

# Interface C ou C++ et Python avec SWIG



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

Date de publication : 30/01/2008

Dernière mise à jour :

Pour programmer une interface entre Python et le C ou le C++ sans utiliser directement l'interface C de Python, on utilise des outils externes tels que SWIG.

Ce texte est issu du livre **Python - les fondamentaux du langage - la programmation pour les scientifiques** aux éditions ENI.

Introduction

I - Utilisation simple

II - SWIG et le C++

III - Création et utilisation de convertisseurs

IV - SWIG avec Numpy


V - Exemple d'encapsulation d'une structure avec exposition de l'interface Numpy


Conclusion

## Introduction

L'interface C est très riche et par conséquent aussi lourde à gérer. Heureusement, des outils automatiques permettent d'encapsuler des bibliothèques C ou C++ avec peu de code. SWIG est spécialisé dans ce domaine mais sa présentation pourrait faire l'objet d'un livre complet, seule une introduction à son utilisation sera effectuée ici.

**SWIG** est un utilitaire permettant de créer à partir d'un fichier de configuration **.i** des modules d'encapsulation pour plusieurs langages, dont Python. L'exécutable SWIG crée à partir d'un fichier **module.i** et avec l'option **-python** un fichier **module.py** et un fichier **module\_wrap.c** (il est possible d'encapsuler du code C++ avec l'option supplémentaire **-c++** auquel cas le fichier possède une extension **.cpp**). Le fichier **.c** peut alors être compilé en un module appelé **\_module.(so, dll ou pyd** selon la plateforme et la version).

 **distutils** est capable de générer un module à partir des fichiers **.i** et des autres fichiers **.c** ou **.cpp** nécessaires. Il est important d'ajouter l'argument **swig\_opt=['-c++']** dans la création de l'extension si un code C++ doit être généré.

 Contrairement à plusieurs indications sur Internet, SWIG est capable de gérer le code C++ et les cas particuliers qu'il ne supportait pas ont été globalement résolus. En effet, ces critiques portent sur des anciennes versions (la 1.1) tandis que les versions 1.3 actuellement utilisées sont tout à fait adaptées à l'encapsulation de code C ou C++.

Les directives SWIG commencent par % et un code SWIG n'est donc pas compilable par un compilateur C ou C++. Si un code SWIG doit être inclus dans un fichier source sans être lu par le compilateur ou si un code C/C++ ne doit pas être lu par SWIG, il est possible de tester la macro **SWIG**. Si elle est définie, SWIG est en train d'analyser le fichier, sinon il s'agit vraisemblablement d'un compilateur.

Lors de la génération du code, toute fonction retournant un résultat complexe par valeur ou prenant des arguments complexes par valeur sera transformée en une fonction retournant un pointeur et prenant en argument des pointeurs. Par complexe, il est entendu tous les types de structures ou de classes non déclarés au moment de l'exécution de SWIG (en effet, celui-ci est plus permissif qu'un compilateur C ou C++).


## I - Utilisation simple

L'architecture globale d'un fichier `.i` est la suivante :

```
%module mon_module
%{
/* Déclarations supplémentaires */
%}

/*Liste de fonctions, variables ou constantes à inclure dans le module */
void ma_fonction(void);
```

Tout ce qui se trouve dans le bloc `%{%` sera intégralement copié dans le fichier source généré, donc toute inclusion d'entête est à déclarer à cet endroit. De même, il est important d'ajouter le ou les fichier(s) d'entête où sont stockés les prototypes des fonctions utilisées. La déclaration `%module` déclare le nom du module à créer, il s'agit couramment du nom du fichier sans l'extension (ceci est une convention). Enfin, une liste de fonctions, de variables ou de constantes qui seront exposées du module doit être écrite. Cette liste d'exposition peut inclure un entête où une sous-partie est déclarée avec la directive `%include` (semblable à la directive du préprocesseur `#include`).

 *Pour chaque module externe compilé, un module Python est créé. Ce module Python encapsule le module externe, ce dernier possède alors un autre nom, un préfixe '\_' devant être ajouté au nom du module Python lors de sa compilation. C'est le module Python qui sera utilisé directement, comme cela est préconisé dans le tutoriel sur l'API C.*

Quelques directives peuvent spécifier des comportements particuliers :


- `%immutable;` et `%mutable;` interdisent ou autorisent l'écriture sur les attributs suivant la directive
- `%rename(nouveau_nom) ancien_nom;` modifie toutes les références à `ancien_nom` et les remplace par `nouveau_nom` (ce qui est pratique dans le cas où `ancien_nom` est un mot-clé réservé de Python)
- `%ignore nom;` permet d'ignorer toutes les déclarations portant sur `nom`

Lorsqu'une fonction déclare pouvoir prendre en paramètre une autre fonction et que ces fonctions sont exposées dans le module, il n'est pas possible de les utiliser dans Python. Par exemple :

```
void fonction((int (*argument)(int, int)));

int add(int, int);
```

La fonction `add` accessible depuis Python est une version encapsulée de la version C qui n'est donc plus disponible pour être donnée comme argument à `fonction()`. Pour résoudre ce problème, on utilise la directive `%callback("%s_callback")` et `%nocallback` pour indiquer à SWIG de créer deux objets : le premier est la fonction `add()` classique, la seconde s'appellera `add_callback()` et pourra être donnée comme argument à `fonction`. Elle n'est en revanche plus appellable depuis Python.

 *A la place de `"%s_callback"`, il est possible d'utiliser `"%(title)s"` qui utilisera le nom de la fonction avec une majuscule, `"%(upper)s"` pour créer un nom en majuscules uniquement ou `"%(lower)s"` pour uniquement des minuscules. Cette chaîne est semblable à la chaîne de format de la fonction `printf()`.*

Un grand intérêt de SWIG est d'autoriser la création de classes à partir d'une structure C. Si une structure a été déclarée, la directive `%extend` permet de créer un bloc dans lequel pourront être définies des méthodes travaillant

sur l'instance courante de la structure (dans le cas contraire, SWIG génère lui-même plusieurs wrappers vers les attributs de la structure) :


```
typedef struct structure{
} Structure;
%extend Structure{
    Structure(/* Arguments */)
    {
        Structure* s;
        s = (Structure*) malloc(sizeof(Structure));
        /* Initialisations */
        return s;
    }


    ~Structure()
    {
        free($self);
    }


    void methode()
    {
        /* Methode de la structure */
    }
}
```

L'instance courante dans ces méthodes est **\$self**. Pour le constructeur, celle-ci n'existe pas et l'utilisateur doit allouer l'espace mémoire et le désallouer à la destruction de l'objet. En revanche, il est maintenant possible d'exécuter en Python :

```
a = Structure()
a.methode()
```

 Le mot-clé **%extend** peut être utilisé au moment de la déclaration de la structure (dans le bloc **struct**). SWIG est aussi capable de détecter les méthodes potentielles parmi les fonctions libres proposées. Par exemple si **new\_Structure()** existe, la déclaration du constructeur peut se limiter à **Structure(/\* avec les arguments \*/)**; si **delete\_Structure()** prend un seul paramètre, elle pourra être utilisée pour le destructeur directement (pas d'implémentation du destructeur dans **%extend**) et si une fonction **Structure\_methode()** dont le premier argument est un pointeur vers une Structure existe, la méthode **methode()** pourra être déclarée sans implémentation dans **%extend** et cette fonction sera appelée automatiquement.

 Si la fonction libre se termine en plus par **\_get** ou **\_set**, SWIG implémentera un attribut en lecture ou en écriture et non une méthode.

 Une structure n'a pas besoin d'être déclarée totalement à SWIG. En effet, seul le compilateur par la suite aura besoin d'une définition complète, SWIG ne génère que des fonctions qui vont travailler ensuite sur la structure dans son ensemble.

Si une fonction a besoin d'être déclarée dans le bloc **%{ }** ainsi que dans la liste d'exposition, la directive **%inline** peut être ajoutée avant le bloc **%{ }**. Dans ce cas, SWIG et le compilateur utiliseront ce code (très utile pour déclarer des fonctions utilitaires qui devront être exposées comme des allocations ou des destructions, mais il n'est pas possible d'ajouter des directives SWIG dans ce code puisqu'il sera compilé en C ou en C++).

Une macro SWIG est déclarée par un bloc **%define ma\_macro %endef** dans le cas d'une constante ou pour une fonction **%define ma\_macro(ARG1, ARG2) %endef**.

Avant de passer à une partie plus avancée, quelques règles sont proposées pour créer un module encapsulant un code existant :


- établir une liste précise des fonctions, variables, structures à exposer
- inclure uniquement le minimum utile de fichiers d'entêtes dans les blocs d'inclusion `%{%`
- créer un fichier d'interface contenant les inclusions des entêtes et les définitions des fonctions nécessaires
- ne pas hésiter à séparer en plusieurs morceaux le fichier d'interface et à inclure les différents sous-fichiers afin d'avoir un fichier d'interface principal clair

## II - SWIG et le C++

SWIG supporte presque tout le langage C++, à part quelques surcharges telles que celles de **new** et **delete** ou encore les classes imbriquées, mais le reste est intégralement supporté.

Lorsqu'il s'agira de classes, SWIG va générer une classe dite proxy en Python qui va gérer l'interface avec la classe sous-jacente C++. Des mécanismes internes sont proposés pour qu'une instance de la classe proxy puisse être assignée à un attribut d'une classe cible. Par exemple **Classe1** possède un pointeur sur **Classe2**, en Python, il sera possible d'assigner une instance Python à ce pointeur. En général, le mécanisme interne suffit pour gérer correctement la mémoire. Si cela n'est pas le cas, il faudra gérer à la main les cas particuliers à l'aide des méthodes **disown()** et **acquire()** de la classe proxy. Attention aussi, si un objet Python est créé à partir d'un pointeur C++ sur **Classe2**, si le pointeur est détruit (car l'instance de **Classe1** a été détruite), l'objet sera dans un état indéfini car il référencera un objet détruit.


SWIG crée par défaut une encapsulation du constructeur et du destructeur, même s'ils ne sont pas explicitement générés (et que le C++ génère une version par défaut). Pour désactiver cette création, les directives **%nodefaultctor**; et **%nodefaultdtor**; sont proposées, ainsi que **%clearnodefaultctor**; et **%clearnodefaultdtor**; pour réactiver la génération. En ce qui concerne le constructeur de copie, l'activation se fait par la directive **%copyctor**; et la désactivation par **%nocopyctor**; (par défaut le constructeur de copie n'est pas créé).

 *Si les constructeurs ou le destructeur sont protégés ou privés ou si la classe encapsulée est virtuelle pure, ils ne seront pas encapsulés.*

En ce qui concerne les attributs et les méthodes statiques, SWIG génère automatiquement le code adéquat, soit des accesseurs soit un appel simple à la fonction statique.

SWIG gère aussi de manière transparente l'héritage, si une classe hérite d'une autre en C++, la version encapsulée de l'une héritera aussi de la version encapsulée de l'autre.

Enfin, SWIG est capable de gérer dans la limite du possible la surcharge des fonctions en testant le nombre d'arguments et leur type pour savoir quelle fonction ou quelle méthode appeler par la suite.


 *Si une fonction prenant un entier est surchargée par une fonction prenant en entrée un autre type d'entier, SWIG ne sera pas capable de choisir laquelle utiliser car en Python il n'y a qu'un seul type d'entier (sans compter les entiers longs). L'une des fonctions sera alors choisie et un message d'avertissement affiché. Une solution pour contrer ce problème est d'ignorer la fonction inutile ou de renommer les fonctions.*

Un point majeur du C++ est le concept de template. Pour réussir à encapsuler un template (créer un wrapper), il faut l'instancier. Il est toujours possible de le faire explicitement à l'aide d'une déclaration dans la liste d'exposition du template complet instancié pour un type, mais cela est long et le nom du type doit être changé (Python ne supporte pas **vector<int>** comme type de données). Dans ce cas-là, pour un template présenté dans un bloc **%{ }**, la directive **template(vector\_int) vector<int>**; permet d'instancier le type **vector** avec le type **int** et d'appeler ce type **vector\_int**.

En ce qui concerne les espaces de nom du C++, ils sont tous supprimés et les classes sont exposées dans le module courant (pour exposer un sous espace de nom, un sous-module peut être envisagé). Il est impératif de vérifier si différentes classes dans différents espaces de nom ont des noms identiques.

Les exceptions sont supportées de manière native grâce aux convertisseurs automatiques (présentés un peu plus tard) si les exceptions sont spécifiées dans le prototype de la fonction. Dans le cas où une fonction ou méthode lève une exception qui n'est pas dans cette liste, une directive **%catches()** précédant une déclaration de la fonction indique qu'une certaine série d'exceptions (celles proposées dans la parenthèse) sont transformées en des exceptions Python

semblables, et/ou si ... est donné dans cette parenthèse, toute exception est attrapée et relancée sous la forme d'une exception générique.

 *Les pointeurs intelligents sont gérés par SWIG. Si une classe expose l'opérateur `->`, tous les attributs et méthodes de la classe pointée seront accessibles par une instance du pointeur intelligent, sauf si l'attribut ou la fonction existe dans le pointeur.*

### III - Création et utilisation de convertisseurs

Plusieurs convertisseurs par défaut sont proposés par SWIG et aussi par Numpy. Les conteneurs de la STL sont définis chacun dans un fichier **std\_conteneur.i**. En important ces fichiers, SWIG saura comment convertir le type C ou C++ en type Python.

En ce qui concerne le fichier **exception.i** qui permet de rediriger les exceptions C++ en exceptions Python, une directive permet de changer le mode de conversion :


```
%exception /*une fonction ou méthode particulière à protéger*/{
  try
  {
    $action
  }
  catch(const std::runtime_error& e)
  {
    SWIG_exception(SWIG_RuntimeError, const_cast<char*>(e.what()));
  }
}
```

Par défaut toutes les fonctions ou méthodes seront protégées si aucune fonction ou méthode n'est indiquée. Dans ce cas-ci, SWIG génère l'exception adéquate, le type d'erreur proposé peut être **SWIG\_MemoryError**, **SWIG\_IOError**, **SWIG\_RuntimeError**, **SWIG\_IndexError**, **SWIG\_TypeError**, **SWIG\_DivisionZeroError**, **SWIG\_OverflowError**, **SWIG\_SyntaxError**, **SWIG\_ValueError** ou **SWIG\_SystemError**.

Le principal outil de conversion est la directive **%apply** et la directive **%clear** qui stoppe les effets de la première. La directive a cette syntaxe :

```
%apply resultat{type_a_modifieur};
/*code à convertir*/
%clear resultat ;
```

Le résultat est le nom de la règle de conversion utilisée et est le type qui remplacera le type donné entre accolades.

 Cette solution doit être utilisée pour indiquer qu'un pointeur vers une **std::string** doit être transformé en un **const string&** ou une autre solution basée sur des références. Pour cela, la directive **%apply const string& {std::string\*}**; automatise toutes les conversions.

Un atout de taille des conversions est de pouvoir spécifier qu'un argument passé en paramètre est un argument de retour, cet argument ne sera pas donné à l'appel en Python mais retourné par la fonction Python (et s'il existe plusieurs arguments, un tuple sera retourné). De même des arguments en entrée et en sortie sont possibles, ils seront toujours retournés et modifiés sur place si leur type le permet.

```
%apply int* OUTPUT {int* resultat};
void ma_fonction(int a, int b, int* resultat);
```

Cette fonction sera appelée ainsi en Python :

```
resultat = ma_fonction(a, b);
```

Une conversion en entrée/sortie est donnée par **INOUT** et une conversion en entrée seulement est donnée par **INPUT**.

Des contraintes peuvent aussi être appliquées :

- **POSITIVE** pour assurer qu'un nombre est strictement positif
- **NONNEGATIVE** pour assurer qu'il est positif ou nul
- **NEGATIVE** pour un nombre strictement négatif
- **NONPOSITIVE** pour un nombre négatif ou nul
- **NONZERO** pour un nombre différent de zéro
- **NONNULL** pour un pointeur non nul.

Si ces contraintes ne sont pas respectées, une exception Python sera levée.

 *Certaines de ces contraintes sont disponibles depuis le fichier **constraints.i**.*


Le réel mécanisme de conversion permettant à l'utilisateur de définir une méthode de conversion d'un type en un autre est les typemaps. La syntaxe est la suivante :

```
%typemap(méthode[, modificateurs]) liste_types code;
```

La méthode indique ce que le typemap pourra faire dont les suivantes :


- **in** effectue une conversion du type Python vers le C ou le C++
- **out** effectue l'opération inverse
- **typecheck** permet de vérifier le type donné en entrée (utilisé pour déterminer quelle fonction surchargée doit être utilisée) en retournant 1 si le type est correct et 0 sinon
- **argout** permet d'ajouter à un résultat d'autres valeurs données en argument, le résultat final étant alors un tuple ou une liste (c'est ce qui est utilisé pour les variables passées en argument qui sont en fait des résultats)
- **check** vérifie les valeurs données en entrée (par exemple autoriser uniquement les valeurs positives d'un entier) après la conversion
- **arginit** initialise une donnée avant la conversion (en général, ce n'est pas utile)
- **default** permet d'appliquer une valeur par défaut à l'argument d'une fonction
- **freearg** est appelée à la fin de l'encapsulation pour libérer de la mémoire allouée pendant un typemap in
- **throw** gère les types d'exceptions

La liste de types **list\_types** est une liste séparée par des virgules des motifs qui seront validées pour la conversion. Il peut s'agir d'un type simple, d'un type avec un nom d'argument (auquel cas seuls les arguments avec le même nom et le même type seront valides) ou d'une liste de types simples ou avec un nom entre parenthèses (dans le cas où un objet Python serait transformé en plusieurs objets en C ou en C++). Les types qui sont redéfinis avec un typedef dans des espaces de nom différent valident aussi le motif si la définition originale valide ce motif. Enfin, avec la directive **%apply**, il est possible d'appliquer à un nouveau motif un motif défini par un typemap.

 *Un typemap est valide pour tout le code qui suit jusqu'à ce qu'il soit redéfini ou effacé par **%clear**. En revanche, si **%extend** est utilisé pour étendre les possibilités d'une classe, le typemap doit être défini avant la définition de la classe.*

Les arguments donnés au typemap peuvent être récupérés lors de la déclaration. Par exemple **\$n** est l'argument **n** donné dans le motif (il s'agit donc de la destination de la conversion), **\$n\_name** est son nom, **\$n\_type** son type et

**\$n\_ltype** son type simplifié (sans mot clé **const**, les tableaux statiques sont transformés en pointeurs, ...). Si **\$\*** est utilisé à la place de **\$**, le type retourné est le type pointé et **\$&** rajoute un pointeur. **\$n\_basetype** indique quant à lui le type sous-jacent simple (**int**, **float**, ...) sans aucun pointeur ni **const**. Pour des tableaux passés en paramètre dont la taille est fichée, **\$n\_dimi** donne la taille selon la dimension **i**.

 Les *typemaps* **in**, **out** et **argout** possèdent des variables supplémentaires. La première est **\$symname**, contenant le nom de la fonction en cours d'encapsulation. Ensuite **\$input** est défini pour **in** et **argout**, et **\$output** pour **out** et **argout**.

Si des valeurs temporaires sont nécessaires pour contenir les données pendant la durée d'appel de la fonction, un *typemap* **in** peut les spécifier entre parenthèses après **liste\_types**, avant le code. Même si plusieurs *typemaps* identiques sont nécessaires pour convertir les données, les variables temporaires ne se chevaucheront pas, SWIG rajoutera un numéro distinctif à la fin du nom de ces variables. Cette variable temporaire peut être accédée dans un *typemap* **argout** en postfixant le nom de la variable par **\$argnum** qui est le numéro de l'argument modifié (accéder à la variable temporaire doit faire l'objet de vérifications minutieuses pour bien être certain de ce qui se passe).

## IV - SWIG avec Numpy

Numpy propose des typemaps préexistants pour SWIG dans le fichier **doc/numpy.i**. Ce fichier permet de définir :

- des tableaux en entrée
- des tableaux modifiés sur place
- des tableaux en sortie mais passés en arguments

Ces typemaps peuvent fonctionner sur un tableau à plusieurs dimensions (3 maximum), sur un pointeur avec les dimensions données avant ou après le pointeur. Par exemple :

```
%apply (int IN_ARRAY2[ANY][ANY]) {(int matrix[ANY][ANY]);  
void ma_fonction(int matrix[3][3]) ;
```

Le mot-clé **ANY** permet d'indiquer que toutes les tailles sont permises. Ici, on applique le typemap indiquant que l'on travaille sur un tableau à 2 dimensions (**IN\_ARRAY2**) et lorsque **ma\_fonction()** sera encapsulée, un tableau Numpy pourra être passé en paramètre.

Pour les tableaux modifiés sur place, **INPLACE\_ARRAY\*** sera utilisé et pour les tableaux en sortie, il s'agira de **ARGOUT\_ARRAY\***. A noter que les tableaux de dimension 2 et 3 ne peuvent être spécifiés que sous la forme **tableau[][]** ou **tableau[][][]** et non sous la forme de pointeurs et de dimensions associées. C'est une limitation des typemaps proposés et non de SWIG.

Pour fonctionner correctement, le module d'extension de Numpy doit être chargé au préalable (la fonction **import\_array()**). Pour cela, la séquence suivante est utilisée :

```
%{  
#define SWIG_FILE_WITH_INIT  
%}  
%include "numpy.i"  
%init %{  
import_array();  
%}
```

Le bloc **%init %{}%** colle son contenu directement dans l'initialisation du module proposé. Un seul appel de ce type doit se produire lors du chargement du module, il ne faut pas le copier dans chaque sous-fichier d'interface.



*Le fichier d'interface **numpy.i** est un bon exemple complet de typemap mais aussi de définition de macros et de fonctions.*

## V - Exemple d'encapsulation d'une structure avec exposition de l'interface Numpy

Une classe Python complète sera créée, avec possibilité d'affichage et exposition de l'interface Numpy à l'aide de `__array_struct__`.

Voici la structure dans un fichier `numpy_swig.h` :

```
#ifndef NUMPY_SWIG
#define NUMPY_SWIG

typedef struct _SignedIntBuf
{
    int* data;
    int shape[2];
    int strides[2];
} SignedIntBuf;

#endif
```

Le fichier d'interface `numpy_swig.i` est maintenant exposé ici en plusieurs parties :

```
%{
#define SWIG_FILE_WITH_INIT
%}
#include "numpy.i"
%init %{
import_array();
%}

%module (docstring="Ceci est un module Python encapsule par SWIG") numpy_swig
```

Un module est créé et une documentation associée est proposée par l'argument `docstring`.

```
%{
#include "numpy_swig.h"

void delete_SignedIntBuf(SignedIntBuf* buffer)
{
    free(buffer->data);
    free(buffer);
}

void free_array_interface( void* ptr, void *arr )
{
    PyArrayInterface* inter;
    PyObject* arrpy;

    inter = (PyArrayInterface*)ptr;
    arrpy = (PyObject*)arr;
    Py_DECREF(arrpy);
    free(inter);
}
%}
```

Deux fonctions de destruction sont données telles quel dans le code qui sera généré. La première est le destructeur d'un `SignedIntBuf`, structure qui a été définie dans l'entête précédent, et la seconde est la fonction de désallocation.

```


%inline %{PyObject* get__array_struct__(PyObject* self, int* shape, int* strides, int*data)
{
  PyArrayInterface* inter;
  PyObject* obj;
  int nd;
  nd = 2;


  inter = (PyArrayInterface*)malloc(sizeof(PyArrayInterface));
  if (inter==NULL)
    return PyErr_NoMemory();

  inter->two = 2;
  inter->nd = nd;
  inter->typekind = 'i';
  inter->itemsz = sizeof(int);
  inter->flags = NPY_NOTSWAPPED | NPY_ALIGNED | NPY_WRITEABLE;
  inter->strides = strides;
  inter->shape = shape;
  inter->data = data;
  Py_INCREF(self);
  obj = PyCObject_FromVoidPtrAndDesc((void*)inter, (void*)self, free_array_interface);
  return obj;
}
%}

```

Voici la fonction principale créant l'interface Numpy à partir d'un objet. En réalité, cette fonction prend en paramètre l'objet encapsulé **SignedIntBuf** mais aussi directement les paramètres nécessaires à la création de l'interface comme les dimensions et les données. Rien n'est extrait manuellement ici, c'est SWIG qui devra faire le travail.

 Si l'allocation de la mémoire échoue, une exception est levée, celle-ci fait partie des exceptions présentées dans une partie précédente.

 Ce code est reporté intégralement dans le code généré et SWIG connaît la fonction, grâce à la directive **%inline**.

Maintenant vient la déclaration des fonctions et variables exportées avec les extensions :

```

#include numpy_swig.h

%extend SignedIntBuf{
  %feature("autodoc", "Une chaîne de commentaire")SignedIntBuf;
  SignedIntBuf(int width, int height)
  {
    SignedIntBuf* buffer = (SignedIntBuf*) malloc(sizeof(SignedIntBuf));
    buffer->shape[0] = height;
    buffer->shape[1] = width;
    buffer->strides[0] = width * sizeof(int);
    buffer->strides[1] = sizeof(int);
    buffer->data = (int*) malloc(width*height*sizeof(int));
    return buffer;
  }

~SignedIntBuf();

char *__str__()
{
  static char tmp[1024];
  int i, j;
  int used = 0;
  used += sprintf(tmp, "Tableau :\n");
  for(i=0; i < $self->shape[0]; i++)


```

```
{
    for(j=0; j < $self->shape[1]; j++)
        used += sprintf(tmp + used, "%d\t", $self->data[j + i*$self->shape[1]]);
    used += sprintf(tmp + used, "\n");
}
return tmp;
}

%pythoncode
{
    def __array_struct__get(self):
        return get__array_struct__(self, self.shape, self.strides, self.data)

    __array_struct__ = property(__array_struct__get, doc='Array protocol')
}
};
```

Après avoir inclus l'entête C, une extension de la structure est proposée. Le constructeur est ici exposé entièrement tandis que le destructeur est simplement prototypé puisqu'il a été défini dans une partie précédente.

 **%feature("autodoc", valeur);** permet de créer une chaîne de documentation automatiquement pour valeur valant "0" ou "1". Ici, une chaîne explicite est donnée pour une fonction particulière.

Une solution est donnée pour retourner une chaîne de caractères décrivant l'objet. Ici, en C, un tableau statique de caractères est utilisé et retourné, mais une chaîne de caractères C++ aurait été plus simple à manipuler.

Enfin, une directive **%pythoncode** est donnée, elle permet de recopier intégralement le bloc dans le code Python généré. Ici cela est nécessaire car **\_\_array\_struct\_\_** doit être une propriété Python. De plus, si une méthode C est directement donnée pour l'accessor, il est impossible de récupérer le pointeur original Python sur l'objet, ce qui est nécessaire pour indiquer que l'objet est référencé dans l'interface (ou on risque une corruption de la mémoire). C'est pourquoi l'accessor appelle une fonction libre du module avec les paramètres supplémentaires qui seront automatiquement convertis par SWIG en les types sous-jacents.

## Conclusion

Ainsi s'achève cette introduction à SWIG et son utilisation pour Numpy. L'exemple donné en dernière partie est unique sur Internet mais est extrêmement utile pour encapsuler un ancien code scientifique et de le réutiliser facilement avec Numpy sans effectuer de copies à chaque utilisation.

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**.

