# Distutils et Scons

Xavier Juvigny

ONERA/CHP

Décembre 2010

# Plan

# Plan

# Plan

# Présentation générale

Le package `distutils` permet :

- D'écrire un script de production et d'installation en Python (appelé généralement `setup.py`) ;
- En option, d'écrire un fichier de configuration pour la production ;
- De créer une distribution des sources ;
- En option, de créer une ou plusieurs distributions binaires.

Quelques limitations cependant :

- Les modules Python doivent être en C, C++ ou en Python ;
- Compilation de fichiers Fortran (ou d'un autre langage que C ou Python) pas immédiat ;
- Pas de gestion de dépendance automatique pour les sources autres que Python, C et C++.

# Un premier exemple

But : Distribuer le module Python `MonModule` mis en œuvre dans le fichier `MonModule.py` :
Le fichier `setup.py` :

```
from distutils.core import setup
setup(name="MonModule",
      version='1.0',
      py_modules=['MonModule']
      )
```

Quelques remarques :

- La plupart des informations données dans `setup.py` sont données comme mot-clefs à la fonction `setup`.
- Deux catégories de mot-clefs :
  1. Les meta-informations : nom du module, de l'auteur, numéro de version, email de l'auteur, page web du projet,. . .
  2. Le contenu du paquet : ici une liste de fichiers python.
- Nom du module donné par son nom, pas par son fichier ;

# Utilisation du `setup.py`

À partir du fichier créé, plusieurs actions possibles :

1. Création distribution des sources (archivage fichiers) :

```
python setup.py sdist
```

2. Installation de la distribution :

```
python setup.py install --prefix=/usr/local
```

3. Création exécutable d'installation windows :

```
python setup.py bdist_wininst
```

4. Création fichier rpm (pour unix) :

```
python setup.py bdist_rpm
```

5. Connaître les formats de distributions possibles :

```
python setup.py bdist --help-formats
```

# Plan

# Distribution de packages

Possible d'installer modules pythons par paquets (package) :

- Pour chaque package, un répertoire avec ses fichiers python ;
- Pour chaque répertoire contenant un package, un fichier `__init__.py` (même vide !) d'initialisation du package ;
- Si répertoires même noms que packages :

```
from distutils.core import setup
setup(name="MonModule",
      version='1.0',
      packages=['UnPackage', 'UnAutrePackage']
     )
```

- Sinon, correspondance. Exemple :

```
from distutils.core import setup
setup(name="MonModule",
      version='1.0',
      package_dir = {'':'src','UnPackage':'src/Rep2'},
      packages=['UnPackage', 'UnAutrePackage']
     )
```

# Installation de modules de divers packages

Possibilité vision par module : spécifier le package auquel appartient chaque module :

```
from distutils.core import setup
setup(name="MonModule",
      version='1.0',
      py_modules=['UnPackage.MonModule',
                  'UnAutrePackage.UnAutreModule']
     )
```

Implique existance d'un fichier `__init__.py` dans le répertoire `UnPackage` et dans le répertoire `UnAutrePackage`.
On peut, là encore, spécifier la correspondance entre les packages et les répertoires à l'aide de l'option `package_dir`.

# Autres possibilités

Possibilité de gérer des fichiers autres que modules ou packages :

- Exécuter des scripts pythons :

```
setup(...,
      scripts=['scripts/run_tests','scripts/comp_fortran'])
```

- Installer des données associées à un package :

```
setup(...,
      packages=['mypkg'],
      package_dir={'mypkg': 'src/mypkg'},
      package_data={'mypkg' : ['data/*.dat']},)
```

- Installer des données associées à la distribution :

```
setup(...,
      data_files=[('bitmaps',['bm/bill.gif','bm/boul.gif']),
                  ('config', ['cfg/data.cfg']),
                  ('/etc/init.d', ['init-script'])] )
```

# Plan

# Installer un module écrit en API C

Énumérer les fichiers C (comme les fichiers Python) ne suffit pas !
On doit utiliser le sous-paquet `Extension`.
Si le module `MonModule` n'a besoin que d'un seul fichier C `MonAPI.c` :

```
from distutils.core import setup, Extension

monmod = Extension('MonModule', ['MonAPI.c'])
setup(name='MonModule',
      version='1.0',
      ext_modules=[monmod],
      )
```

Les fichiers C++ sont reconnus à l'aide de leurs extensions (`.cc` ou `.cpp`).

# Extensions et Packages

Le premier argument du constructeur `Extension` et le nom du module, dont le nom du package le contenant. Par exemple :

```
monmod = Extension('MonModule', ['MonAPI.c'])
```

définit un module se trouvant dans le package racine, et

```
monmod = Extension('MonPackage.MonModule', ['MonAPI.c'])
```

définit un module se trouvant dans le package `MonPackage`. Si plusieurs extensions se trouvent dans le même package, on peut utiliser l'option `ext_package` :

```
from distutils.core import setup, Extension

setup(...,
      version='1.0',
      ext_package='MonPackage',
      ext_modules=[Extension('Module1', ['Module1.c']),
                   Extension('subpkg.Module2', ['Module2.c'])],
     )
```

# Options du préprocesseur

Trois options pour le préprocesseur peuvent être passées aux extensions :

- `include_dirs` : Pour donner le chemin des fichiers d'entêtes C/C++ en relatif (par rapport à la racine de la distribution) ou en absolu :

```
md1 = Extension('mod1', ['mod1.c'],include_dirs=['include'])
md2 = Extension('xmod2', ['xmod.c'],
                include_dirs=['/usr/include/X11'])
```

On peut aussi automatiser la recherche d'un fichier d'entête provenant d'une extension Python :

```
import os
from distutils.sysconfig import get_python_inc
incdir=os.path.join(get_python_inc(plat_specific=1),'numpy')
setup(...,
      Extension(..., include_dirs=[incdir]),
     )
```

# Options du préprocesseur...

- Définition ou suppressions de macros :

```
Extension(...,
          define_macros=[('NDEBUG',1),('TRACE',None)],
          undef_macros =['USE_MEMORY_DEBUG'] )
```

ce qui est équivalent à inclure dans tous les fichiers C :

```
#define NDEBUG 1
#define TRACE
#undef  USE_MEMORY_DEBUG
```

# Options à l'édition de liens et autres options

On peut également passer des options pour l'éditeur de lien à l'aide de `libraries`, `library_dirs` et `runtime_library_dirs` :

```
mod = Extension(...,
                library_dirs=['/usr/X11R6/lib'],
                libraries    = ['X11', 'Xt'])
```

D'autres options peuvent être passées :

- `extra_objects` : liste de fichiers objets à rajouer à l'édition de liens. Ne pas mettre les extensions (`.o` ou `.obj`) qui seront rajoutées selon la plateforme ;
- `extra_compile_args` et `extra_link_args` : pour rajouter d'autres options à la compilation ou l'édition de lien ;
- `export_symbols` : Uniquement pour windows. Spécifie les symboles à exporter pour être visibles à l'édition de lien.

# Plan

# Présentation de Scons

Scons en bref :

- Utilitaire par scripts utilisant python comme langage de base ;
- Permet de produire des programmes ou autres fichiers ;
- Assure une mise à jour correcte lors d'un changement de fichiers sources (analyse de dépendance) ;
- Très bonne rapidité d'analyse de dépendance et de compilation ;
- Essaye au maximum de proposer les bons outils par défaut en fonction de la plateforme afin de minimiser le travail du programmeur.
- Reconnaît un grand nombre de langages ;
- Existe sur un grand nombre de plateformes : Unix, Windows, Mac OS X...

# Principes de base

Quelques régles d'écriture doivent être appliquées :

- Le script à la racine du projet doit être nommé `SConstruct`. Il peut produire lui-même des fichiers ou appeler des scripts se trouvant dans des sous-répertoires ;
- Les scripts dans les sous–répertoires peuvent produire ou appeler eux-mêmes des scripts se trouvant dans des sous-sous-répertoires.
- Ne pas préjuger de l'ordre d'exécution : langage déclaratif. L'ordre d'exécution décidé par scons à son exécution.

# Invocation de scons

Scons doit toujours être invoqué à la racine du projet (là où se trouve le fichier `SConstruct`).
Quelques commandes de base :

- Compilation simple :

```
scons
```

- Compilation parallèle (option `-j #n`) :

```
scons -j 2
```

- Nettoyage des fichiers intermédiaires :

```
scons -c
```

- Installation des fichiers du projet :

```
scons install
```

# Exemples basiques de fichiers SConstruct

Quelques exemples simples :

- Compiler un programme C :

```
src_files = ['mainProg.c', 'fichier1.c',
             'fichier2.c']
Program('mainProg', src_files)
```

- Compiler une bibliothèque statique Fortran :

```
Library('mathutils', 'alg.f90 geom.f90 fem.f90')
```

- Construire une bibliothèque dynamique C++ :

```
SharedLibrary('visu3d', ['screenGL.cpp', 'gui.cpp'])
```

- Générer de la documention pdf avec LaTeX :

```
PDF('MaDocumentation.pdf', 'refman.tex')
```

# Plan

# Notion d'environnement

Un environnement est une collection de variables d'environnement affectant la manière de produire :

- Construire un nouvel environnement :

```
env  = Environment()
env2 = Environment(CCFLAGS=['-g','-O2'])
env3 = env2.clone()
```

- Interrogation d'un environnement :

```
print "Compilateur C utilise :", env['CC']
dicEnv = env.Dictionary()
for key in ['OBJSUFFIX', 'LIBSUFFIX', 'PROGSUFFIX']:
  print "key = %s, value = %s"%(key, dict[key])
print env.Dump() # dump toutes les var. d'environnement.
```

- Utiliser un environnement pour compiler

```
lib = env.Library('libFor', Split('alg.f90','fem.f90'))
env2.Program('forprog',LIBS=['libFor'],LIBPATH=['#lib'])
```

- Modification d'un environnement :

```
env.AppendUnique(CPPPATH=env['CPPPATH']+['#'])
env.Replace(LIBPATH=['/usr/local/lib'])
```

# Production hiérarchique

On peut écrire des fichiers `Sconscript` dans des sous–répertoires pour produire localement dans chaque répertoire.

Exemple :

- À la racine, dans fichier `Sconstruct` :

```
envGlob = Environment()
...
# La deuxieme liste permet de passer des variables au
# fichier Sconscript
# On pourrait aussi faire un Export('envGlob')
SConscript(['LinearAlgebra/linalg.scons', 'FEM/fem.scons'],
           ['envGlob'])
```

- Dans un sous–répertoire, dans fichiers `Sconscript`.

  Exemple `fem.scons` :

```
# On aurait pu ecrire Import('*')
# qui importe toutes les variables passees
# par le fichier SConstruct
Import('envGlob')
envGlob.SharedLibrary('fem', ['fem.cpp'])
```

Fichiers `Sconscript` peuvent appeler d'autres scripts scons.

# Récupérer les options de scons

On utilise pour cela la fonction `GetOption` :

Par exemple :

```
if GetOption('help'):
  # Rajout un message d'aide en plus
if not GetOption('clean'):
  # Dans le cas ou on ne nettoie pas, faire certaines actions.
```

# Variables en lignes

Scons permet de définir une variable de construction en ligne :

```
scons debug=1
```

Scons stocke ces variables dans un dictionnaire `ARGUMENTS` :

```
env = Environment()
debug = ARGUMENTS.get('debug',0)
if int(debug):
  env.Append(CCFLAGS='-g')
```

Si ordre des variables important ou si variable peut avoir multiples valeurs, on peut récupérer les variables dans une liste `ARGLIST` :

```
cppdefines = []
env = Environment()
for key, val in ARGLIST :
  if key == 'def' : cppdefines.append(val)
env = Environment(CPPDEFINES=cppdefines)
```

```
scons def=DEBUG def=TRACE
```

# Variables en ligne

Si beaucoup d'options possibles, `ARGLIST` et `ARGUMENTS` pas pratiques.
De plus, pas d'aides possibles pour l'utilisateur.
On utilise alors le constructeur `Variables` :

```
vars = Variables()
vars.Add('RELEASE', 'Mettre a 1 pour une release', 0)
env  = Environment(variables=vars,
                   CPPDEFINES={'RELEASE_BUILD' : '${RELEASE}'})
Help(vars.GenerateHelpText(env))
```

Pour obtenir l'aide en ligne :

```
scons -h
```

Mettre toutes les options en ligne peut être pénible. On peut demander à lire ces options dans un fichier :

```
vars = Variables('custom.py')
```

avec, par exemple, dans `custom.py` :

```
RELEASE=1
```

# Séparation Sources et compilation

Par défaut, Scons construit les binaires dans les mêmes répertoires que les sources !
En appelant un script `Sconscript` dans un sous–répertoire, on peut préciser où construire les binaires :

```
SConscript('src/Sconscript', variant_dir='build')
```

Par défaut, Scons duplique arbre des sources dans cible de compilation pour garantir une construction correcte.
On peut éviter cette duplication (mais c'est déconseillé !) en rajoutant l'option `duplicate=0`

```
SConscript('src/Sconscript', variant_dir='build',duplicate=0)
```

Le problème de cette approche est que divers scripts Sconscript peuvent dans ce cas construire des binaires dans le même répertoire !

# Séparation Sources et compilation...

On peut utiliser le constructeur `VariantDir` :

```
VariantDir('build', 'src')
env=Environment()
env.Program('build/hello.c')
```

Si on utilise des scripts `Sconscript` :

```
VariantDir('build', 'src')
SConscript('build/Sconscript')
```

# Construire des binaires en fonction des plateformes

Exemple :

```
platform = ARGUMENTS.get("OS", Platform())
include = "#export/$PLATFORM/include"
lib = "#export/$PLATFORM/lib"
bin = "#export/$PLATFORM/bin"
env = Environment(PLATFORM = platform,
                  BINDIR = bin,
                  INCDIR = include,
                  LIBDIR = lib,
                  CPPPATH = [include],
                  LIBPATH = [lib],
                  LIBS = "world")
Export("env")
env.SConscript("src/SConscript", variant_dir="build/$PLATFORM")
```

# Définitions de cibles

Exemple : Créer une cible pour installer notre production

```
env = Environment()
p = env.Program('foo.c')
l = env.Library('bar.c')
env.Install('/usr/local/bin', p)
env.Install('/usr/local/lib', l)
ib = env.Alias('install-bin', '/usr/local/bin')
il = env.Alias('install-lib', '/usr/local/lib')
env.Alias('install', [ib, il])
```

# Outils d'autoconfigurations

Le principe fondamental est d'attacher un contexte de configuration à un environnement :

```
env = Environment()
conf= Configure(env)
# Verification des librairies, headers, autres...
...
env = conf.Finish()
```

Vérification de l'existence de headers C :

```
if not conf.CheckCHeader('GL/gl.h'):
    print "OpenGL doit etre installe"
    Exit(1)
if conf.CheckCHeader('GL/glut.h'):
    conf.env.Append('-DHAS_GLUT_H')
```

Vérification de l'existence de headers C++ :

```
if not conf.CheckCXXHeader('freefemm.h'):
    print "Freefem doit etre installe"
    Exit(1)
```

# Outils d'autoconfigurations...

Vérification de l'existence de fonction dans la bibliothèque standard :

```
if not conf.CheckFunc('strcpy'):
   print 'Ne trouve pas la fonction strcpy !'
   print 'Utilisation fonction interne'
   conf.env.Append(CPPDEFINES='-Dstrcpy=local_strcpy')
```

Vérification de l'existence de librairies :

```
if not conf.CheckLib('GL'):
  print 'Ne trouve pas la librairie OpenGL !\n'
  Exit(1)
```

Vérification de l'existence des headers et librairies :

```
if not conf.CheckLibWithHeader('GL','GL/gl.h', 'c'):
  print 'Ne trouve pas OpenGL !\n'
  Exit(1)
```

Attention, vérification fait même durant un nettoyage ! Utiliser le constructeur `GetOption`.

# Outils de déboguages

Scons fournit quelques outils de déboguages :

- Savoir pourquoi un fichier se reconstruit (donne ses dépendances) :

```
scons --debug=explain
```

# Installation d'un module C Python avec Scons

Pour cela, on va utiliser le package distutils de Python !
Plusieurs étapes à respecter :

1. Importation du module `sysconfig` du package `distutils` :

```
import distutils.sysconfig, os
```

2. Récupération de l'environnement de compilation de python :

```
vars=distutils.sysconfig.get_config_vars('CC','CXX','OPT',
                                         'BASECFLAGS',
                                         'CCSHARED',
                                         'LDSHARED', 'SO')
```

3. Remplacer les `None` par des chaînes de caractère vides :

```
for i in range(len(vars)):
    if vars[i] is None:
        vars[i] = ""
(cc,cxx,opt,basecflags,ccshared,ldshared,so_ext) = vars
```

# Installation d'un module Python avec Scons...

1. Récupération de la version de python utilisée :

```
python_version = distutils.sysconfig.get_python_version()
```

2. Racine de l'endroit où Python est installé :

```
sysprefix = distutils.sysconfig.PREFIX
```

3. Définition de l'environnement adéquat :

```
incpath = sysprefix+"/include/python"+python_version
env = Environment(CC=cc, CXX=cxx, CCFLAGS=basecflags,
                  LIBPATH=[sysprefix+"/libs"],
                  CPPPATH=[incpath],
                  SHLIBSUFFIX=so_ext, SHLIBPREFIX='')
```

4. Compilation du module :

```
heatmod = env.SharedLibrary('HeatEqua', ['PyStaticHeat.c'])
```

# Installation d'un module Python avec Scons...

1. Description de l'installation du module :

```
ModDir='#lib/python'+python_version+'/site-packages'
dp1 = env.Install(ModDir, heatmod)
env.Alias(target="install", source=[dp1])
```

# Autres fonctionnalités de Scons

- Changer le type de détection pour la mise à jour. Par défaut, basé sur signature MD5. Mais on peut le baser sur la date de changement des fichiers, par exemple.
- Gestion explicite des dépendances ;
- Construire ses propres constructeurs (Exemple : pour compiler un programme Cuda, preprocessing).
- Auto–configuration similaire aux autotools.
- Construire une distribution binaire (rpm, msi, tgz, . . . )