```
lib.add.argtypes = [ctypes.c_int, ctypes.c_int]
```


Signification

- La fonction add attend deux entiers **C**.
- **Python** vérifie et convertit les arguments fournis.
- Cela rend les appels plus sûrs et plus lisibles.

Déclarer le type de retour avec `restype`

```
lib.add.restype = ctypes.c_int
```

Déclarer le type de retour avec `restype`

```
lib.add.restype = ctypes.c_int
```


Signification

- La fonction **C** retourne un entier.
- **Python** reçoit la valeur et la transforme en objet **Python**.
- Sans cette information, le résultat peut être mal interprété.

Présentation du module ctypes

ctypes est pratique, mais pas magique

- Il faut connaître précisément les signatures des fonctions **C**.
- Les erreurs de type peuvent provoquer des comportements indéfinis.
- La gestion de la mémoire reste délicate.
- Les structures, pointeurs et tableaux demandent plus de rigueur.

Présentation du module ctypes

ctypes est pratique, mais pas magique

- Il faut connaître précisément les signatures des fonctions **C**.
- Les erreurs de type peuvent provoquer des comportements indéfinis.
- La gestion de la mémoire reste délicate.
- Les structures, pointeurs et tableaux demandent plus de rigueur.

À retenir

ctypes convient bien pour appeler des fonctions **C** simples ou des bibliothèques existantes.

Chargement d'une librairie C

Considérons le fichier `calcul.c`

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

int sub(int a, int b)
{
    return a - b;
}
```

Chargement d'une librairie C

Considérons le fichier `calcul.c`

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

int sub(int a, int b)
{
    return a - b;
}
```


Objectif

Nous allons compiler ce fichier **C** en bibliothèque partagée, puis l'appeler depuis **Python**.

Chargement d'une librairie C

Compilation sous Linux

```
gcc -shared -fPIC calcul.c -o libcalcul.so
```

Chargement d'une librairie C

Compilation sous Linux

```
gcc -shared -fPIC calcul.c -o libcalcul.so
```


Explication

- `-shared` : produit une bibliothèque partagée.
- `-fPIC` : génère du code indépendant de sa position mémoire.
- `-o libcalcul.so` : nom du fichier généré.

Chargement d'une librairie C

Compilation sous Windows avec MinGW

```
gcc -shared calcul.c -o calcul.dll
```

Chargement d'une librairie C

Compilation sous Windows avec MinGW

```
gcc -shared calcul.c -o calcul.dll
```


Remarque

- Il faut utiliser une chaîne de compilation compatible avec l'architecture de **Python** : 32 bits ou 64 bits.
- Par exemple, Un **Python** 64 bits doit charger une **DLL** 64 bits.

Chargement d'une librairie C

Exemple d'arborescence (Organisation minimale du projet)

```
projet/
   |- calcul.c
   |- libcalcul.so
   |- main.py
```

Chargement d'une librairie C

Exemple d'arborescence (Organisation minimale du projet)

```
projet/
   |- calcul.c
   |- libcalcul.so
   |- main.py
```


Conseil

Pour commencer, placez le script **Python** et la bibliothèque compilée dans le même dossier.

Chargement d'une librairie C

Chargeons la bibliothèque depuis `main.py`

```
import ctypes

lib = ctypes.CDLL("./calcul.dll")

print(lib)
```

Chargement d'une librairie C

Chargeons la bibliothèque depuis `main.py`

```python
import ctypes

lib = ctypes.CDLL("./calcul.dll")

print(lib)
```


Résultat attendu

Si le chargement réussit, **Python** affiche un objet représentant la bibliothèque chargée.

Chargement d'une librairie C

Erreurs fréquentes lors du chargement

- fichier introuvable
- mauvais chemin relatif ou absolu
- architecture incompatible
- dépendance native manquante
- bibliothèque compilée pour un autre système

Chargement d'une librairie C

Erreurs fréquentes lors du chargement

- fichier introuvable
- mauvais chemin relatif ou absolu
- architecture incompatible
- dépendance native manquante
- bibliothèque compilée pour un autre système

Bonne pratique

Commencer par un exemple très simple permet de distinguer les erreurs de compilation, de chargement et d'appel.

Appel d'une fonction C

Code Python pour appeler une fonction simple

```
import ctypes

lib = ctypes.CDLL("./libcalcul.so")

lib.add.argtypes = [ctypes.c_int, ctypes.c_int]
lib.add.restype = ctypes.c_int

resultat = lib.add(10, 20)
print(resultat)
```

Appel d'une fonction C

Code Python pour appeler une fonction simple

```python
import ctypes

lib = ctypes.CDLL("./libcalcul.so")

lib.add.argtypes = [ctypes.c_int, ctypes.c_int]
lib.add.restype = ctypes.c_int

resultat = lib.add(10, 20)
print(resultat)
```


Résultat

```
30
```

Appel d'une fonction C

Côté C : fonction appelée

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

Appel d'une fonction C

Côté C : fonction appelée

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


Correspondance

- `int a` devient `ctypes.c_int`.
- `int b` devient `ctypes.c_int`.
- `return a + b` est converti en entier **Python**.

Appel d'une fonction C

Appeler une fonction avec des doubles

```c
double moyenne(double a, double b)
{
    return (a + b) / 2.0;
}
```

Appel d'une fonction C

Appeler une fonction avec des doubles

```c
double moyenne(double a, double b)
{
    return (a + b) / 2.0;
}
```

Code Python

```python
lib.moyenne.argtypes = [ctypes.c_double, ctypes.c_double]
lib.moyenne.restype = ctypes.c_double

print(lib.moyenne(12.5, 15.5))
```

Appeler une fonction sans retour

```c
void afficher(int n)
{
    printf("Valeur : %d\n", n);
}
```

Appeler une fonction sans retour

```c
void afficher(int n)
{
    printf("Valeur : %d\n", n);
}
```

Code Python

```python
lib.afficher.argtypes = [ctypes.c_int]
lib.afficher.restype = None

lib.afficher(42)
```

Appeler une fonction sans retour

```c
void afficher(int n)
{
    printf("Valeur : %d\n", n);
}
```

Code Python

```python
lib.afficher.argtypes = [ctypes.c_int]
lib.afficher.restype = None

lib.afficher(42)
```


Remarque

Pour compiler ce code **C**, il faut inclure `stdio.h` (`#include "stdio.h"`).

Passage d'une chaîne de caractères

```c
void hello(const char *name)
{
    printf("Bonjour %s\n", name);
}
```

Passage d'une chaîne de caractères

```c
void hello(const char *name)
{
    printf("Bonjour %s\n", name);
}
```

Code Python

```python
lib.hello.argtypes = [ctypes.c_char_p]
lib.hello.restype = None

lib.hello(b"Achref")
```

Passage d'une chaîne de caractères

```c
void hello(const char *name)
{
    printf("Bonjour %s\n", name);
}
```

Code Python

```python
lib.hello.argtypes = [ctypes.c_char_p]
lib.hello.restype = None

lib.hello(b"Achref")
```


Attention

c_char_p attend une chaîne d'octets, d'où le préfixe b.

Appel d'une fonction C

Passage d'un pointeur

```
void incrementer(int *n)
{
    (*n)++;
}
```

Appel d'une fonction C

Passage d'un pointeur

```
void incrementer(int *n)
{
    (*n)++;
}
```

Code Python

```
valeur = ctypes.c_int(10)

lib.incrementer.argtypes = [ctypes.POINTER(ctypes.c_int)]
lib.incrementer.restype = None

lib.incrementer(ctypes.byref(valeur))
print(valeur.value)
```

Appel d'une fonction C

Passage d'un tableau

```c
int somme(int *tab, int taille)
{
    int s = 0;
    for(int i = 0; i < taille; i++)
        s += tab[i];
    return s;
}
```

Appel d'une fonction C

Passage d'un tableau

```c
int somme(int *tab, int taille)
{
    int s = 0;
    for(int i = 0; i < taille; i++)
        s += tab[i];
    return s;
}
```


Idée côté Python

Il faut créer un tableau compatible **C** à l'aide de `ctypes`.

Appel d'une fonction C

Créer un tableau C avec ctypes

```
TableauInt = ctypes.c_int * 5
valeurs = TableauInt(1, 2, 3, 4, 5)

lib.somme.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]
lib.somme.restype = ctypes.c_int

print(lib.somme(valeurs, 5))
```

Appel d'une fonction C

Créer un tableau C avec ctypes

```
TableauInt = ctypes.c_int * 5
valeurs = TableauInt(1, 2, 3, 4, 5)

lib.somme.argtypes = [ctypes.POINTER(ctypes.c_int), ctypes.c_int]
lib.somme.restype = ctypes.c_int

print(lib.somme(valeurs, 5))
```


Résultat

```
15
```

Appel d'une fonction C

Exercice : bibliothèque de calcul

- Créer un fichier `utils.c`.
- Ajouter les fonctions `add`, `sub`, `mul`, `diviser`.
- Compiler le fichier en bibliothèque partagée.
- Charger la bibliothèque avec `ctypes`.
- Appeler chaque fonction depuis **Python**.

Appel d'une fonction C

Exercice : bibliothèque de calcul

- Créer un fichier `utils.c`.
- Ajouter les fonctions `add`, `sub`, `mul`, `diviser`.
- Compiler le fichier en bibliothèque partagée.
- Charger la bibliothèque avec `ctypes`.
- Appeler chaque fonction depuis **Python**.

Point d'attention

Gérer correctement le cas de la division par zéro.

Appel d'une fonction C

Correction : fichier `utils.c`

```c
int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int mul(int a, int b)
{
    return a * b;
}
double diviser(double a, double b)
{
    if (b == 0)
        return 0;

    return a / b;
}
```

Appel d'une fonction C

Correction : utilisation avec `ctypes`

```
from ctypes import *

lib = CDLL("./libutils.so")

lib.add.argtypes = [c_int, c_int]
lib.add.restype = c_int

lib.diviser.argtypes = [c_double, c_double]
lib.diviser.restype = c_double

print(lib.add(10, 5))
print(lib.sub(10, 5))
print(lib.mul(10, 5))
print(lib.diviser(10, 5))
print(lib.diviser(10, 0))
```

Résultat

```
15
5
50
2.0
0.0
```

Problème classique : réécrire une fonction **Python** en **C**

Une fonction **Python** est claire et correcte, mais elle est appelée très souvent ou manipule un grand volume de données.

API Python/C

Problème classique : réécrire une fonction **Python** en **C**

Une fonction **Python** est claire et correcte, mais elle est appelée très souvent ou manipule un grand volume de données.

Solution possible

Réécrire cette fonction en **C** et l'exposer comme un module **Python**.

API Python/C

Problème classique : réécrire une fonction **Python** en **C**

Une fonction **Python** est claire et correcte, mais elle est appelée très souvent ou manipule un grand volume de données.

Solution possible

Réécrire cette fonction en **C** et l'exposer comme un module **Python**.

Attention

Cette solution doit être envisagée après mesure avec un profileur, pas au hasard.

Exemple de fonction Python à optimiser

```python
def somme_carres(n):
    s = 0
    for i in range(n):
        s += i * i
    return s
```

Exemple de fonction Python à optimiser

```python
def somme_carres(n):
    s = 0
    for i in range(n):
        s += i * i
    return s
```


Observation

Cette fonction est simple, mais la boucle peut devenir coûteuse pour de grandes valeurs de `n`.

API Python/C

Qu'est-ce que l'**API Python/C** ?

L'**API Python/C** est l'interface officielle permettant d'écrire du code **C** qui interagit directement avec l'interpréteur **Python**.

API Python/C

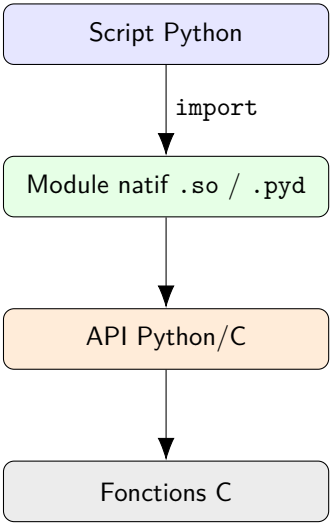
Qu'est-ce que l'**API Python/C** ?

L'**API Python/C** est l'interface officielle permettant d'écrire du code **C** qui interagit directement avec l'interpréteur **Python**.

Elle permet de :

- créer des modules **Python** écrits en **C**
- manipuler des objets **Python** depuis **C**
- convertir des valeurs **Python** vers des types **C**
- convertir des valeurs **C** vers des objets **Python**

API Python/C



Architecture d'une extension **C**

Le module compilé se comporte comme un module **Python** classique.

Structure d'une fonction C exposée à Python (à placer dans `cmaths.c`)

```c
static PyObject* ma_fonction(PyObject* self, PyObject* args)
{
    // corps de la fonction
}
```

Structure d'une fonction C exposée à Python (à placer dans `cmaths.c`)

```c
static PyObject* ma_fonction(PyObject* self, PyObject* args)
{
    // corps de la fonction
}
```

Explication

- Le retour est un `PyObject*`.
- Les arguments **Python** sont reçus dans `args`.
- Il faut analyser les arguments avant de les utiliser en **C**.

API Python/C

Inclure Python.h (à placer dans `cmaths.c`)

```c
#include <Python.h>
```

Inclure Python.h (à placer dans `cmaths.c`)

```
#include <Python.h>
```


Rôle de `Python.h`

- Donne accès aux types de l'API **Python/C**.
- Donne accès aux fonctions de conversion.
- Permet de créer et initialiser des modules **Python**.

API Python/C

Analyser les arguments avec PyArg_ParseTuple

```c
int n;

if (!PyArg_ParseTuple(args, "i", &n))
{
    return NULL;
}
```

API Python/C

Analyser les arguments avec PyArg_ParseTuple

```c
int n;

if (!PyArg_ParseTuple(args, "i", &n))
{
    return NULL;
}
```


Signification

- `"i"` indique que l'on attend un entier.
- `&n` reçoit la valeur convertie côté **C**.
- `return NULL` indique une erreur à **Python**.

Quelques formats de PyArg_ParseTuple

Format	Signification
`i`	entier C `int`
`l`	entier C `long`
`d`	nombre réel `double`
`s`	chaîne C `char *`
`O`	objet Python générique

Quelques formats de PyArg_ParseTuple

Format	Signification
`i`	entier C `int`
`l`	entier C `long`
`d`	nombre réel `double`
`s`	chaîne C `char *`
`O`	objet Python générique

Remarque

Les chaînes de format sont très importantes : une erreur peut provoquer un comportement incorrect.

Créer un objet Python en retour

```
return PyLong_FromLong(resultat);
```

API Python/C

Créer un objet Python en retour

```
return PyLong_FromLong(resultat);
```

Retourner un réel

```
return PyFloat_FromDouble(resultat);
```

API Python/C

Créer un objet Python en retour

```
return PyLong_FromLong(resultat);
```

Retourner un réel

```
return PyFloat_FromDouble(resultat);
```


Idée essentielle

Une fonction exposée à **Python** doit retourner un objet **Python**, même si le calcul interne est fait avec des types **C**.

API Python/C

Exemple : somme des carrés en C (à placer dans `cmaths.c`)

```c
static PyObject* somme_carres(PyObject* self, PyObject* args)
{
    int n;
    long s = 0;

    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;

    for (int i = 0; i < n; i++)
        s += i * i;

    return PyLong_FromLong(s);
}
```

API Python/C

Déclarer les méthodes du module avec `PyMethodDef` (à placer dans `cmaths.c`)

```
static PyMethodDef Methodes[] = {
    {"somme_carres", somme_carres, METH_VARARGS, "Calcule la somme des carres."},
     {NULL, NULL, 0, NULL}
};
```

API Python/C

Déclarer les méthodes du module avec `PyMethodDef` (à placer dans `cmaths.c`)

```c
static PyMethodDef Methodes[] = {
    {"somme_carres", somme_carres, METH_VARARGS, "Calcule la somme des carres."},
     {NULL, NULL, 0, NULL}
};
```


Rôle

Cette table associe les noms visibles depuis Python aux fonctions C correspondantes.

Structure `PyModuleDef` (à placer dans `cmaths.c`)

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "cmaths",
    "Module C d'exemple",
    -1,
    Methodes
};
```

Structure `PyModuleDef` (à placer dans `cmaths.c`)

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "cmaths",
    "Module C d'exemple",
    -1,
    Methodes
};
```


Nom du module

Le nom `cmaths` sera utilisé ensuite dans Python avec `import cmaths`.

API Python/C

Initialiser le module (à placer dans `cmaths.c`)

```c
PyMODINIT_FUNC PyInit_cmaths(void)
{
    return PyModule_Create(&moduledef);
}
```

API Python/C

Initialiser le module (à placer dans `cmaths.c`)

```c
PyMODINIT_FUNC PyInit_cmaths(void)
{
    return PyModule_Create(&moduledef);
}
```


Attention

Le nom de la fonction doit respecter la forme `PyInit_nomdumodule`.

API Python/C

Fichier `setup.py`

```
from setuptools import setup, Extension

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

setup(
    name="cmaths",
    version="1.0",
    ext_modules=[module]
)
```

Commande de compilation

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

API Python/C

Commande de compilation

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

Résultat

Cette commande génère un fichier compilé importable depuis Python, par exemple :

`cmaths.cpython-312-x86_64-linux-gnu.so`

Utiliser le module C depuis Python

```
import cmaths

print(cmaths.somme_carres(1000000))
```

Utiliser le module C depuis Python

```
import cmaths

print(cmaths.somme_carres(1000000))
```


Intérêt

Du point de vue **Python**, le module natif s'utilise presque comme un module classique.

Avantages de l'**API Python/C**

- Très performant.
- Interface officielle de CPython.
- Permet un contrôle très fin.

API Python/C

Avantages de l'**API Python/C**

- Très performant.
- Interface officielle de CPython.
- Permet un contrôle très fin.

Inconvénients de l'**API Python/C**

- Plus complexe que du Python classique.
- Gestion des références à maîtriser.
- Risque de bugs mémoire.

Mini-exercice : module **C**

- Écrire une fonction **Python** `factorielle(n)`.
- Réécrire la fonction en **C** avec l'**API Python/C**.
- Compiler le module.
- Importer le module depuis **Python**.
- Comparer les temps d'exécution.

Créer des modules **C** avec **Pyrex**

Pyrex est un langage permettant d'écrire des modules d'extension **Python** en utilisant une syntaxe proche de **Python**, puis de générer du code **C**.

Pyrex et Cython

Créer des modules **C** avec **Pyrex**

Pyrex est un langage permettant d'écrire des modules d'extension **Python** en utilisant une syntaxe proche de **Python**, puis de générer du code **C**.

Idée pédagogique

Au lieu d'écrire directement toute l'**API Python/C**, on écrit un code plus proche de **Python**, qui est ensuite traduit en **C**.

Pyrex et Cython

Créer des modules **C** avec **Pyrex**

Pyrex est un langage permettant d'écrire des modules d'extension **Python** en utilisant une syntaxe proche de **Python**, puis de générer du code **C**.

Idée pédagogique

Au lieu d'écrire directement toute l'**API Python/C**, on écrit un code plus proche de **Python**, qui est ensuite traduit en **C**.

Remarque historique

Aujourd'hui, **Cython** est beaucoup plus utilisé que **Pyrex**. **Cython** peut être vu comme un successeur moderne et enrichi.

Pyrex vs Cython

Critère	Pyrex	Cython
Syntaxe	Proche Python	Proche Python + extensions
Usage actuel	Historique	Très utilisé
Types C	Oui	Oui, plus riche
Écosystème	Limité	Mature
Performance	Bonne	Très bonne

Pyrex et Cython

Premier exemple Cython (fichier `cmodule.pyx`)

```
def add(int a, int b):
    return a + b
```

Pyrex et Cython

Premier exemple Cython (fichier `cmodule.pyx`)

```
def add(int a, int b):
    return a + b
```

Observation

Le code ressemble beaucoup à du **Python**, mais les paramètres sont typés comme en **C**.

Variables typées en Cython

```
def somme_carres(int n):
    cdef int i
    cdef long s = 0

    for i in range(n):
        s += i * i

    return s
```

Variables typées en Cython

```
def somme_carres(int n):
    cdef int i
    cdef long s = 0

    for i in range(n):
        s += i * i

    return s
```


Mot-clé cdef

`cdef` permet de déclarer des variables **C** dans le code **Cython**.

Compiler un module Cython (fichier `setup.py`)

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

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

Compiler un module Cython (fichier `setup.py`)

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

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


Installez `Cython`

```
pip install cython
```

Compiler un module Cython (fichier `setup.py`)

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

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


Installez `Cython`

```
pip install cython
```

Puis exécutez

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

Utiliser le module généré

```
import cmodule

print(cmodule.add(2, 3))
print(cmodule.somme_carres(1000000))
```

Utiliser le module généré

```
import cmodule

print(cmodule.add(2, 3))
print(cmodule.somme_carres(1000000))
```


Avantage

Le module compilé s'utilise ensuite comme un module **Python** classique.

Quand utiliser **Cython** ?

Cython est intéressant quand :

- une fonction Python est identifiée comme lente
- le code contient beaucoup de boucles numériques
- on veut garder une syntaxe proche de **Python**
- on veut éviter d'écrire directement toute l'**API Python/C**.

Quand utiliser **Cython** ?

Cython est intéressant quand :

- une fonction Python est identifiée comme lente
- le code contient beaucoup de boucles numériques
- on veut garder une syntaxe proche de **Python**
- on veut éviter d'écrire directement toute l'**API Python/C**.

Attention

Cython n'accélère pas automatiquement tout le code **Python** : le typage et la structure du code restent importants.

Exercice : version Cython

- Reprendre la fonction **Python** `somme_carres`.
- Créer une version **Cython** dans un fichier `.pyx`.
- Ajouter des types C avec `cdef`.
- Compiler le module.
- Comparer avec la version **Python** pure.

L'interpréteur Python dans C

Intégrer **Python** dans un programme **C**

L'embedding consiste à intégrer l'interpréteur **Python** dans une application écrite en **C**.

L'interpréteur Python dans C

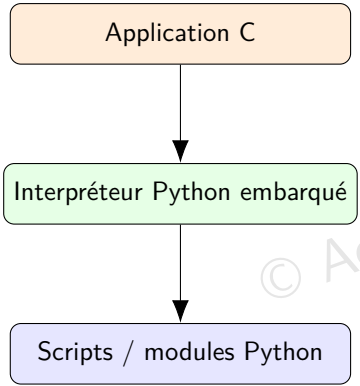
Intégrer **Python** dans un programme **C**

L'embedding consiste à intégrer l'interpréteur **Python** dans une application écrite en **C**.

Différence avec une extension

- Extension C : **Python** appelle du **C**.
- Embedding : **C** lance et utilise **Python**.

L'interpréteur Python dans C



Architecture de l'embedding

L'application **C** peut exécuter du code **Python** pour ajouter des fonctionnalités dynamiques ou scriptables.

L'interpréteur Python dans C

Cas d'utilisation de l'embedding

- Logiciels qui permettent d'écrire des scripts d'automatisation.
- Moteurs de jeux utilisant **Python** comme langage de script.
- Applications scientifiques extensibles.
- Outils industriels nécessitant des scripts personnalisés.

L'interpréteur Python dans C

Cas d'utilisation de l'embedding

- Logiciels qui permettent d'écrire des scripts d'automatisation.
- Moteurs de jeux utilisant **Python** comme langage de script.
- Applications scientifiques extensibles.
- Outils industriels nécessitant des scripts personnalisés.

Idée importante

Python devient ici un langage embarqué dans une application hôte.

L'interpréteur Python dans C

Initialiser l'interpréteur Python

```c
#include <Python.h>

int main()
{
    Py_Initialize();

    // utilisation de Python

    Py_Finalize();
    return 0;
}
```

L'interpréteur Python dans C

Exécuter une instruction Python depuis C

```c
#include <Python.h>

int main()
{
    Py_Initialize();

    PyRun_SimpleString("print('Bonjour depuis Python')");

    Py_Finalize();
    return 0;
}
```

Exécuter plusieurs instructions

```
PyRun_SimpleString(
    "x = 10\n"
    "y = 20\n"
    "print('Somme =', x + y)\n"
);
```

Exécuter plusieurs instructions

```
PyRun_SimpleString(
    "x = 10\n"
    "y = 20\n"
    "print('Somme =', x + y)\n"
);
```


Remarque

Les retours à la ligne `\n` sont nécessaires pour séparer les instructions **Python**.

L'interpréteur Python dans C

Compiler un programme **C** qui embarque **Python**

Il faut compiler le programme **C** en le liant avec les bibliothèques **Python**.

L'interpréteur Python dans C

Compiler un programme **C** qui embarque **Python**

Il faut compiler le programme **C** en le liant avec les bibliothèques **Python**.

Commencer par exécuter la commande suivante pour récupérer le chemin d'installation de Python

```
python -c "import sysconfig; print(sysconfig.get_paths()['include'])"
```

L'interpréteur Python dans C

Compiler un programme **C** qui embarque **Python**

Il faut compiler le programme **C** en le liant avec les bibliothèques **Python**.

Commencer par exécuter la commande suivante pour récupérer le chemin d'installation de Python

```
python -c "import sysconfig; print(sysconfig.get_paths()['include'])"
```

Puis

```
python -c "import sysconfig; print(sysconfig.get_paths()['libs'])"
```

L'interpréteur Python dans C

Pour compiler

```
gcc test.c -I"Chemin\vers\Include" -L"Chemin\vers\libs" -lpython312 -o
test.exe
```

L'interpréteur Python dans C

Pour compiler

```
gcc test.c -I"Chemin\vers\Include" -L"Chemin\vers\libs" -lpython312 -o
test.exe
```

Et pour exécuter

```
.\test.exe
```

L'interpréteur Python dans C

Et si jamais ça ne marche pas (Sous Power Shell)

```
$env:PATH = "C:\Users\pyenv-win\versions\3.12.3;$env:PATH"
```

Et si jamais ça ne marche pas (Sous Power Shell)

```
$env:PATH = "C:\Users\pyenv-win\versions\3.12.3;$env:PATH"
```

Ou (Sous Invite de commandes)

```
set PATH=C:\Users\pyenv-win\versions\3.12.3;%PATH%
```

Comparaison : extension vs embedding

Critère	Extension C	Embedding Python
Programme principal	Python	C
Direction	Python appelle C	C appelle Python
Objectif	Accélérer / étendre Python	Rendre une app scriptable
Exemple	Module natif	Logiciel avec scripts Python

Utilisation du profileur de code

Pourquoi profiler le code ?

Avant d'optimiser un programme, il faut mesurer précisément où le temps est réellement consommé.

Utilisation du profileur de code

Pourquoi profiler le code ?

Avant d'optimiser un programme, il faut mesurer précisément où le temps est réellement consommé.

Le profilage permet de :

- repérer les fonctions les plus coûteuses
- identifier les boucles ou appels répétés
- comparer plusieurs versions d'un même code
- éviter les optimisations inutiles

Utilisation du profileur de code

Le module `cProfile`

`cProfile` est un profileur fourni avec **Python**.

Utilisation du profileur de code

Le module cProfile

cProfile est un profileur fourni avec **Python**.

Utilisation simple

```
import cProfile

cProfile.run("main()")
```

Utilisation du profileur de code

Le module cProfile

cProfile est un profileur fourni avec **Python**.

Utilisation simple

```
import cProfile

cProfile.run("main()")
```

Remarque

cProfile mesure le temps passé dans les fonctions et le nombre d'appels.

Utilisation du profileur de code

Exemple de programme à profiler

```
def somme_carres(n):
    s = 0
    for i in range(n):
        s += i * i
    return s


def main():
    for _ in range(10):
        somme_carres(100000)
```

Utilisation du profileur de code

Profilage depuis le terminal

```
python -m cProfile programme.py
```

Utilisation du profileur de code

Profilage depuis le terminal

```
python -m cProfile programme.py
```

Tri des résultats

```
python -m cProfile -s cumulative programme.py
```

Utilisation du profileur de code

Profilage depuis le terminal

```
python -m cProfile programme.py
```

Tri des résultats

```
python -m cProfile -s cumulative programme.py
```


Intérêt

Le tri par temps cumulé aide à repérer les fonctions qui consomment le plus globalement.

Utilisation du profileur de code

Lire les résultats de `cProfile` : colonnes principales

Colonne	Signification
`ncalls`	nombre d'appels
`tottime`	temps passé dans la fonction seule
`percall`	temps moyen par appel
`cumtime`	temps cumulé avec les appels internes
`filename:lineno`	emplacement de la fonction

Utilisation du profileur de code

Temps propre : `tottime`

Temps passé directement dans la fonction, sans compter le temps passé dans les fonctions qu'elle appelle.

Utilisation du profileur de code

Temps propre : `tottime`

Temps passé directement dans la fonction, sans compter le temps passé dans les fonctions qu'elle appelle.

Temps cumulé : `cumtime`

Temps total passé dans la fonction, en incluant les fonctions appelées à l'intérieur.

Utilisation du profileur de code

Temps propre : `tottime`

Temps passé directement dans la fonction, sans compter le temps passé dans les fonctions qu'elle appelle.

Temps cumulé : `cumtime`

Temps total passé dans la fonction, en incluant les fonctions appelées à l'intérieur.

Conseil

Pour comprendre le coût global d'une fonction, regarder souvent `cumtime`. Pour identifier un corps de fonction lent, regarder `tottime`.

Utilisation du profileur de code

Exploiter les résultats avec pstats : sauvegarder les statistiques

```
python -m cProfile -o stats.prof programme.py
```

Utilisation du profileur de code

Exploiter les résultats avec pstats : sauvegarder les statistiques

```
python -m cProfile -o stats.prof programme.py
```


Lire avec Python

```python
import pstats

stats = pstats.Stats("stats.prof")
stats.sort_stats("cumulative")
stats.print_stats(10)
```

Visualiser les résultats : outils possibles

- `pstats` pour une analyse textuelle.
- `snakeviz` pour une visualisation graphique.
- `gprof2dot` pour générer des graphes d'appels.

Visualiser les résultats : outils possibles

- `pstats` pour une analyse textuelle.
- `snakeviz` pour une visualisation graphique.
- `gprof2dot` pour générer des graphes d'appels.

Commande avec SnakeViz

```
snakeviz stats.prof
```

Visualiser les résultats : outils possibles

- `pstats` pour une analyse textuelle.
- `snakeviz` pour une visualisation graphique.
- `gprof2dot` pour générer des graphes d'appels.

Commande avec SnakeViz

```
snakeviz stats.prof
```

N'oublions pas d'installer SnakeViz

```
pip install snakeviz
```

Utilisation du profileur de code

Méthodologie d'optimisation

- Écrire une version **Python** claire et correcte.
- Écrire des tests pour vérifier le comportement.
- Mesurer avec un profileur.
- Identifier les vraies fonctions coûteuses.
- Optimiser localement.
- Comparer les performances avant / après.

Utilisation du profileur de code

Quand passer au **C** ?

Le passage au **C** peut être pertinent si :

- la fonction est réellement un goulot d'étranglement ;
- l'algorithme Python est déjà raisonnablement optimisé ;
- le coût de conversion Python/C ne dépasse pas le gain attendu ;
- la complexité supplémentaire est acceptable.

Utilisation du profileur de code

Quand passer au **C** ?

Le passage au **C** peut être pertinent si :

- la fonction est réellement un goulot d'étranglement ;
- l'algorithme Python est déjà raisonnablement optimisé ;
- le coût de conversion Python/C ne dépasse pas le gain attendu ;
- la complexité supplémentaire est acceptable.

À retenir

Optimiser sans mesurer conduit souvent à complexifier inutilement le code.

TP : accélération progressive d'une fonction **Python**

Comparer plusieurs façons d'optimiser une fonction **Python** simple en utilisant le profilage, `ctypes`, l'**API Python/C** et **Cython**.

Étape 1 : version **Python** pure

- Écrire une fonction `somme_carres(n)` en **Python**.
- Ajouter un programme principal qui appelle la fonction plusieurs fois.
- Vérifier le résultat obtenu pour de petites valeurs.
- Profiler le programme avec `cProfile`.

Étape 2 : version **C** appelée avec `ctypes`

- Écrire la fonction équivalente en **C**.
- Compiler le code en bibliothèque partagée.
- Charger la bibliothèque avec `ctypes`.
- Déclarer correctement `argtypes` et `restype`.
- Comparer les temps d'exécution.

Étape 3 : module **C** avec Python/C API

- Créer un fichier **C** utilisant `Python.h`.
- Écrire une fonction exposée à **Python**.
- Déclarer les méthodes du module.
- Compiler avec `setuptools`.
- Importer le module et tester la fonction.

Étape 4 : version Cython

- Créer un fichier `.pyx`.
- Ajouter des types **C** avec `cdef`.
- Compiler le module avec `cythonize`.
- Comparer la lisibilité et la performance.

TP et synthèse

Questions de synthèse

- Quelle différence entre `ctypes` et une extension **C** ?
- Pourquoi faut-il déclarer `argtypes` et `restype` ?
- Pourquoi une fonction **C** exposée à Python retourne-t-elle un `PyObject*` ?
- Dans quel cas **Cython** est-il plus simple que l'**API Python/C** ?
- Pourquoi faut-il profiler avant d'optimiser ?

Synthèse du chapitre

- **Python** peut appeler du code **C** grâce à différents mécanismes.
- `ctypes` permet de charger et d'utiliser des bibliothèques partagées.
- L'**API Python/C** permet d'écrire des modules natifs performants.
- Pyrex et surtout **Cython** simplifient la création de modules compilés.
- Le langage **C** peut aussi embarquer l'interpréteur **Python**.
- Le profilage est indispensable pour optimiser intelligemment.

TP et synthèse

Ce qu'il faut retenir

- **Python** et **C** sont complémentaires.
- Le **C** apporte de la performance, mais aussi de la complexité.
- `ctypes` est utile pour appeler rapidement une bibliothèque existante.
- L'**API Python/C** est puissante, mais plus difficile à maîtriser.
- **Cython** représente souvent un bon compromis entre lisibilité et performance.
- On optimise toujours après avoir mesuré.

TP et synthèse

Pour aller plus loin

- Étudier plus en détail la gestion des références dans l'**API Python/C**.
- Manipuler des structures **C** avec `ctypes.Structure`.
- Utiliser **Cython** avec **NumPy**.
- Comparer `ctypes`, `cffi`, **Cython** et extensions **C** natives.
- Étudier les implications du GIL dans le code natif.