

# L'interface C de CPython et de Numpy



par Matthieu Brucher (<http://matthieu-brucher.developpez.com/>) (Blog)

Date de publication : 24/01/2008

Dernière mise à jour :

L'interface C de CPython permet d'interfacer de manière basique les langages C ou C++ avec Python. L'interface pour Numpy est aussi introduite.  
Ce texte est issu du livre **Python - les fondamentaux du langage - la programmation pour les scientifiques** aux éditions ENI.

## Introduction

### I - L'interface C de l'interpréteur CPython

I-A - Utilisation élémentaire

I-B - Utilisation avancée et compteur de références

I-C - Quelques objets et fonctions utiles

I-D - Notes au sujet du parallélisme

### II - Introduction à l'interface C de Numpy

## Conclusion

## Introduction

L'interface C intervient souvent lors de la construction de différentes interfaces même si un grand nombre de fonctions dédiées comme la conversion entre les types Python et les types C ou C++ seront cachées du programmeur (cela sera présenté dans un prochain tutoriel).

Lors de la sortie de Python 3.0, l'interface C va être modifiée tant pour Python que pour Numpy. L'introduction qui en est faite ne sera plus forcément valable mais l'esprit restera.

## I - L'interface C de l'interpréteur CPython

Tout objet Python est une variable de type **PyObject\***. Cette structure est déclarée dans l'entête **Python.h** qui est toujours inclus avant autre chose, y compris les bibliothèques systèmes (un certain nombre de drapeaux sont placés modifiant le comportement de certaines fonctionnalités sur certains systèmes).

### I-A - Utilisation élémentaire

Toute fonction utilisable dans Python est de la forme **PyObject\* ma\_fonction(PyObject\* self, PyObject\* args)**. Le premier argument est facultatif et représente l'instance de la classe sur laquelle une méthode est appelée. Dans le cas d'une fonction, seul le second argument est utile. Celui-ci contient un tuple contenant les arguments passés à la fonction.

La récupération des arguments se réalise traditionnellement par la fonction **PyArg\_ParseTuple()**. Le premier argument est le tuple contenant les arguments, puis une chaîne de format et des pointeurs vers les données à remplir. Le retour de cette fonction est différent de 0 si tout s'est bien passé.


#### Exemple simple :

```
static PyObject* exemple(PyObject* self, PyObject* args)
{
    const char* chaine;

    if(!PyArg_ParseTuple(args, "s", &chaine))
        return NULL;


    return Py_BuildValue("i", 0);
}
```

Dans cet exemple, la fonction **exemple()** prend en argument une seule valeur qui est une chaîne de caractères. Cette chaîne de caractères est remplie par **PyArg\_ParseTuple()** à qui il est précisé qu'il n'y a qu'un seul argument de type 's' ou chaîne de caractères. L'opération inverse est effectuée par **Py\_BuildValue()** qui crée un objet Python à partir d'une chaîne de format et plusieurs arguments, ici un entier est créé.


 *Python ne vérifie pas si le pointeur donné est effectivement un pointeur vers un entier lorsque la chaîne de format indique qu'un entier doit être lu. D'autres exemples de chaînes de format sont "f|i" qui lira un flottant et peut-être un entier, "[fs]" qui lira ou créera une liste contenant un flottant et une chaîne de caractères.*

La valeur de retour de **PyArg\_ParseTuple()** est testée et si elle est nulle, un pointeur nul sera retourné à la fonction appelante. En fait, cette fonction a levé une exception Python qui sera propagée par ce pointeur nul. De même, si **Py\_BuildValue()** échoue, une exception Python est levée et est propagée à la fonction appelante. Si une exception est levée dans une fonction fille, il ne faut surtout pas lever une nouvelle exception mais simplement la propager en retournant un pointeur nul. Arrivé dans le code Python appelant, l'utilisateur pourra décider de l'action à mener.

Lever une exception se fait par la fonction **PyErr\_SetString()**, prenant en paramètre un type d'erreur et une chaîne de commentaire. Heureusement, plusieurs fonctions d'aide existent pour simplifier les appels, comme **PyExc\_ZeroDivisionError()** qui lève une exception indiquant une division par zéro, **PyErr\_NoMemory()** levée en cas d'erreur d'allocation ou encore **PyErr\_BadArgument()** qui indique une erreur de type (c'est l'exception levée par **PyArg\_ParseTuple()** en cas d'erreur de conversion). Si une fonction C est capable de gérer une erreur (détectée en vérifiant le retour d'une fonction appelée), elle peut effacer l'exception en appelant **PyErr\_Clear()**.

 D'autres exceptions peuvent être levées, elles ont un nom de fonction commençant par **PyExc\_\***.

Une fois les fonctions créées, elles ne sont pas visibles pour le moment par Python. En effet, dans l'exemple donné, le mot-clé **static** indique expressément que la fonction ne sera pas visible de l'extérieur et donc a fortiori par Python. Une seule fonction doit être visible de l'extérieur, il s'agit de **init\_module()** où **\_module** est le nom du module qui devra être chargé. Cette fonction a pour mission de créer un **PyObject\*** et de le retourner, cet objet contenant la liste des fonctions disponibles dans le module. En fait, ceci sera effectué par un appel à **Py\_InitModule()** qui prend en argument le nom du module ainsi qu'un tableau de **PyMethodDef**. Cette structure **PyMethodDef** contient quant à elle le nom de la fonction, le pointeur vers cette fonction, quel est le mode de passage des arguments (**METH\_VARARGS** pour passer les arguments dans un tuple) et une chaîne de commentaire sur la fonction.


 La fonction d'initialisation est déclarée avec **PyMODINIT\_FUNC** qui dit que la fonction ne renvoie rien (**void**) et donne aussi plusieurs informations au compilateur utilisé pour que cette fonction soit visible de l'extérieur.

#### Exemple de déclaration de la fonction précédente

```
static PyMethodDef fonctions[]={
    {"exemple", exemple, METH_VARARGS, "Un commentaire"},
    {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC init_mon_module(void)
{
    PyObject* m;
    m = Py_InitModule("_mon_module", fonctions);
}
```

L'objet module **m** est automatiquement enregistré dans l'interpréteur, mais peut être utilisé pour ajouter manuellement d'autres objets. Par exemple une nouvelle exception **ExempleErreur** (déclarée comme variable statique dans le fichier source) pourra être déclarée avec **PyErr\_NewException()**, le premier argument étant le nom de l'exception (avec le nom du module donc **\_mon\_module.Exception**), le deuxième donne le type dont l'exception héritera et le troisième est un dictionnaire donnant une liste de méthodes (en général, ces deux arguments sont placés à **NULL** et l'exception héritera de la classe de base **Exception**). Elle doit ensuite être enregistrée dans le module à l'aide de la fonction **PyModule\_AddObject()** prenant en paramètres le module où inclure l'objet, le nom de cet objet dans le module et l'objet lui-même. Une fois l'exception créée et enregistrée, elle pourra être utilisée avec **PyErr\_SetString(ExempleErreur, "Une explication")**.

 Ainsi créée, l'exception peut être effacée par un code externe, ce qui pourrait causer des problèmes lors de l'exécution du code. Il faut explicitement dire que la variable **ExempleErreur** possède une référence sur l'exception afin qu'elle ne soit pas détruite à un moment donné si la variable présente dans le module est détruite.

Voici le code complet de cet exemple :

#### api\_c.c

```
#include <Python.h>

/**
 * Une fois importe, le module définit deux fonctions :
 * - exemple qui prend une chaîne en argument et retourne 0
 * - exemple2 qui lève une exception créée pour l'occasion
 */

static PyObject* ExempleErreur;
```

## api\_c.c

```
static PyObject* exemple(PyObject* self, PyObject* args)
{
    const char* chaine;

    if(!PyArg_ParseTuple(args, "s", &chaine))
        return NULL;

    return Py_BuildValue("i", 0);
}

static PyObject* exemple2(PyObject* self, PyObject* args)
{
    const char* chaine;

    PyErr_SetString(ExempleErreur, "Exemple de levée d'erreur");
    return NULL;
}

static PyMethodDef fonctions[]={
    {"exemple", exemple, METH_VARARGS, "Un commentaire"},
    {"exemple2", exemple2, METH_VARARGS, "Une methode levant une exception"},
    {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC init_mon_module(void)
{
    PyObject* m;
    m = Py_InitModule("_mon_module", fonctions);

    ExempleErreur = PyErr_NewException("_mon_module.Exception", NULL, NULL);
    Py_INCREF(ExempleErreur);
    PyModule_AddObject(m, "Exception", ExempleErreur);
}
```

## setup.py pour compiler le module

```
#!/usr/bin/python
# -*- coding: cp1252 -*-

from distutils.core import setup
from distutils.core import Extension

setup(name = 'exemples',
      version = '1.0',
      ext_modules = [Extension('_mon_module', ['api_c.c']), ],
      cmdclass = cmdclass,
      url='http://matthieu-brucher.developpez.com/livre/python/scientifiques',
      author='Matthieu Brucher',
      author_email='@',
      )
```


## Compilation du module sur place

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

## utilisation du module

```
import _mon_module


print _mon_module.exemple()
print _mon_module.exemple2()
```

 *Le compilateur utilisé pour compiler le module d'extension doit être de préférence le compilateur utilisé pour compiler Python. Cela est particulièrement le cas pour Visual Studio où le compilateur utilisé doit être Visual Studio 2003 (la version 2005 ou 2008 n'est pas acceptée).*

## I-B - Utilisation avancée et compteur de références


Une fois ces éléments de base maîtrisés, d'autres concepts peuvent être présentés.

Le premier consiste à comprendre le mécanisme de références utilisé par Python pour gérer l'utilisation de la mémoire. A chaque fois qu'un objet Python est créé, un compteur de référence y est attaché, comptant le nombre de variables référençant l'objet. Une fois ce compte nul, l'objet est détruit. Il est donc nécessaire d'indiquer explicitement si on veut garder l'objet ou non. On parle alors d'acquisition de l'objet.

 *En réalité, le Garbage Collector est capable de détecter si un cycle d'objets (des objets se référençant les uns les autres) peut être effacé sauf si l'un des éléments de ce cycle définit une méthode `__del__()` particulière, auquel cas l'objet sera visible dans `gc.garbage`.*

Pour compliquer les choses, lorsqu'une fonction Python est appelée, elle emprunte les objets Python passés en paramètre (par exemple pour la fonction `exemple()`, aucune incrémentation du compteur de référence n'a été faite, en effet, l'objet est acquis par l'une des fonctions parentes et donc ne sera pas détruit), elle ne les acquiert pas. Pour les acquérir explicitement, les fonctions `Py_INCREF()` et `Py_XINCRREF()` doivent être appelées (la seconde permet de passer un pointeur nul, ce n'est pas le cas de la première) et pour les relâcher, les fonctions réciproques `Py_DECREF()` et `Py_XDECREF()` sont à utiliser. Lorsqu'une fonction retourne un objet, l'acquisition de l'objet passe à la fonction appelante. C'est pourquoi la fonction `Py_BuildValue()` n'est pas suivie d'un appel à `Py_INCREF()` puisque c'est la fonction qui s'en charge elle-même lors de l'appel. En revanche, si un objet est lu depuis une liste ou toute autre classe (par exemple un attribut d'une classe), il est prudent d'acquérir l'objet lu et de le relâcher explicitement. En effet, lors d'opérations annexes, le compteur de référence de l'objet lu peut descendre à 0 et l'objet sera détruit alors qu'on veut travailler dessus.

Pour résumer : une fonction appelée emprunte chaque objet passé en paramètre et une fonction retournant un objet créé passe son acquisition à la fonction appelante.

 *Il y a malheureusement quelques exceptions à cette règle. La première est que même si les fonctions extrayant des attributs retournent un objet et l'acquisition de cet objet, ce n'est pas le cas de fonctions telles que `PyList_GetItem()`, `PyTuple_GetItem()` et `PyDict_GetItem()` qui retournent des objets empruntés, tout comme la fonction `PyImport_AddModule()`. D'autres fonctions prennent l'acquisition d'une fonction qui les appelle, comme par exemple `PyTuple_SetItem()` ou `PyList_SetItem()`. Pour ne pas avoir de problème avec ces fonctions, il est nécessaire de vérifier dans la documentation ce qu'il en est de ces acquisitions ou emprunts d'objets. Une valeur de retour indiquée comme *new reference* est passée comme *acquise* et si elle est indiquée comme *borrowed reference*, il s'agit d'une référence empruntée.*


Les fonctions définies peuvent aussi passer les arguments par dictionnaire. Dans ce cas, un troisième argument est fourni qui est le dictionnaire des arguments utilisés. Pour remplir les arguments C à partir de ces deux objets Python, une autre fonction doit être utilisée, `PyArg_ParseTupleAndKeywords()`, prenant ces deux objets en paramètres, une chaîne de format (toujours pour indiquer le ou les type(s) attendu(s)), une liste des arguments nommés attendus puis les pointeurs vers les variables.

Exemple d'un fonction prenant des arguments positionnels et formels et ne retournant rien :

### Exemple d'une fonction prenant des arguments positionnels et formels et ne retournant rien :

```
static PyObject* ma_fonction(PyObject* self, PyObject *args, PyObject *kwargs)
{
    int argument1, argument2;
    float argument3;
    static char* liste[] = {"argument1", "argument2", "argument3"};
    if(!PyArg_ParseTupleAndKeywords(args, kwargs, "iif", liste, &argument1, &argument2, &argument3);

    Py_INCREF(Py_None);
    return Py_None;
}
```

 Pour éviter de tester le type des données en C, le module créé peut être encapsulé dans un autre module Python. Par exemple le module **\_mon\_module** sera encapsulé dans **mon\_module** et chaque fonction du premier module fera l'objet d'une fonction identique dans le second module. Lors de l'appel à une fonction du module Python, les tests de type pourront être effectués et traités en cas d'erreur puis la fonction C appelée. C'est le même principe qui est appliqué lors de l'encapsulation d'une bibliothèque par CTypes, un module est créé pour gérer correctement les arguments passés en paramètre avant de les donner à la fonction C de la bibliothèque.

La fonction n'est plus du même type que les fonctions traditionnelles ne prenant qu'un seul argument. De plus, elle prend en argument un dictionnaire. Cela s'écrit sous la forme suivante dans le tableau des méthodes :

```
static PyMethodDef fonctions = {
    ...,
    {"ma_fonction", (PyCFunction)ma_fonction, METH_VARARGS|METH_KEYWORDS, "Un commentaire"}
};
```

## I-C - Quelques objets et fonctions utiles

Dans l'exemple précédent, un objet **Py\_None** a été utilisé. Il s'agit de l'objet Python **None**, et il est unique. Puisqu'on le retourne de la fonction, on transfère son acquisition. Pour cela, il faut donc l'acquiescer au préalable, d'où l'incrémention de son compteur.

Chaque fonction Python est préfixée par Py ou PY. Dans le cas de méthodes sur des types Python, comme les entiers, les listes ou les tuples, ce préfixe est suivi du nom de ce type (par exemple **PyList\_SetItem()** et **PyList\_GetItem()**). Ensuite se situe le nom de la méthode qui sera appelée. L'exposé des différentes méthodes n'est pas l'objet de cette partie, mais ce sont ces fonctions qui permettront de convertir un objet Python en une variable C associée ou inversement, transformer un objet Python en un autre, ajouter un objet Python à un autre, etc...

Afin de proposer des extensions concrètes à Python, une série de fonctions a vu le jour. L'objectif est de permettre à une extension de pouvoir être accessible par une autre extension (c'est typiquement le cas de l'extension Numpy qui est accédée par une extension créée pour optimiser certains calculs). Plutôt que de forcer tous les modules à exporter leurs fonctions et à ce que chaque module soit lié directement à l'interpréteur, une solution a été trouvée pour que les types et fonctions d'une extension dynamique puissent être passés à une autre extension dynamique, et c'est le rôle des objets **PyObject**. Cet objet contient un pointeur quelconque sur des données qui pourra ensuite être réutilisé.

Les fonctions intéressantes sont :

- **PyObject\* PyObject\_FromVoidPtr(void\* cobj, void (\*destr)(void\*))** qui crée un objet **PyObject** à partir du pointeur donné et qui utilisera la fonction passée en paramètre (si le pointeur n'est pas nul) lors de la destruction de l'objet

- **PyObject\* PyCObject\_FromVoidPtrAndDesc(void\* cobj, void\* desc, void (\*destr)(void \*, void \*))** qui crée un objet à partir d'un pointeur et d'une description et qui appellera la fonction **destr** lors de la destruction de l'objet avec les deux pointeurs passés en paramètre. L'utilisation classique de cette fonction avec numpy est de passer en second argument l'objet Python duquel une vue a été faite et pour permettre de relâcher cet objet lors de la destruction de la vue
- **void\* PyCObject\_AsVoidPtr(PyObject\* self)** qui récupère le pointeur stocké dans l'objet
- **void\* PyCObject\_GetDesc(PyObject\* self)** qui récupère la description de l'objet si elle existe

Pour utiliser des objets et les communiquer à un autre module, ils doivent tout d'abord être enregistrés. Pour cela, la fonction d'initialisation du module d'extension va enregistrer la totalité des fonctions accessibles dans un tableau de fonction appelé par exemple **PyMonModule\_API** de type **void\*\***. A partir de ce tableau, un objet de type **PyCObject** sera créé et enfin cet objet sera ajouté au module chargé avec un nom explicite.

```
void* PyMonModule_API[]={/*Ma liste de pointeurs vers une liste de fonctions*/};

PyMODINIT_FUNC init_mon_module(void)
{
    PyObject* m;
    PyObject* c_api;

    m = Py_InitModule("_mon_module", MesFonctions);

    c_api = PyCObject_FromVoidPtr((void *)PyMonModule_API, NULL);

    if (c_api != NULL)
        PyModule_AddObject(m, "c_api", c_api);
}
```

Ainsi, le module exporte un seul attribut qui sera un tableau de fonctions. Un nom plus explicite que **c\_api** pour cet attribut est souhaitable. En effet, comme il sera vu lors de l'utilisation de cet attribut, si tous les modules utilisent le même nom, il sera impossible de savoir si le module correct a été chargé, le test se faisant justement sur l'attribut chargé.


Voici le fichier d'entête simplifié qui doit être créé pour permettre de charger dans tout autre module d'extension les fonctions C mises à disposition par **\_mon\_module** :

```
static void **PyMonModule_API;

static int import_mon_module(void)
{
    PyObject* m = PyImport_ImportModule("_mon_module");

    if (m!=NULL)
    {
        PyObject* c_api = PyObject_GetAttrString(m, "c_api");
        if (c_api == NULL)
            return -1;
        if (PyCObject_Check(c_api))
            PyMonModule_API = (void **)PyObject_AsVoidPtr(c_api);
        Py_DECREF(c_api);
        Py_DECREF(m);
    }
    else
        return -1;
    return 0;
}
```

Cette fonction est a priori complexe, mais une fois assimilée, toutes les autres fonctions d'importation de fonctions d'un autre module d'extension sont connues. La première étape est d'importer le module que l'on souhaite charger. Une fois chargé, l'objet contenant la liste des fonctions de ce module est récupéré et acquis par **PyObject\_GetAttrString()**. Un test supplémentaire est effectué pour vérifier que l'objet est bien un **PyObject** et si c'est le cas, son contenu est récupéré et si la création du module a suivi le protocole précédent, le contenu est un pointeur vers un tableau de fonctions. Enfin, il faut relâcher l'objet récupéré dans le module ainsi que le module lui-même.

 *Un point négatif de cette approche est qu'il faut connaître le nom du tableau, ici **PyMonModule\_API**, et le numéro de chaque fonction à utiliser et son prototype. Ceci est à la charge de l'utilisateur de l'autre module. Il est donc judicieux de déclarer des macros avec des noms explicites comme **PyMonModule\_Fonction1()** qui vont se charger de récupérer le bon pointeur dans le tableau et de le transtyper dans le type original.*

*Une fonction **int ma\_fonction(float argument1, const char\* argument2)** étant la 3ème fonction enregistrée dans le tableau, la macro **#define PyMonModule\_MaFonction (\*(int (\*)(float, const char\*)) PyMonModule\_API[2])** permettra d'appeler directement cette fonction dans le code d'un module par **PyMonModule\_MaFonction(2, "un texte")**.*

## I-D - Notes au sujet du parallélisme

Par défaut, il n'est pas possible d'exécuter plusieurs fonctions dans Python en même temps sur plusieurs processeurs. Dans le cas où des accès longs d'entrée/sortie, Python propose des macros pour relâcher le GIL et permettre à un autre thread d'accéder à l'interpréteur, et lorsque les accès seront terminés, le GIL sera réacquis.


Voici les deux macros :

```
Py_BEGIN_ALLOW_THREADS
/* Effectuer des opérations extérieures à Python */
Py_END_ALLOW_THREADS
```

Ces deux macros créent un bloc donc toute variable déclarée entre ces deux macros est détruite en sortant du bloc. Numpy utilise ce système lors de ses appels aux fonctions C sous-jacentes pour permettre à un autre thread Python de travailler.


La réacquisition du GIL pendant le bloc (par exemple lors d'un appel à une fonction Python) se produit avec les instructions :


```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();
Puis son relâchement par :
PyGILState_Release(gstate);
```

 *Numpy propose une série plus complexe de macros pour ces opérations qui sont exposées dans le livre [Guide To Numpy](#).*

## II - Introduction à l'interface C de Numpy

Numpy est un module d'extension proposant donc de récupérer des fonctions pour travailler sur un tableau en C. Pour importer ces fonctions, le fichier d'entête **numpy/arrayobject.h** doit être importé et la fonction **import\_array()** exécutée. Le dossier à ajouter dans la liste des inclusions peut s'obtenir en Python grâce à la fonction **numpy.get\_include()**.

 La fonction **import\_array()** n'est pas exactement celle proposée ci-dessus. Il s'agit en réalité d'une macro encapsulant la fonction présentée et si le résultat est négatif, symbole d'une importation manquée, une exception est levée et la fonction appelante sera quittée (puisqu'il s'agit d'une macro, cela est possible).

 L'interface complète propose plus de 200 fonctions, sans compter les différents types de structures. Pour un exposé plus complet, le livre de Travis Olliphant [Guide To Numpy](#) est indispensable.

L'objet Numpy usuel est **PyObjectArray** :

```
typedef struct _PyObjectArray {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyObjectArray;
```

La macro **PyObject\_HEAD** définit le compteur de référence ainsi que le type utilisé (une instance de **PyTypeObject**) et est indispensable à l'utilisation de la structure en Python. C'est cette macro qui permet aussi de transformer la structure en un **PyObject**. Les champs suivants sont un champ de données, un champ contenant le nombre de dimensions, le champ des tailles selon chacune des dimensions et le champ des strides. Ensuite vient un champ **base** qui contient en réalité un objet acquis, par exemple s'il s'agit d'une vue sur un tableau, **base** contient une référence vers cet objet. Le champ **descr** contient un champ de description plus complexe décrit un peu plus loin. Ensuite viennent les drapeaux et un pointeur vers une liste de références faibles (généralement non utilisé).

 Dans Python 3k, **PyObject\_HEAD** ne déclarera plus simplement quelques champs mais sera une véritable structure afin d'être conforme au standard du C.

Les drapeaux proposés les plus courants pour cette structure sont :

- **NPY\_CONTIGUOUS** ou **NPY\_C\_CONTIGUOUS** indiquant que le tableau est continu avec un ordre C (**0x01**)
- **NPY\_FORTRAN** ou **NPY\_F\_CONTIGUOUS** indiquant que le tableau est continu et avec un ordre Fortran (**0x02**)
- **NPY\_OWNDATA** précisant que le tableau possède les données et qu'elles devront être détruites avec la destruction du tableau (**0x04**)
- **NPY\_ALIGNED** indiquant que les données sont alignées, utile pour utiliser des instructions vectorielles sur les processeurs les supportant (**0x0100**)
- **NPY\_NOTSWAPPED** indiquant que les données sont stockées dans le bon format (little endian sur une machine little endian par exemple) (**0x0200**)
- **NPY\_WRITEABLE** précisant qu'il est autorisé d'écrire dans le tableau (**0x0400**)

- **NPY\_UPDATEIFCOPY** indiquant que le contenu de base doit être mis à jour lorsque ce tableau sera détruit (**0x1000**)
- **NPY\_ARR\_HAS\_DESCR** indique que le tableau possède un champ de description **PyArray\_Descr**

La structure **PyArray\_Descr** est la suivante :

```
typedef struct _PyArray_Descr {
    PyObject_HEAD
    PyTypeObject *typeobj;
    char kind;
    char type;
    char byteorder;
    char hasobject;
    int type_num;
    int elsize;
    int alignment;
    struct _arr_descr *subarray;
    PyObject *fields;
    PyObject *names;
    PyArray_ArrFuncs *f;
} PyArray_Descr;
```


Plusieurs champs ne sont pertinents en première approche. Les champs utiles sont **kind**, une abréviation d'un caractère d'un type complet (comme **'f'** pour un nombre flottant 64bits), **type**, un caractère représentant ce type (**'i'** pour un entier, **'u'** pour un entier non signé, ...), **byteorder**, un autre caractère indiquant le format d'enregistrement (**'<'** pour little endian, **'>>'** pour big endian, **'|'** lorsque cela n'est pas pertinent et **'='** pour indiquer l'utilisation du format native), **type\_num** étant un nombre décrivant le type complet (par exemple un double est symbolisé par **NPY\_DOUBLE**), **elsize** donnant la taille de l'élément (un entier sur 1, 2 ou 4 octets par exemple) et **alignment** permettant d'obtenir l'alignement nécessaire pour ce type de données. Les autres champs sont utilisés pour des types plus complexes ou pour stocker les fonctions travaillant sur ce type de tableau.

Lorsqu'un tableau de ce type est passé à un fonction C, le champ de la fonction **PyArg\_ParseTuple** est un peu plus complexe. On utilise le symbole **"O!"** pour indiquer que le type est un type additionnel et deux arguments doivent être donnés pour cet argument. Le premier est un pointeur vers un objet de conversion, ici **PyArray\_Type**, et le second est un pointeur vers l'objet à utiliser.

#### Exemple

```
PyArrayObject *tableau;


PyArg_ParseTuple("O!", &PyArray_Type, &tableau);
```

 *La chaîne **"O"** indique que l'objet retourné doit être un objet Python classique. Il n'y a donc pas de conversion effectuée et dans ce cas, l'objet Python retourné est emprunté. Lors de la création d'un objet (généralement un tuple) avec **Py\_BuildValue()**, **"O"** indiquera d'acquérir l'élément pour le placer dans l'objet résultat et **"N"** indiquera que l'objet placé empruntera la référence. En général, ce sera donc **"N"** qui sera utilisé.*

Quelques fonctions indispensables :

- **PyArray\_FromDims()** ou **PyArray\_SimpleNew()** permet de créer un nouveau tableau, le premier argument étant le nombre de dimensions, le suivant un tableau contenant la taille sur chacune d'elle et le dernier est un caractère décrivant le type (**NPY\_DOUBLE**, **NPY\_LONG**, **NPY\_CDOUBLE** pour un nombre complexe double, ...)

- **PyArray\_SimpleNewFromData()** crée un tableau comme les fonctions précédentes avec un argument supplémentaire qui contient les données qui seront copiées dans le tableau
- **PyArray\_ContiguousFromObject()** prenant un objet Python en paramètre, le type de données à retourner (toujours **NPY\_LONG** ou autre), le nombre de dimension minimum et le nombre maximum de dimension supportés)
- Cette dernière fonction a été remplacée par **PyArray\_FromAny()** qui prend comme argument l'objet à convertir, un **PyArray\_Descr\*** décrivant le type de données à obtenir, puis le nombre minimum et maximum de dimensions, une série de drapeaux tels que **NPY\_CONTIGUOUS** et enfin un contexte si l'objet Python passé en paramètre n'est pas un objet Numpy mais possède une méthode **\_\_array\_\_()**, le contexte étant passé comme deuxième paramètre à cette méthode, le premier argument étant le code
- **PyArray\_DescrFromType()** crée un **PyArray\_Descr\*** à partir d'un type donné
- **PyArray\_ContiguousFromAny()** est une fonction d'aide basée sur **PyArray\_FromAny()** et **PyArray\_DescrFromType()** pour fournir la même interface et le même résultat que **PyArray\_ContiguousFromObject()**
- **PyArray\_NewFromDescr()** et **PyArray\_New()** créent à partir d'un pointeur sur un objet type (généralement **PyArrayType** ou une dérivée), d'une description ou d'un type de données, d'un certain nombre de dimensions, d'un tableau de dimensions associées et d'un autre tableau de strides (potentiellement vide si les données sont contigües), d'un pointeur vers des données (nul si la mémoire doit être allouée) et un ensemble de drapeaux associé. Un dernier argument est utilisable, il s'agit d'un pointeur vers un objet Python qui sera donné en argument à la méthode **\_\_array\_finalize\_\_** de la classe dérivée de **PyArrayType** utilisée.
- **PyArray\_Return()** simplifie le retour, assurant par exemple qu'il ait au moins une dimension 1 (il est en effet possible de créer des tableaux de dimension 0 qui peuvent se comporter comme des valeurs simple dans les fonctions Numpy, mais pas dans les autres routines)

 *La création d'un tableau à partir de données existantes est réalisée sur le pointeur des données passé en paramètre. Il faut donc s'assurer de la validité de ces données pendant toute la durée de vie du tableau. Si ces données proviennent d'un autre objet Python, il est possible d'acquérir une référence et de placer le champ base sur cet objet qui sera alors relâché à la destruction du tableau.*

Enfin, la structure **PyArrayInterface** permet d'adapter un type existant à Numpy. Cette interface doit être retournée lors de la récupération de l'attribut **\_\_array\_struct\_\_** d'une classe.

```
typedef struct {
    int two;
    int nd;
    char typekind;
    int itemsize;
    int flags;
    Py_intptr_t *shape;
    Py_intptr_t *strides;
    void *data;
    PyObject *descr;
} PyArrayInterface;
```

Voici la signification de ces champs. **two** contient le nombre 2, **nd** est le nombre de dimensions du tableau, **typekind** est un caractère indiquant le type de données stockées, **itemsize** étant le nombre d'octets utilisés, **flags** est une série de drapeaux décrivant le tableau, **shape** est un tableau contenant la taille selon chaque dimension, **strides** est un tableau contenant la distance d'un élément dans une dimension à un autre et **data** contient les données. L'objet **descr** est présent si le drapeau correspondant est placé (**NPY\_ARR\_HAS\_DESCR**) et contient une description plus fine des objets contenus (on parle de version 3 de l'interface).

## Conclusion

C'est la fin de cette première partie consacrée à l'API C. Pour télécharger le code source des exemples ainsi que d'autres exemples, reportez vous à **la page consacrée à mon livre**.

