4V748
Faire communiquer Python avec d'autres langages en particulier C, R, SQL, Java

4V748

Enseignement Supérieur Public

02/04/2014

Des langages

... c'est pas ce qui manque !

Plus de 400 langages de programmation
(`http://www.scriptol.fr/programmation`)

Chacun a ses + et ses -

Nom	Type	Paradigme	Domaine	Exec.	Appr.	Lisib.
Bash	Interprété	Procédural	Système	-	+	-
C	Compilé	Procédural	Généraliste	++	-	-
C++	Compilé	Objet	Généraliste	++	- -	-
Haskel	Bytecode	Fonctionnel	Généraliste	-	-	+
Java	Bytecode	Objet	Généraliste	+	-	+
Mathematica	Interprété	Fonctionnel	Sciences	+	-	+
PHP	Interprété	Objet	Web	- -	+	+
Python	Interprété	Objet	Généraliste	-	+	++
R	Interprété	Objet	Statistiques	-	-	-
SQL	Interprété	Relationel	BdD	-	-	+

N.B. contient quelques appréciations subjectives

Que manque-t-il à Python ?

Quelques années

- C : 1972
- Python : 1991

Que manque-t-il à Python ?

Quelques années

- C : 1972
- Python : 1991

Conséquence : Plus de bibliothèques spécialisées en C qu'en Python

Que manque-t-il à Python ?

Quelques années

- C : 1972
- Python : 1991

Conséquence : Plus de bibliothèques spécialisées en C qu'en Python

Du graphisme spécialisé

- R : `plot(data)` $\rightarrow$ et hop, un graphe !
- Python : ...

Que manque-t-il à Python ?

Quelques années

- C : 1972
- Python : 1991

Conséquence : Plus de bibliothèques spécialisées en C qu'en Python

Du graphisme spécialisé

- R : `plot(data)` $\rightarrow$ et hop, un graphe !
- Python : ...

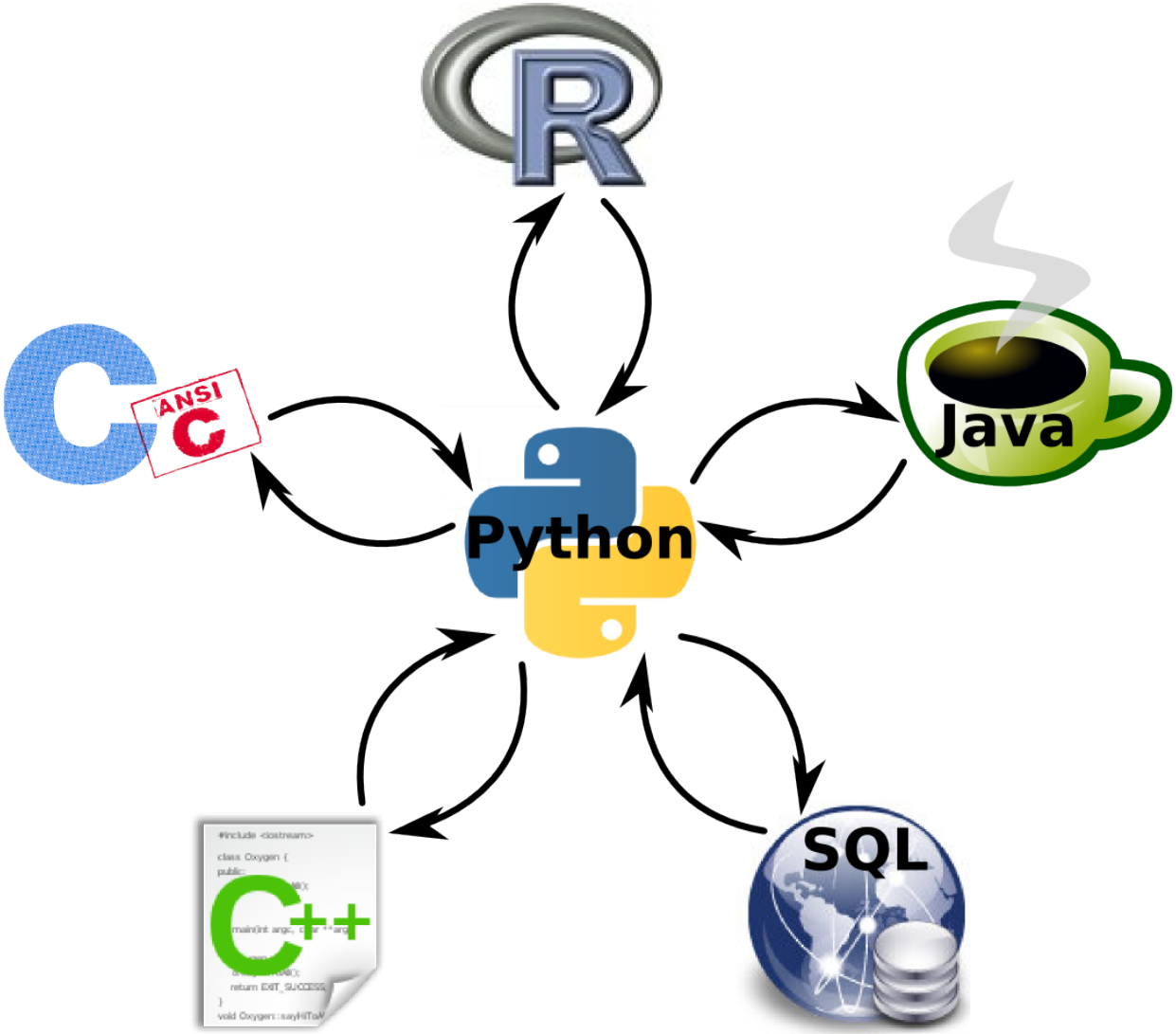
On peut tout faire en Python, mais c'est parfois compliqué

Si ça existe ailleurs...

Mark Parisi

... pourquoi ne pas l'utiliser ?

Que veut-on faire ?



Comment s'y prendre ?

On peut imaginer

- permettre à Python d'interpréter du code écrit en X
- appeler en Python des fonctions qui ont été écrites en X
- utiliser des instructions Python dans du code en X
- traduire du code Python en X

Question :
laquelle de ces stratégies a été mise en œuvre ?

Réponse :
toutes

Différentes méthodes

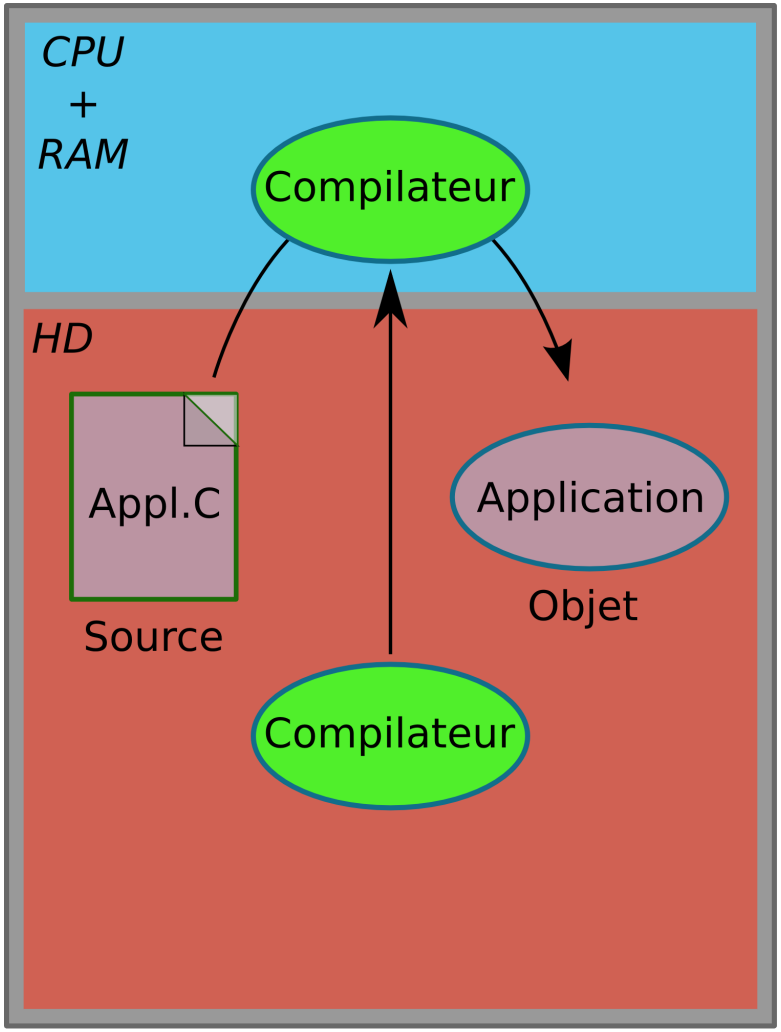
Selon

- le type du langage avec lequel dialogue Python
- l'utilisation souhaitée
- le temps de développement accordé
- les performances espérées
- l'évolutivité attendue

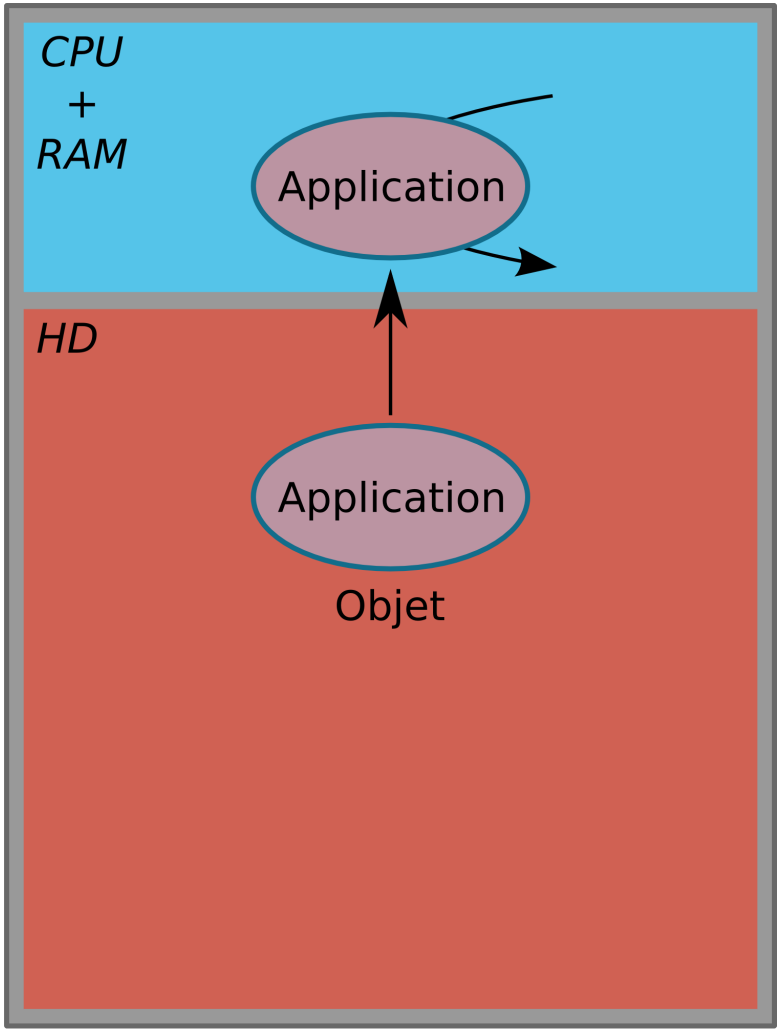
Sommaire

1. Langages compilés
   - Ecrire une extension
   - Conversion de code
   - Code incorporé
   - Enrichissement du C
2. Langages interprétés
   - Code R incorporé
   - Accès SQL avec un curseur
3. Langages à bytecode
   - Compilation dynamique

Compilation

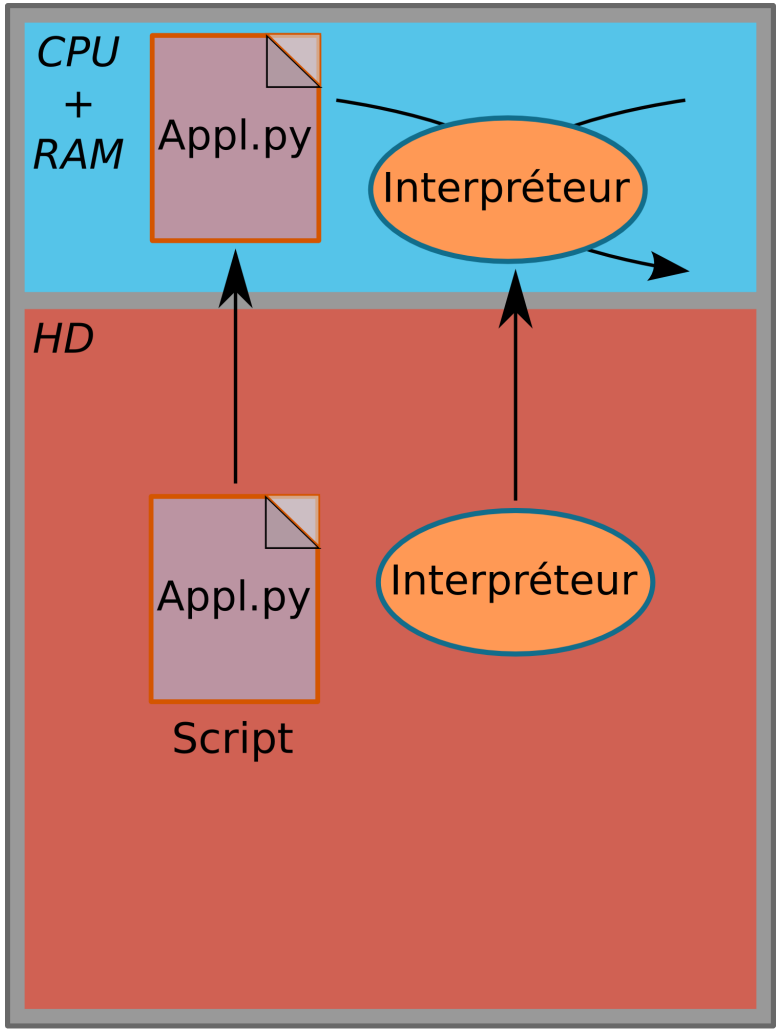


t=0 : compilation



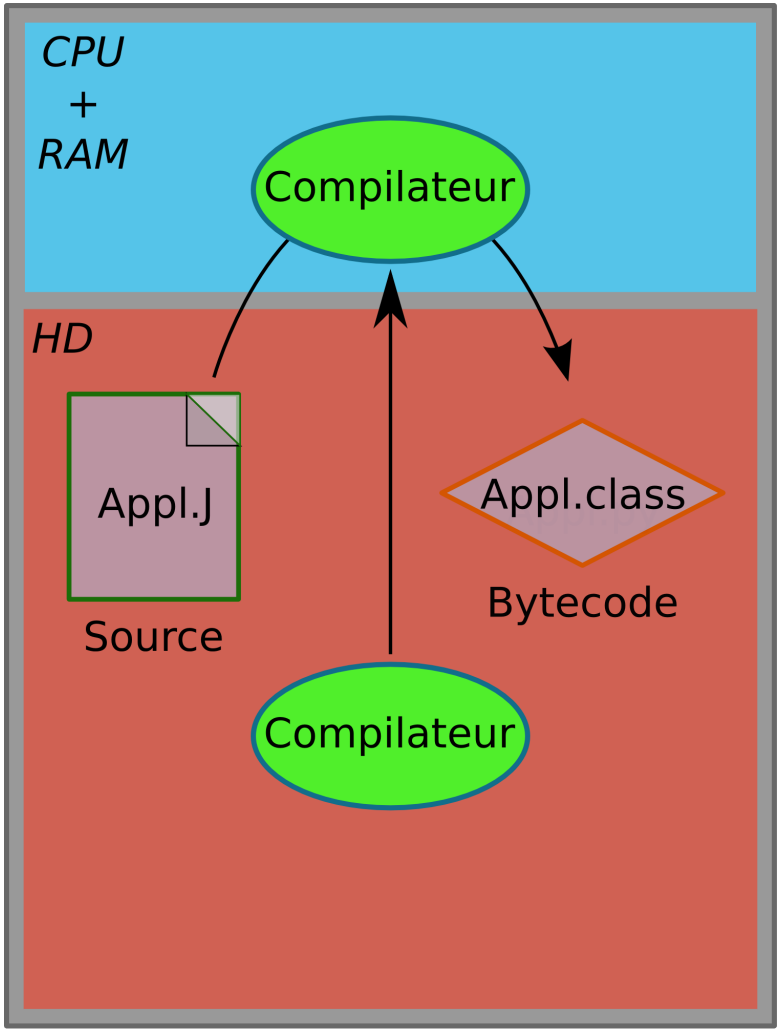
t>0 : exécution

Interprétation

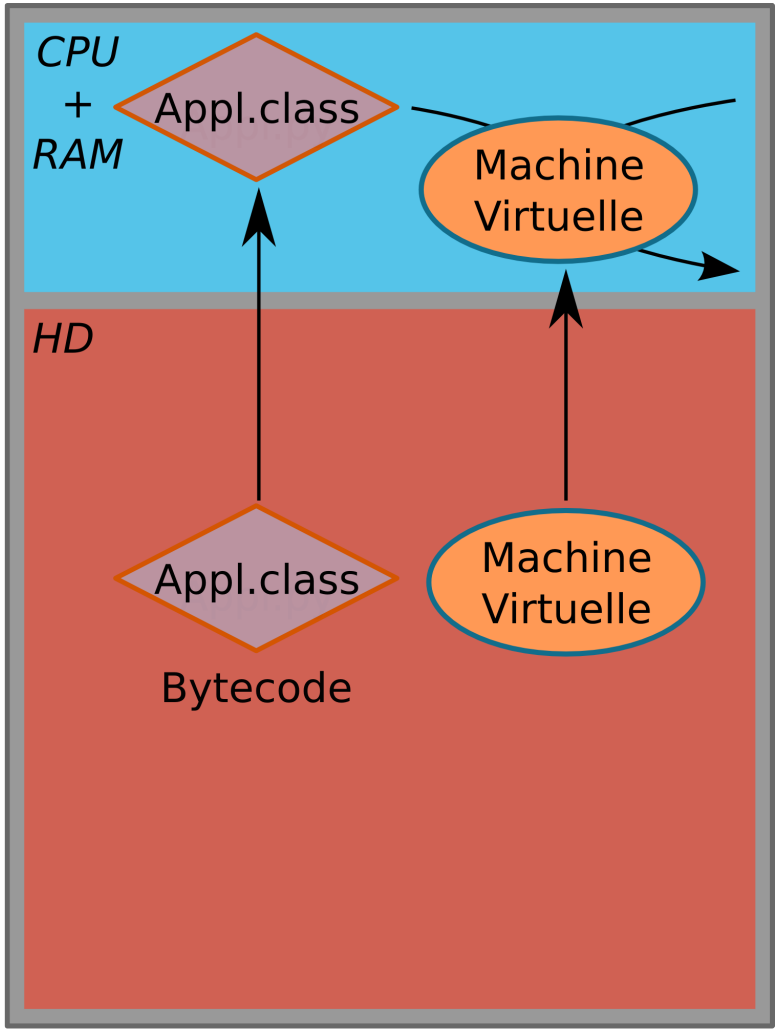


interprétation

Bytecode



t=0 : compilation



t>0 : exécution

Module d'extension (C++)

Principe

Ecrire des fonctions C++ en utilisant des objets spécifiques permettant de communiquer avec du code en Python

Méthode

- Ecrire du code en C++
- Dans ce code, utiliser des objets d'interface avec Python
- Compiler pour obtenir un module Python
- Ecrire du code Python qui appelle les méthodes du module

L'*API* Python en C++

```
#include <Python.h>
```

Définit

- objets
- fonctions
- macros
- variables

spécifiquement dédiées à la communication avec Python

Exemple de fonction

4V748module.cpp

```
#include <Python.h>
...

static PyObject *4V748_fct(PyObject *self, PyObject *args) {

  const char *ch;
  int fctres;
  ...                                   // autres declarations

  if (!PyArg_ParseTuple(args, "s", &ch)) // recupere les arguments
      return NULL;

  ...                                   // instructions diverses

  ... any_cpp_function(ch);             // fonction quelconque

  ...                                   // instructions diverses

  fctres = another_cpp_function(...);   // valeur de retour

  return Py_BuildValue("i", fctres);    // empaquetage pour Python

}
```

Exemple (suite)

Il faut aussi inclure la méthode dans une table et prévoir son initialisation

4V748module.cpp

```
#include <Python.h>
...

static PyMethodDef 4V748Methods[] = {
  ...
  {"fct",  4V748_fct, METH_VARARGS, "Demo pour 4V748"},
  ...
  {NULL, NULL, 0, NULL}                // Sentinelle
};

PyMODINIT_FUNC init4V748(void) {       // appelee lorsqu'on importe 4V748

  (void) Py_InitModule("4V748", 4V748Methods);

}

static PyObject *4V748_fct(PyObject *self, PyObject *args) {
  ...
}
```

Plus de détails sur http://docs.python.org/extending/extending.html

Compilation du module par distutils

L'utilitaire distutils est installé avec Python
Il fonctionne avec un fichier de configuration

setup.py

```
from distutils.core import setup, Extension

module1 = Extension('4V748', sources = ['4V748module.cpp'])

setup (name = '4V748_utils', version = '1.0',
       description = 'Demo pour 4V748',
       ext_modules = [module1])
```

Création du module

```
[prompt] python setup.py build
```

génère le fichier build/lib.system/4V748.pyd

Utilisation du module

testappli.py

```
import 4V748
...
res = 4V748.fct("une chaine")
```

Le fait que `4V748.fct` soit écrite en C++ est transparent

Destiné à développer de nouvelles bibliothèques

Distribuer le module

`distutils` permet aussi de générer un package

```
[prompt] python setup.py sdist
```

génère le fichier `4V748-1.0.tar.gz`, qui contient `setup.py` et `4V748.py`

Pour installer

```
[prompt] tar zxf 4V748-1.0.tar.gz
[prompt] cd 4V748-1.0
[prompt] python setup.py install
```

Conversion (C/C++)

Principe

Convertir des fonctions C/C++ en un module Python

Méthode

- Prendre du code écrit en C/C++
- Convertir pour obtenir un module Python
- Ecrire du code Python qui appele les méthodes du module

Exemple de fichier *header* en C

Contient les prototypes des fonctions

4V748_func.h

```
#include <string.h>
char *complinv(char *seq);
char *tradorf(char *seq, char *code);
int posmotif(char *mot, char *seq);
```

N.B. il n'est pas obligatoire de fournir les noms des arguments, le type suffit

Par convention, les fonctions sont écrites dans `4V748_func.c`

Conversion par SWIG

Pour convertir, il faut écrire un fichier d'interface

4V748_func.i

```
%module 4V748_func
%{
#include <string.h>
%}
char *complinv(char *seq);
char *tradorf(char *seq, char *code);
int posmotif(char *mot, char *seq);
```

SWIG génère le code pour la communication entre chaque fonction et Python

```
[prompt] swig -python 4V748_func.i      # cree le fichier 4V748_func_wrap.c
```

Compilation

```
[prompt] gcc -fpic -c 4V748_func.c 4V748_func_wrap.c -I/usr/local/include/python2.1
[prompt] ld -shared 4V748_func.o 4V748_func_wrap.o -o _4V748_func.so
```

Utilisation du module

restrimap.py

```
import 4V748_func
...
pos = 4V748_func.posmotif("GAATTC",seq_insert)
```

Le fait que `4V748_func.posmotif` soit écrite en C est transparent

Permet de convertir une bibliothèque pré-existante, sans en changer le code
Repose sur `Python.h`

Code C incorporé

Principe

Incorporer dans le code Python une fonction C, qui est compilée à la volée lors de l'interprétation du code.

Méthode

- Ecrire du code en Python
- Insérer une fonction C comme argument d'une fonction Python : `build`

Incorporation de code C

kodo.py

```
import PyInline, __main__

PyInline.build(code="""
  #include <stdio.h>
  #include<string.h>
  int encode(char *co){
    int p,c,k=0;
    char *al="ACGT";
    for(c=0,p=16;c<3;c++,p/=4)
      k+=(strchr(al,co[c])-al)*p;
    return k;
  }
  """, targetmodule=__main__, language="C")

print encode("TGA")
```

Utilisation simple : ni fichier supplémentaire, ni compilation spécifique
Destiné à optimiser un petit nombre de fonctions, plutôt qu'une bibliothèque

Dans la même famille

... que SWIG : utilisation de bibliothèques C/C++

Boost.Python bibliothèque C++. Modérément portable.

ctypes module Python standard. Fonctionne sans fichier supplémentaire (interface), mais il faut traduire les types complexes (`struct`, ...). C mais pas C++.

... que PyInline : utilisation de C pour accélérer l'exécution

Pyrex Ajoute les types C (statiques) à Python et génère du C.

Cython dérivé de Pyrex, plus complet.

Enrichissement du C/C++

Principe

L'*API* Python du C/C++ permet au compilateur C/C++ de compiler du code mixte écrit en C/C++ et en Python

Méthode

- Ecrire un programme en C/C++
- Incorporer des objets, du code, des modules Python
- Compiler le tout

Incorporation de code Python

exepy.c

```
#include <Python.h>
...
  Py_Initialize();
  PyRun_SimpleString("Code Python");        # Executer du code explicite
  PyRun_SimpleFile("UneAppli.py");          # Executer un source Python
  Py_Finalize();
```

Il est également possile de lancer l'interpéteur interactif

interpy.c

```
#include <Python.h>
...
void python_inter(int arg, char** argv) {
  Py_Initialize();
  Py_Main(argc, argv);
  Py_Finalize();
}
```

Code R incorporé

Principe

Ecrire du code Python, et transmettre à R les données et instructions spécifiques

Méthode

- Ecrire du code en Python en important le module choisi
- Transmettre une à une les instructions R comme arguments d'une méthode dédiée. Cette méthode se charge de :
  - traduire les données du programme Python en types R
  - appeler les fonctions R sur ces données
  - restituer les résultats sous formes d'objets Python

Pour exécuter des instructions R depuis Python : Rpy

Rpy définit l'objet `r` pour faire communiquer Python et R
Il suffit (presque) de

- préfixer les fonctions avec `r.`
- remplacer les `.` par des _
- remplacer les $ par des `['']`

freqs.R

```
frq=c(5,3,8,12,7)
cnt=round(rnorm(length(frq),mean=frq)*20)

X=chisq.test(cnt,p=frq,rescale.p=TRUE)
print(X$p.value)
```

freqs.py

```
from rpy import *
frq=r.c(5,3,8,12,7)
tmp=r.rnorm(r.length(frq),mean=frq)
cnt=r.round([c * 20 for c in tmp])
X=r.chisq_test(cnt,p=frq,rescale_p=r.TRUE)
print X['p.value']
```

NB les vecteurs de R deviennent des listes en Python
Impossible alors de multiplier les éléments avec la syntaxe `v2 = n * v1`

Pour exécuter des instructions R depuis Python : Rpy2

En Rpy2, l'objet utilisé pour communiquer est `robjects.r`
A noter

- La syntaxe `robjects.r['fonction'](arguments)` (existe aussi en Rpy)
- `TRUE` devient `True`

freqs.R

```
frq=c(5,3,8,12,7)
cnt=round(rnorm(length(frq),mean=frq)*20)

X=chisq.test(cnt,p=frq,rescale.p=TRUE)
print(X$p.value)
```

freq2.py

```
import rpy2.robjects as robjects
r=robjects.r
frq=r.c(5,3,8,12,7)
tmp=r.rnorm(r.length(frq),mean=frq)
cnt=r.round(tmp.ro*20)
X=r['chisq.test'](cnt,p=frq,rescale_p=True)
print X[2]
```

NB L'attribut `.ro` extrait l'objet R
On peut multiplier les éléments d'un vecteur avec `v2 = n * v1.ro`

Pour exécuter des instructions R depuis Python : PypeR

le module pypeR définit l'objet R

freqs.R

```
frq=c(5,3,8,12,7)
cnt=round(rnorm(length(frq),mean=frq)*20)
X=chisq.test(cnt,p=frq,rescale.p=TRUE)
print(X$p.value)
```

freqP.py

```
from pyper import *
r = R()
r.frq=[5,3,8,12,7]
r("cnt=round(rnorm(length(frq),mean=frq)*20)")
r("X=chisq.test(cnt,p=frq,rescale.p=TRUE)")
print r('X$p.value')
```

En cas de grosse fatigue

... et si les données peuvent être générées ou chargées par R lui-même

Rpy

```
from rpy import *
r.source('freqs.R')
```

Rpy2

```
import rpy2.robjects as robjects
robjects.r.source('freqs.R')
```

PypeR

```
from pyper import *
print runR("source('freqs.R')")
```

Dans la même famille

Mais avec une syntaxe différente

PyRseve utilise un mécanisme client/serveur

RSPIus permet aussi d'exécuter du code Python dans R

Accès SQL avec un curseur

Principe

Se connecter depuis Python à une base SQL pour accéder à son contenu en lecture et écriture *via* du code passé en argument à une méthode

Méthode

- Créer la base avec un SGBD SQL (PostgreSQL, MySQL, ...)
- Dans le code Python, importer le module adéquat (`psycopg2`, `MySQLdb`, ...)
- se connecter à la base et créer une instance de la classe `cursor`
- Utiliser la méthode `execute` de l'objet `cursor` pour
  - effectuer des requêtes de type `SELECT`
  - traiter en Python le résultat de ces requêtes
  - modifier le contenu de la base, par exemple avec `INSERT`

Création d'une base, puis accès depuis Python

```
[prompt] createdb labase
[prompt] psql labase
labase=# CREATE TABLE ... (...)
labase=# INSERT INTO ... (...) VALUES (...)
labase=# ...
labase=# \q
```

AccesBdD.py

```
import psycopg2;
conn = psycopg2.connect("dbname=labase")                    # Connexion
c = conn.cursor()                                           # Curseur

c.execute("SELECT ... FROM ... WHERE ...")                  # Consultation
res = c.fetchall()
...                                                         # Traitement

c.execute("INSERT INTO ... VALUES  (?)", (...))             # Ecriture
conn.commit()                                               # Validation

c.close()                                                   # Fini
```

Base créée depuis Python, et accès

CreeAccedeBdD.py

```
import gadfly;
conn = gadfly.gadfly()                                  # Initialisation
conn.startup("labase", "chemin")                        # Creation de la base

c = conn.cursor()                                       # Curseur

c.execute("CREATE TABLE ... (...)")                     # Nouvelle table
c.execute("INSERT INTO ... (...) VALUES (...)")         # Entree de la table

c.execute("SELECT ... FROM ... WHERE ...")              # Consultation
res = c.fetchall()
...                                                     # Traitement

c.execute("INSERT INTO ... VALUES  (?)", (...))         # Ecriture
conn.commit()                                           # Validation

c.close()                                               # Fini
```

Les sessions interactives sont aussi possibles, en utilisant le programme `gfplus.py`

Dans la même famille

... que psycopg2, selon la base

MySQLdb pour MySQL

cx_Oracle pour Oracle

sqlite3 pour SQLite3 (standard avec Python)

... que Gadfly

APSW pour SQLite3

Réimplémentation de Python

Principe

Ecrire du code Python, utilisant éventuellement des classes Java, et le faire exécuter par une Machine Virtuelle Java (*JVM*)

Méthode

- Ecrire du code en Python
- Dans ce code, utiliser si besoin des classes Java
- Pour faire tourner le programme, deux possibilités au choix :
  - Exécuter le code dans l'interpéteur Jython
  - Compiler le code et l'exécuter dans une *JVM*

Utilisation d'une classe Java depuis Python

Fenetre.jy

```
import javax.swing as swing

win = swing.JFrame("Jython 4V748")
win.size = (600, 600)
win.show()

field = swing.JTextField(preferredSize=(600,50))
win.contentPane.add(field)
win.pack( )
...
```

Jython exécute du code Python standard, et peut importer directement des classes Java

Quel usage ?

Faire tourner le code dans une *JVM*

- sur une architecture ne possédant pas d'interpréteur Python
- sous forme d'*applet*, par exemple dans un navigateur *web*

Bénéficier de classes Java

- peu ou pas supportées par Python : JDBC...
- plus rapides que leurs équivalents Python

Dans la même famille

SPIRO permet d'utiliser n'importe quelle classe Java

JPype permet à Python d'ouvrir une *JVM* Java

... inversement

Jepp pour exécuter du code Python depuis Java

... et les deux

JPE pour utiliser Java et Python ensemble

Conclusion

Python peut déjà faire beaucoup de choses tout seul

De plus, il peut bénéficier des apports de nombreux autres langages