Interopérabilité

TP Python - C++

Python embarqué

Eric Ramat
ramat@lil.univ-littoral.fr

2 mai 2008

Durée : 3 heures

---

1 Objectif

L'objectif de ce TP est d'embarquer un langage de scripts au sein d'une application écrite en C++. Le langage de scripts que nous allons utiliser sera Python. L'écriture du code sera piloté par les tests unitaires. La suite du sujet est donc présentée sous forme de cartes d'histoire.

2 Cahier des charges

Le développement doit se faire au fur et à mesure des besoins du client et on doit se limiter **strictement** à ces besoins. Le code est à écrire en C++ et on utilisera le framework boost : :unit_test_framework.

Comme tout le monde le sait, les besoins du client évoluent constamment, nous allons donc présenter ces besoins sous forme d'un scénario en plusieurs étapes (histoire). Chaque histoire doit donner naissance à des tests et au code répondant aux tests.

2.1 Premier scénario

2.1.1 Première histoire

Le client demande le résultat de "I + I" au programme et ce dernier réponse : "II". Pour simplifier, le client aimerait que la chaîne de caractères "I + I" soit transformée en "1 + 1" et qu'un interpréteur Python évalue l'expression. Le résultat de cette évaluation est alors convertie en "II".

Indications :
- la librairie Boost dispose d'un algorithme split ;
- le type map existe dans la librairie STL de C++ pour la représentation des dictionnaires.

2.1.2 Deuxième histoire

Dans un même appel, le client aimerait évaluer plusieurs expressions. Il fait donc appel au programme avec "I + V ; III + IX" et il lui retourne "VI ; XII".

2.1.3 Troisième histoire

Le client veut compliquer les expressions et introduit à cet effet la notion de variables. Il propose l'expression suivante : "a = I + II ; b = IV + III ; c = a + b". En retour, le client aimerait connaître la valeur (en chiffres romains) des différentes variables sous la forme "a = III ; b = VII ; c = X"

Si le signe "=" n'est pas présent alors le client veut connaître seulement la valeur des différentes expressions.

2.2 Deuxième scénario

2.2.1 Première histoire

Le client veut écrire un programme en Python qui travaille sur un ensemble d'objets représentants des formes géométriques écrites en C++. Tous ces types de formes géométriques héritent d'une super-classe Shape dont voici le code :

```
class Shape {
  public:
    Shape() {}
    virtual ~Shape() {}
    virtual double computeArea() const =0; // méthode abstraite
};
```

Le client propose une sous-classe concrète de cette classe :

```
class Circle : public Shape {
  private:
    double rayon;

  public:
    Circle(double r):rayon(r) { }
    virtual ~Circle() {}
    virtual double computeArea() const { return 3.1415*rayon*rayon; }
};
```

Afin de tester ces idées, on peut écrire une fonction Python qui, à partir d'un ensemble d'objets de type Shape, calcule la somme des surfaces des objets.

2.2.2 Deuxième histoire

Le client veut ajouter une nouvelle classe C++ de formes géométriques : les carrés. Il faut construire un ensemble d'objets incluant des carrés.

2.2.3 Troisième histoire

Le client devient difficile et désire propose une possibilité de développement de sous-classes de Shape en **Python**. On veut faire appel à la même fonction somme_aire mais maintenant cette fonction est écrite en C++ et est basée sur un vector de pointeurs sur des objets de type Shape (std : :vector<Shape*>). De plus, une nouvelle classe d'objets géométriques, triangle, par exemple, est écrite en Python.

Rappel : un triangle est défini par la longueur de sa base et sa hauteur ; l'aire est égale à la moitié du produit des deux.

3 Annexe

Ce petit programme lance l'interpréteur Python et exécute une commande simple.

Voici le code :

```
#include <python2.4/Python.h>

int main() {
  Py_Initialize();
  PyRun_SimpleString("print 'coucou'");
  Py_Finalize();
}
```

et la ligne de compilation :

```
# gcc -lpython2.4 -lm -L/usr/lib/python2.4/config -o essai essai.c
```

L'API Python/C permet aussi de demander à Python d'exécuter une commande et d'en récupérer le résultat. Voici un exemple :

```
#include <python2.4/Python.h>
#include <iostream>

int main() {
  Py_Initialize();

  PyObject* main = PyImport_AddModule("__main__");
  PyObject* dict = PyObject_GetAttrString(main,"__dict__");

  PyObject* p = PyRun_String("1+1",Py_eval_input,dict,dict);

  if (PyInt_Check(p)) // vérification du type de retour
    std::cout << "result = " << PyInt_AsLong(p) << std::endl;

  Py_Finalize();
}
```

Si vous voulez exécuter un script Python complet et récupérer la valeur de variables manipulées dans le script, en voici un exemple :

```
#include <python2.4/Python.h>
#include <iostream>

int main() {
  Py_Initialize();

  PyObject* main = PyImport_AddModule("__main__");
  PyObject* dict = PyObject_GetAttrString(main,"__dict__");

  PyObject* p = PyRun_String("a=1+1",Py_file_input,dict,dict);
  PyObject* a = PyObject_GetAttrString(m,"a");

  if (PyInt_Check(a))
    std::cout << "a = " << PyInt_AsLong(a) << std::endl;

  Py_Finalize();
}
```

3.1 Références

- STL : http://www.sgi.com/tech/stl/
- Boost : http://www.boost.org

– Python : http://www.python.org/
– Python/C API : http://www.python.org/doc/2.4/api/api.html