Wrapping de code avec SWIG et Python

Matthijs Douze

Motivation

Passerelle Python-C

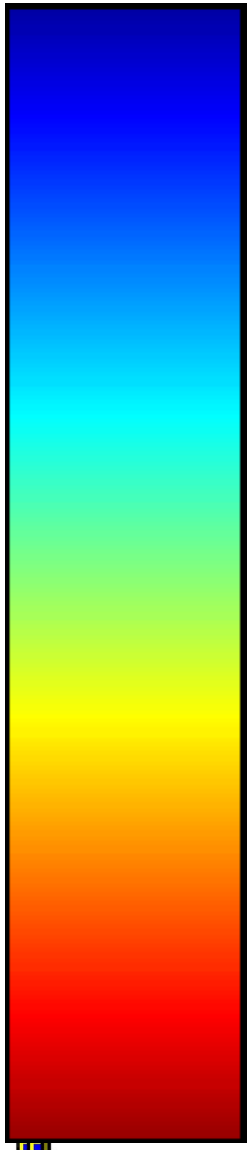
SWIG

Niveaux de langage

- Haut: temps de développement > temps d'exécution
  - **objectifs** : compact, “shell” interactif, backtrace
  - **défauts** : lent, dépendant de ce qui est dispo dans les librairies, peu d'analyse statique
- Bas: temps d'exécution > temps de développement
  - **objectifs** : efficace, contrôle précis
  - **défauts** : verbeux, nécessite compilation, plantages violents
- Frontière se déplace vers le haut niveau
  - loi de Moore pas pour les programmeurs...
  - 1980: C = haut niveau

Opérations

niveaux de langage



Parse ligne de commande

GUI

Lire texte/XML

Lire données binaires

compilation

Multiplication grosse matrice

Comment concilier ?

Langage	Vitesse / C (à la louche)
Cerveau humain	/10^9
Shell, make	/10000
TCL, perl, python, Matlab	/100
Java, caml	/5
C, Fortran	1
intrinsics SSE, assembleur	x4
CUDA, OpenCL	x50

Solution 1: Le Langage Qui Fait Tout

- Exemples : C++, Java, C#
  - Supporte à la fois le haut et le bas niveau
  - Extension “douce” si on connaît C
  - Packaging facile
- Danger : usine à gaz !
  - Complexe :
    - 750 pages de Stroustroup
    - Expansion perpétuelle:
      utilisateurs veulent des features (et on leur donne !)
    - Librairies qui font ce qu'on veut **à peu près**
  - Verbeux par rapport à script
    - Information diluée:
      déclarations, namespace, public/private...
    - Questions sans intérêt :
      vector ou linked list ? Pointeur ou ref ?

Solution 2: combinaison de 2 langages

- Prendre le meilleur des deux
- *Isoler* et optimiser le code critique
- Exemples:

Haut niveau	Bas niveau	Domaine
C	Assembleur, CUDA	système
shell	Fortran	Numéricien traditionnel
Matlab	C/Fortran (mex)	Statistiques, vision, ...
QuakeC, UnrealScript	C/C++	Jeux vidéo
Java	C (javah)	Web services, GUI
Scheme, python	C	GIMP
Python	**C**	numpy, ...

Comment combiner ?

Motivation

Passerelle Python-C

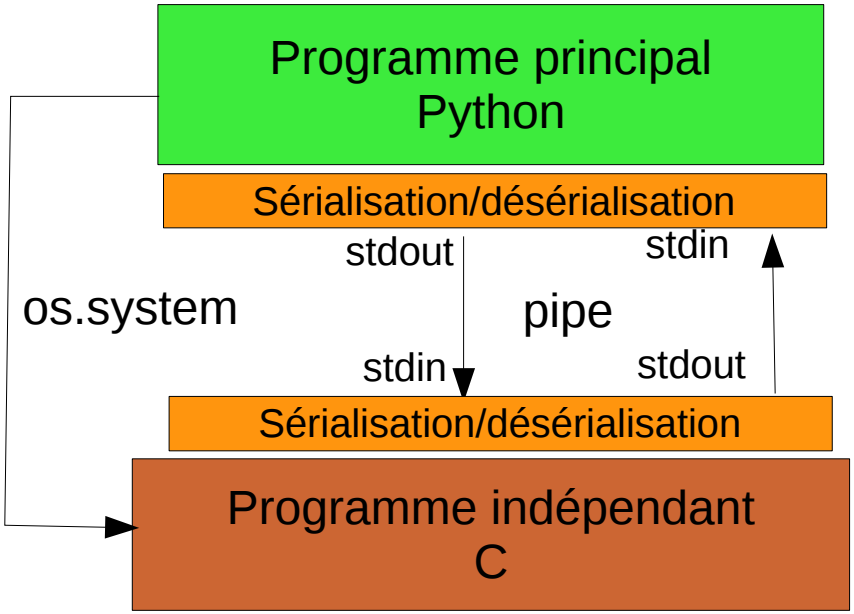
SWIG

Python

- Origine
  - 1991 (entre perl et ruby)
  - Académique : cycle développement rapide
- Syntaxe compacte:
  - indentation → structure
  - minimum de ponctuation
- Propriétés:
  - Moderne: unicode, multithread, générateurs,...
  - librairie standard riche

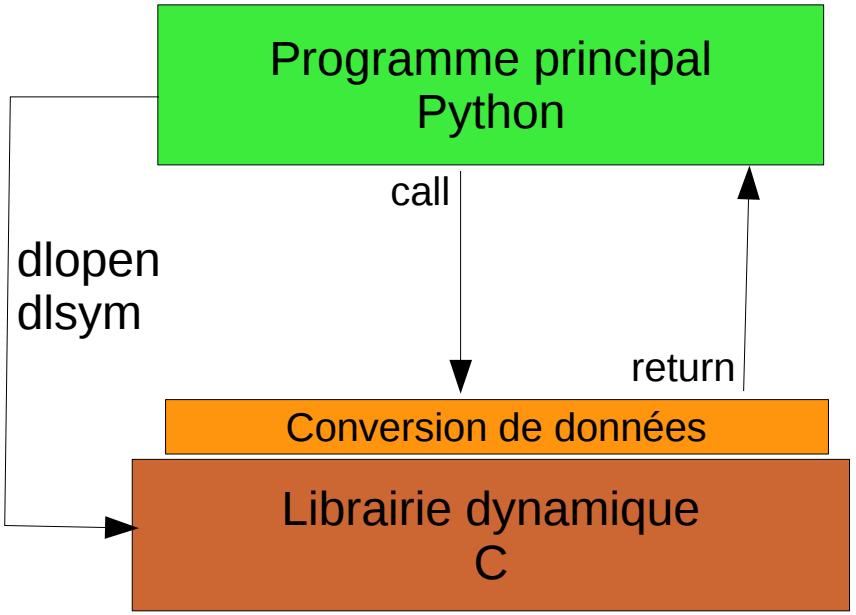
Python appelle C

- Solution classique: sous-processus
- Facile à implémenter
- Échange de données:
  - ligne de commande
  - fichiers/pipes,
  - mémoire partagée
- Overhead: appel système
  - fork / exec
  - read / write



API Python-C

- Module = librairie dynamique (.so, .dll, .dylib)
  - fonctions, prototypes standards
- Pas d'overhead système
- Échange de données
  - Même espace mémoire
  - Conversion de types de données:
    - `PyObject ↔ char*, int...`
    - Fastidieux...




```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

Comment simplifier ?

Faciliter l'intégration Python-C

- ctypes : appel d'une fonction arbitraire dans un .so
  - +: intégré dans librairie standard python, rien à compiler
  - -: verbeux si on a besoin de beaucoup de fonctions
- cython : code Python compilé en C
  - +: migration Python → C facile
  - -: redéclaration des fonctions à appeler
- scipy.weave: code C inline en Python
  - +: facile et transparent pour boucles simples sur des matrices
  - -: difficile à débugger
- SWIG: voir la suite

Motivation

Passerelle Python-C

SWIG

Simple Wrapper and Interface Generator

- Principe:
  - Parse les définitions des fonctions C (.h)
  - génère le code d'appel et de conversion des données
- Exemple:

test_toto.py

```
import toto

print toto.plus(1, 2)
```

toto.c

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

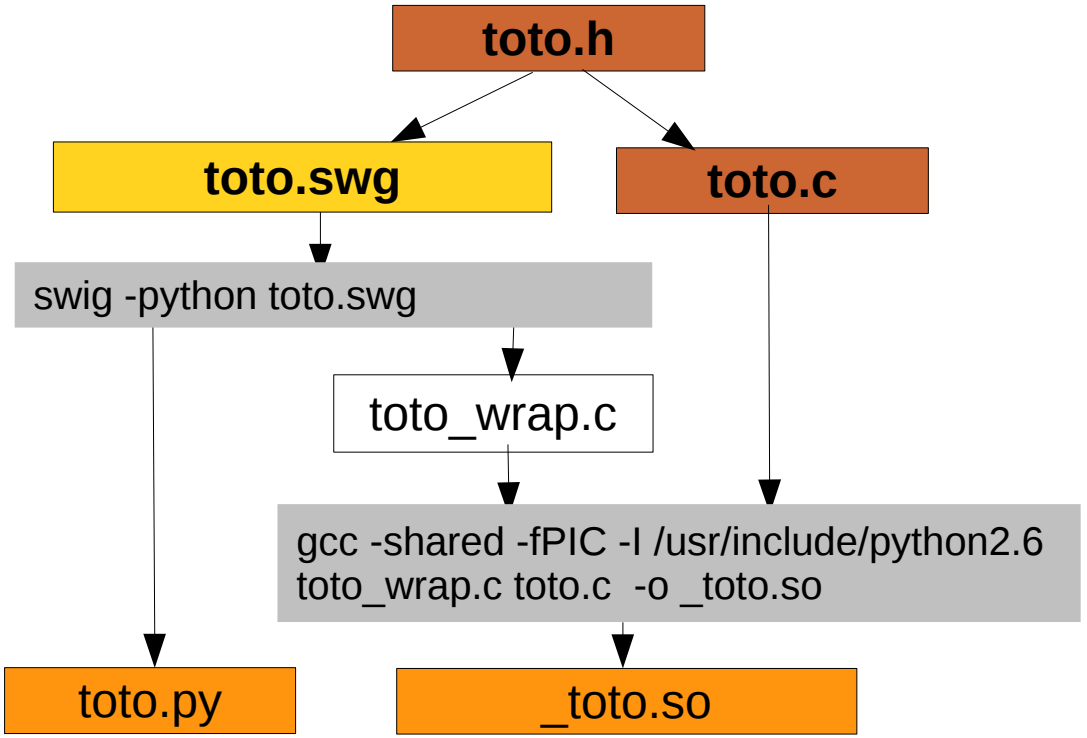
- Fichier SWIG:
  - Nom du module en Python
  - Copié tel quel dans le C généré
  - Déclarations à rendre visibles

toto.swg

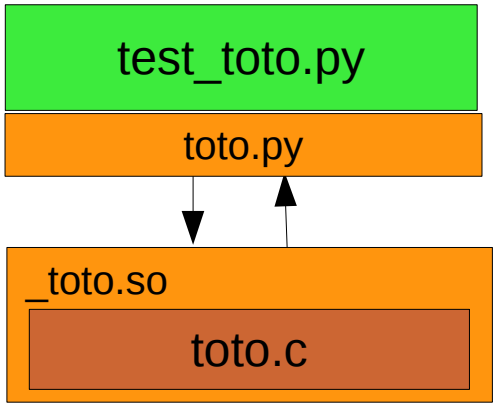
```
%module toto;
%{
#include “toto.h”;
%}
%include “toto.h”;
```

Compilation & exécution

- Compilation:
  - à mettre dans un Makefile



- Exécution:



Conversions de données : cas simples

- Types de base (int, float, …): ça marche
  - cas particulier: string python → `const char *`
- `struct` (et `class` en C++)
  - Mappés sur des objets Python

test_titi.py

```
import titi
a = titi.A()
a.x = 1
a.f(10)
print a.x
```

titi.cpp

```
struct A {
  int x;
  void f(int y) { x += y; }
};
```

  - Constructeur/destructeur: appelé par le garbage collector
- Pointeurs
  - SWIG ne sait pas les déréférencer : types opaques
  - Définir des accesseurs `float float_array_get(float *x, int i);`
    - librairies de macros SWIG (`carrays.i`)...

Autres cas: typemap

- Customisation des conversions
  - Arguments d'entrée et de sortie
  - Exceptions...
- Exemple: %typemap(argout)
  - L'argument `int *mod` en entrée...
  - ...correspond à 0 arguments Python
  - utiliser une variable temporaire à la place
  - En sortie...
  - ...construire un couple résultat, combinant le résultat initial avec la valeur temporaire.
- En Python:

tata.c

```
int divmod(int a, int b, int *mod) {
  *mod = a % b;
  return a / b;
}
```

tata.swg

```
%module tata;
%{
#include “tata.h”;
%}

%typemap(in, numinputs=0) int *mod (int temp) {
   $1 = &temp;
}
%typemap(argout) int *mod {
  $result = Py_BuildValue("(Ni)", $result, *$1);
}
%include “tata.h”;
```

```
toto $ python
>>> import tata
>>> tata.divmod(40,6)
(6, 4)
```

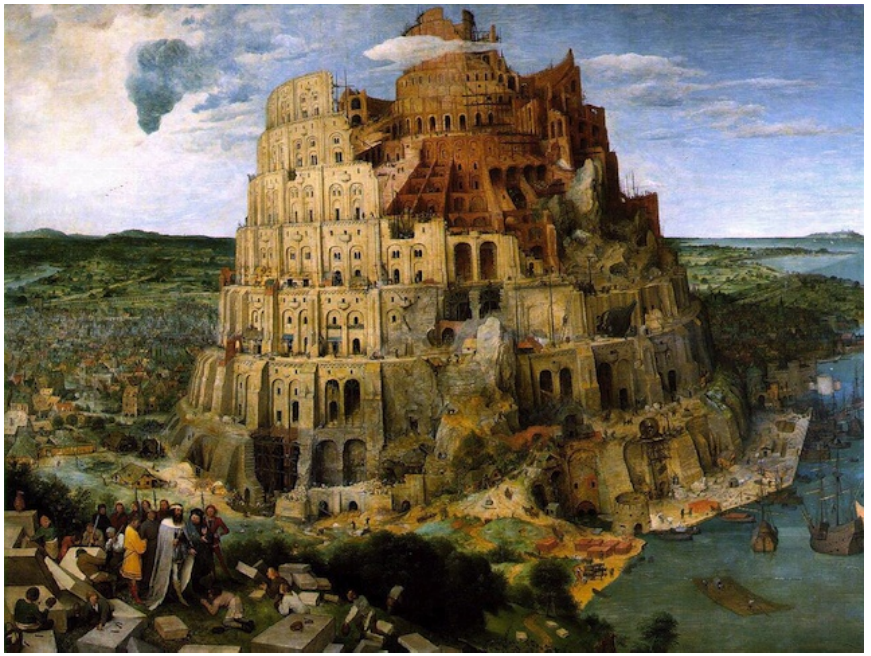
98 %

Pour aller plus loin

- Principe: choix du niveau d'intégration
  - Par défaut → conservatif: peu de conversions automatiques
    - types opaques
  - Transistion douce pour gros codes existants
- Features supportés:
  - C++:
    - Templates: instanciation explicite
    - Surcharge opérateurs: transparent
    - Exceptions: les transformer en exceptions Python (`%except`)
  - Multithread
    - Threads Python (`thread.start_new_thread`)
  - Intégration avec le garbage collector

100%

Conclusion



- Choisir le langage approprié
  - Le langage universel n'existe pas
  - Gros logiciel = plusieurs langages
- SWIG
  - stable
  - ~20 langages de script
    (ocaml, java, perl, ruby...)
  - Transition facile pour gros codes C

Questions ?