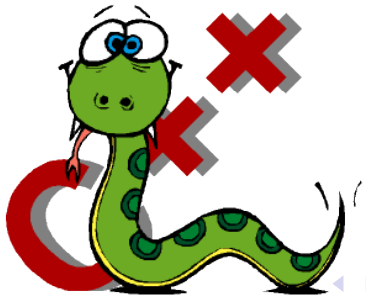
Interfaçage C/C++/Python

Adrien Saladin

Molécules Thérapeutiques In silico
Paris 7 – INSERM

August 29, 2010



Deux mondes totalement opposés ?

C++

- Compilé
- Typage statique
- Moyennement flexible
- Bibliothèque standard bas niveau

Python

- Interprété
- Typage dynamique
- Très flexible
- Bibliothèque standard riche et de haut niveau (serveur http, etc).

Langages complémentaires!

C++

- Rapide
- Relativement flexible

Python

- Pas toujours rapide
- Très flexible
- Bibliothèque standard riche et de haut niveau (serveur http, etc).

Interfaçage

Utiliser le meilleur de chaque langage, selon les circonstances.

Exemple d'applications scientifiques hybrides Python/C++/C/Fortran

- MMTK - The Molecular Modelling Toolkit
- Modeller - Homology modelling
- Sparky - NMR Assignment and Integration Software
- Code Aster (EDF) - Thermodynamics, physics, materials resitivity, ...
  Python - Fortran. 10^6 lines of code
- Civilization IV

API C Python

Exemple

```
static PyObject* llmat_set(
       PyObject* args) {
double val;
int row, col;
if (!PyArg_ParseTuple(args,
     "iid", &row, &col, &val))
return NULL;
Py_INCREF(Py_None);
return Py_None;
```

Pour et contre

Avantages

- Vitesse.
- Accès à toutes les fonctionnalités du langage.

Inconvénients

- Vitesse d'écriture
- Erreurs...
- Amusant les 10 premières minutes, rébarbatif ensuite.

Pour et contre

Avantages

- Vitesse.
- Accès à toutes les fonctionnalités du langage.

Inconvénients

- Vitesse d'écriture
- Erreurs...
- Amusant les 10 premières minutes, rébarbatif ensuite.

SWIG

But

Interfaçage de C/C++ avec de nombreux langages de haut niveau:

- Python, Lua, PHP, Ruby, Java, R, ...

Méthode

Fichier de directives ".i".

Fichier de directives


```
%module mymodule
%{
#include "myheader.h"
%}
// Now list ANSI C/C++ declarations
int foo;
int bar(int x);
...
```

Pour et contre

Avantages

- Plusieurs langages cible

Inconvénients

- Mains dans le cambouis.
- Fichiers .py et .so générés.

Pour et contre

Avantages

- Plusieurs langages cible

Inconvénients

- Mains dans le cambouis.
- Fichiers .py et .so générés.

Librairies C++ Boost

- Fabriquées par des warriors du C++
- Inclues dans C++-0x
- Métaprogrammation

Petit exemple

C++ code

```
string greet() { return "hello, world"; }
int square(int number) { return number * number; }
```

Interface (separated C++ file)

```
BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet);
    def("square", square);
}
```

Python shell

```
>>> import hello
>>> hello.greet()
'hello, world'
>>> hello.square(5)
25
```

Petit exemple

C++ code

```
string greet() { return "hello, world"; }
int square(int number) { return number * number; }
```

Interface (separated C++ file)

```
BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet);
    def("square", square);
}
```

Python shell

```
>>> import hello
>>> hello.greet()
'hello, world'
>>> hello.square(5)
25
```

Petit exemple

C++ code

```
string greet() { return "hello, world"; }
int square(int number) { return number * number; }
```

Interface (separated C++ file)

```
BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet);
    def("square", square);
}
```

Python shell

```
>>> import hello
>>> hello.greet()
'hello, world'
>>> hello.square(5)
25
```

Fonctionnalités avancées

- Surcharge de fonctions
- Exceptions C++ vers Python
- Polymorphisme (méthodes virtuelles)
- Serialization (pickle) des classes exportées
- ...

Exemple: méthodes virtuelles

C++

```
class A {
  virtual int inutile() = 0;
 }
 ...
 int calculate(A & a) {
    int important = a.inutile();
 }
```

Python

```
  class MeilleurA (A):
     def inutile():
        return 2
  b = MeilleurA()
  calculate(b)
```

Bilan Boost.Python

Avantages

- Code optimisé
- Limite les erreurs: fonctionne parfaitement ou bien erreur à la compilation.

Points négatifs

- Relativement long à compiler
- Tendance à l'embonpoint
- Code d'interfaçage à modifier au moindre changement des fonctions C++.
- Messages d'erreur de compilation (templates).
- Systeme de compilation Bjam (optionnel)
- Utilise C++ pour interfacer du C.

Py++

Qu'est-ce ?

- Automatisation de l'écriture du code d'interfaçage Boost.Python
- Évite d'ajouter à la main chaque méthode de chaque classe
- Module en Python
- Pygccxml, gccxml

Py++ example


```
from pyplusplus import module_builder
mb = module_builder.module_builder_t( [os.path.abspath('./
                                      , gccxml_path=r""
                                      , define_symbols=[]


mb.classes().exclude()
mb.free_functions().exclude()

atom = mb.class_("Atom")
atom.include()
rmsd = mb.free_function("rmsd")
rmsd.include()
...
mb.build_code_creator( module_name='hello' )
```

Bilan Py++

- Très pratique
- Facilite le développement rapide d'API

PyBindGen, le petit nouveau

A propos

- Générateur de code Python C-API.
- Code généré propre
- Binaires de petite taille
- Activement développé
- Utilise (facultatif) pygccxml

PyBindGen: exemple


```
#generator.py
import pybindgen
import sys

mod = pybindgen.Module('malib')
mod.add_include('"malib.h"')

mod.add_function("carre",
        pybindgen.retval ('int'),
        [pybindgen.param ('int', 'i')]
        )

mod.generate(sys.stdout)
```

Bilan PyBindGen

Avantages

- Code effectivement propre.
- Peu de dépendances
  - possiblité de ne distribuer que les fichiers d'interfaçage .c
- Rapide à compiler (pas de métaprogrammation C++)
- L'auteur répond aux e-mails

Points négatifs

- Relativement nouveau
- Manque encore de fonctionnalités
- Communauté réduite
- Moins robuste que Boost.Python ?
- Ne fonctionne pas avec easy_install

Autres outils

- SIP (PyQt)
- PyFort, Forthon (FORTRAN 77/90 bindings).
- Ctypes: pour C uniquement.
- Cython, Pyrex: approche top-down privilégiée

Est-ce vraiment utile ?

Python est lent

Oui, mais

- Pas toujours
  Certaines parties de Python sont bien optimisées
- Modules écrits en C/C++/Fortran (NumPy, ...)
- Optimise le temps du développeur
- PyPy, LLVM, ...

Est-ce vraiment utile ?

Python est lent

Oui, mais

- Pas toujours
  Certaines parties de Python sont bien optimisées
- Modules écrits en C/C++/Fortran (NumPy, ...)
- Optimise le temps du développeur
- PyPy, LLVM, ...

Est-ce vraiment utile ?

Python est lent

Oui, mais

- Pas toujours
  Certaines parties de Python sont bien optimisées
- Modules écrits en C/C++/Fortran (NumPy, ...)
- Optimise le temps du développeur
- PyPy, LLVM, ...

Conclusion

Quel outil utiliser ?

Bottom up

- SCWRL: seulement si plusieurs langages ciblés
- Boost.Python: en attendant mieux. Robuste.
- PyBindGen: à suivre de près

Top down

- Boost.Python (pourquoi pas)
- Cython

Remerciements

- PyConFr
- AFPY