INSTITUT
de MATHEMATIQUES
de TOULOUSE

Python intermédiaire pour les scientifiques

Session 3/5 : Python packaging

Laurent Risser

Ingénieur de Recherche à l'Institut de Mathématiques de Toulouse

lrisser@math.univ-toulouse.fr

Résumé

En bref :

- Création de modules et de librairies (packages).
- Packaging, cas tests et codes de bonne conduite.
- Checklist de choses à faire avant de terminer un projet/CDD.
- Distribution d'un code.
- Portabilité Windows/Mac/Linux (format des paths).
- Portabilité Python 2.7 et Python 3.x.
- Création de notebooks.

Rappels de la session 1

→ Un module contient plusieurs fonctions et commandes
→ Fonctions et commandes regroupées dans un fichier *.py*
→ Module appelé en utilisant `import`

toto.py :

```
def SayHi():
    print 'Hi'

def DivBy2(x):
    return x/2.
```

```
import toto

toto.SayHi()
→ Hi

print toto.DivBy2(10)
→ 5.0
```

```
import toto as tt

tt.SayHi()
→ Hi

print tt.DivBy2(10)
→ 5.0
```

```
from toto import SayHi

SayHi()
→ Hi

print DivBy2(10)
→ error
```

```
from toto import *

SayHi()
→ Hi

print DivBy2(10)
→ 5.0
```

Module considéré comme un script lorsqu'il contient des commandes :

- Lors de l'import d'un script, les commandes vont être exécutées
- Lors de l'import de fonctions, elles sont juste chargées en mémoire

Rappels de la session 1

toto.py :

```
def SayHi():
    print 'Hi'

def DivBy2(x):
    return x/2.
```

```
import toto

toto.SayHi()
→ Hi

print toto.DivBy2(10)
→ 5.0
```

```
import toto as tt

tt.SayHi()
→ Hi

print tt.DivBy2(10)
→ 5.0
```

```
from toto import SayHi

SayHi()
→ Hi

print DivBy2(10)
→ error
```

```
from toto import *

SayHi()
→ Hi

print DivBy2(10)
→ 5.0
```

Compilation du module :

- Lors de son 1er appel, un module python est compilé dans un fichier *.pyc*
- Le fichier compilé est utilisé lors des appels suivants
- Pour recharger (et recompiler) un module il faut taper la commande `reload(name)`

Répertoires des modules :

- Listés dans sys.path (après un import sys)
- Les répertoires de sys.path sont ceux contenus dans la variable d'environnement PYTHONPATH

toto.py :

```
def SayHi():
    print 'Hi'

def DivBy2(x):
    return x/2.
```

```
import toto
import time

toto.SayHi()
→ Hi

a=time.time()
print(time.time()-a)
→ 1.18850302696
```

→ sys.path contient le répertoire courant et celui dans lequel on trouve time.py

Répertoires des modules :

- Listés dans sys.path (après un import sys)
- Les répertoires de sys.path sont ceux contenus dans la variable d'environnement PYTHONPATH

toto.py :

```
def SayHi():
    print 'Hi'

def DivBy2(x):
    return x/2.
```

```
import toto
import time

toto.SayHi()
→ Hi

a=time.time()
print(time.time()-a)
→ 1.18850302696
```

→ sys.path contient le répertoire courant et celui dans lequel on trouve time.py

```
import sys


print sys.path
→ ['', '/Users/risser/anaconda/bin', '/Users/risser/
anaconda/lib/python27.zip', ... , '/Users/risser/anaconda/
lib/python2.7/site-packages/IPython/extensions', '/Users/
risser/.ipython']
```

1) Création de modules et de packages

Répertoires des modules :

- Listés dans sys.path (après un import sys)
- Les répertoires de sys.path sont ceux contenus dans la variable d'environnement PYTHONPATH

toto.py :

```
def SayHi():
    print 'Hi'

def DivBy2(x):
    return x/2.
```

```
import toto
import time

toto.SayHi()
→ Hi

a=time.time()
print(time.time()-a)
→ 1.18850302696
```

→ sys.path contient le répertoire courant et celui dans lequel on trouve time.py

```
import sys


print sys.path
→ ['', '/Users/risser/anaconda/bin', '/Users/risser/
anaconda/lib/python27.zip', ... , '/Users/risser/anaconda/
lib/python2.7/site-packages/IPython/extensions', '/Users/
risser/.ipython']
```

- Si toto.py est dans un réperoire [MyPythonModulePath] et pas dans le répertoire courant :
  → Ajouter [MyPythonModulePath] à sys.path : sys.path.append('MyPythonModulePath')
  → Importer ensuite toto

- Lister ce que contient un module : fonction dir

1) Création de modules et de packages

→ Les packages regroupent plusieurs modules dans un (des) répertoire(s) donné(s)

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            ...
      effects/                  Subpackage for sound effects
            __init__.py
            echo.py
            surround.py
            ...
      ...
```

http://docs.python.org/2/tutorial/modules.html

Pour charger un sous-module : `import sound.effects.echo` ou `from sound.effects import echo`

Fichiers *__init__.py* :

- Pour que Python traite le répertoire comme un package
- Peuvent être vide, contenir des commandes à exécuter, ou définir des variables

Nous n'avons pas la prétention lors de cette formation d'expliquer toutes les subtilités pour coder un package comme networkx …

... cependant il est intéressant de s'en inspirer pour créer des packages propres et réutilisables

Packaging et codes de bonne conduite :
Créer un répertoire [MonProjet]
→ Contient tout mon travail sur ce projet

Packaging et codes de bonne conduite :

Créer un répertoire [MonProjet]

→ Contient tout mon travail sur ce projet

Créer un sous répertoire [MonProjet/Code]

→ Contient l'ensemble de mon code

→ Si le code fait appel a des sous-fonctions de type différent (ex : lecture/écriture des données, visualisation de données, traitements, …), mettre les données dans des fichiers différents.

→ Si le code est gros, utiliser des sous répertoires contenant chacun des fichiers de fonctions spécialisés.

→ Dans tous les cas, rédiger un fichier __init__.py dans chaque répertoire. Y indiquer les fichiers que l'on veut ouvrir lors de l'appel du répertoire (comme un module).

Packaging et codes de bonne conduite :

Créer un répertoire [MonProjet]

→ Contient tout mon travail sur ce projet

Créer un sous répertoire [MonProjet/Code]

→ Contient l'ensemble de mon code

→ Si le code fait appel a des sous-fonctions de type différent (ex : lecture/écriture des données, visualisation de données, traitements, …), mettre les données dans des fichiers différents.

→ Si le code est gros, utiliser des sous répertoires contenant chacun des fichiers de fonctions spécialisés.

→ Dans tous les cas, rédiger un fichier __init__.py dans chaque répertoire. Y indiquer les fichiers que l'on veut ouvrir lors de l'appel du répertoire (comme un module).

Rédiger un README dans [MonProjet]

→ Décrit ce que fait le package, les auteurs, la date des dernières modifications

→ Donne un petit exemple d'utilisation à copier/coller.

Packaging et codes de bonne conduite :

Créer un répertoire [MonProjet]

→ Contient tout mon travail sur ce projet

Créer un sous répertoire [MonProjet/Code]

→ Contient l'ensemble de mon code

→ Si le code fait appel a des sous-fonctions de type différent (ex : lecture/écriture des données, visualisation de données, traitements, …), mettre les données dans des fichiers différents.

→ Si le code est gros, utiliser des sous répertoires contenant chacun des fichiers de fonctions spécialisés.

→ Dans tous les cas, rédiger un fichier __init__.py dans chaque répertoire. Y indiquer les fichiers que l'on veut ouvrir lors de l'appel du répertoire (comme un module).

Rédiger un README dans [MonProjet]

→ Décrit ce que fait le package, les auteurs, la date des dernières modifications

→ Donne un petit exemple d'utilisation à copier/coller.

Créer un sous répertoire [MonProjet/data]

→ Contient des donnée de petite taille sur lesquelles on pourra tester le package

→ Les données doivent être a un format typiquement utilisé par la communauté ciblée

Packaging et codes de bonne conduite :

Créer un répertoire [MonProjet]

→ Contient tout mon travail sur ce projet

Créer un sous répertoire [MonProjet/Code]

→ Contient l'ensemble de mon code

→ Si le code fait appel a des sous-fonctions de type différent (ex : lecture/écriture des données, visualisation de données, traitements, …), mettre les données dans des fichiers différents.

→ Si le code est gros, utiliser des sous répertoires contenant chacun des fichiers de fonctions spécialisés.

→ Dans tous les cas, rédiger un fichier __init__.py dans chaque répertoire. Y indiquer les fichiers que l'on veut ouvrir lors de l'appel du répertoire (comme un module).

Rédiger un README dans [MonProjet]

→ Décrit ce que fait le package, les auteurs, la date des dernières modifications

→ Donne un petit exemple d'utilisation à copier/coller.

Créer un sous répertoire [MonProjet/data]

→ Contient des donnée de petite taille sur lesquelles on pourra tester le package

→ Les données doivent être a un format typiquement utilisé par la communauté ciblée

Créer un sous répertoire [MonProjet/examples]

→ Contient une série de petits fichiers Python, chacun permettant de tester facilement les fonctions principales du package à l'aide de copier/coller.

→ En lien avec [MonProjet/data]

Cas tests :

On a évoqué les tests de [MonProjet/examples] avec les données de [MonProjet/data] qui sont à direction des utilisateurs. Les cas tests sont plus pour le développeur :

→ Permet de vérifier que l'ensemble des fonctionnalités du code marchent correctement lorsque l'on étend ou optimise les fonctions existantes.

Cas tests :

On a évoqué les tests de [MonProjet/examples] avec les données de [MonProjet/data] qui sont à direction des utilisateurs. Les cas tests sont plus pour le développeur :

→ Permet de vérifier que l'ensemble des fonctionnalités du code marchent correctement lorsque l'on étend ou optimise les fonctions existantes.

- Définition d'un script python qui va appeler toutes les fonctions.
- Pour chaque fonction, les entrées sont prédéfinies et les sorties connues.
- Si une fonction retourne une valeur incorrecte, le script s'arrête ou (au moins) notifie qu'un test n'a pas marché.

→ Lourd à maintenir mais extrêmement nécessaire quand un code devient gros et/ou que l'on travaille à plusieurs sur le même code.

3) Checklist avant de terminer un projet

A minima :

- Est-ce que mon code est bien commenté ?
- Ai-je fourni des données types avec mon code ?
- Ai-je donné des exemples types qui tournent avec les données type ?
- Y a t-il un fichier README qui explique ce que fait le code et comment l'installer ?

Si une réponse est non, votre code ne sera vraisemblablement jamais réutilisé !

Mieux :

- Suivre tant que possible les règles énoncées dans les transparents précédents.
- Si vous avez publié des résultats à partir de ce code, n'oubliez pas de mentionner la référence de l'article dans le code et -- idéalement -- là où trouver le code dans l'article (parfait pour les citations).

4) Distribution d'un code

Techniques classiques :

- Un bon vieux .tar.gz ou .zip envoyé par email aux collègues.
- Ce même .tar.gz ou .zip téléchargeable sur sa page perso.

→ Simple et efficace mais risque fort de perdre le package au bout de quelques années, et aucun contrôle sur l'utilisation du code.

Ebergement sur un site web gratuit :

- https://sourceforge.net/ ou bien https://github.com/

→ Plus lourd que les techniques classiques mais pas tant que ça
→ Bonne visibilité (largement accrue si l'on fait un site web sur le package en plus)
→ Possibilité de choisir une licence pour protéger ou non l'utilisation/modification du code (BSD licences, … - voir : https://en.wikipedia.org/wiki/Comparison_of_free_and_open-source_software_licenses)
→ … mais tout le monde peut lire le code et il est ébergé on ne sait où ...

Protégez vous !!!

Il est possible que (1) quelqu'un récupère votre code, (2) ça cause des dégâts d'une manière ou d'une autre et (3) la personne se retourne contre vous…

Après tout c'est vous qui lui avez mis entre les mains un code qui a causé des dégats ☹

... peu de chance d'aboutir mais on ne sait jamais. Dans le code distribué, ne pas lésiner sur les messages du type :

Disclaimer: This software has been developed for research purposes only, and hence should not be used as a diagnostic tool. In no event shall the authors or distributors be liable to any direct, indirect, special, incidental, or consequential damages arising of the use of this software, its documentation, or any derivatives thereof, even if the authors have been advised of the possibility of such damage.

5) Portabilité Windows/Mac/Linux

A de rares exceptions près tout code Python est portable Windows/MAC/Linux.

Toutefois, pour tester le système, utiliser sys.platform Pour les systèmes les plus courants, les valeurs sont :

Linux	→	'linux2' (ou 'linux' après Python 3.3)
Windows	→	'win32'
Windows/Cygwin	→	'cygwin'
Mac OS X	→	'darwin'

Pour quasi tous les codes *scientifiques*, les noms de répertoires sont le seul problème.
→ Utiliser : os.path.join()

Sous Linux ou MacOS :

```
import os
SvgFile=os.path.join(".","DirOutputs","resultats.csv")
print SvgFile
→ ./DirOutputs/resultats.csv
```

Sous Windows :

```
import os
SvgFile=os.path.join(".","DirOutputs","resultats.csv")
print SvgFile
→ .\DirOutputs\resultats.csv
```

6) Portabilité Python 2.7 et Python 3.x

Python 2.7 reste le plus utilisé dans un cadre scientifique mais devrait ne plus être supporté un jour ou l'autre.

→ la date d'arrêt du support est prévu en 2020 mais la date *pourrait* être repoussée

→ Python 3 assure la rétrocompatibilité de la grande majorité des fonctions Python 2.7

Principales différences :

PYTHON 2.7	→	PYTHON 3	
`print "Bonjour"`	→	`print("Bonjour")`	
`print "Val=", 3`	→	`print("Val", 3)`	
`raise IOError, "file error"`	→	`raise IOError("file error")`	-- (appel des exceptions)
`Tkinter`	→	`tkinter`	-- (nom de quelques modules)
`xrange()`	→	`range()`	-- (nom de quelques objets)

Remarque :

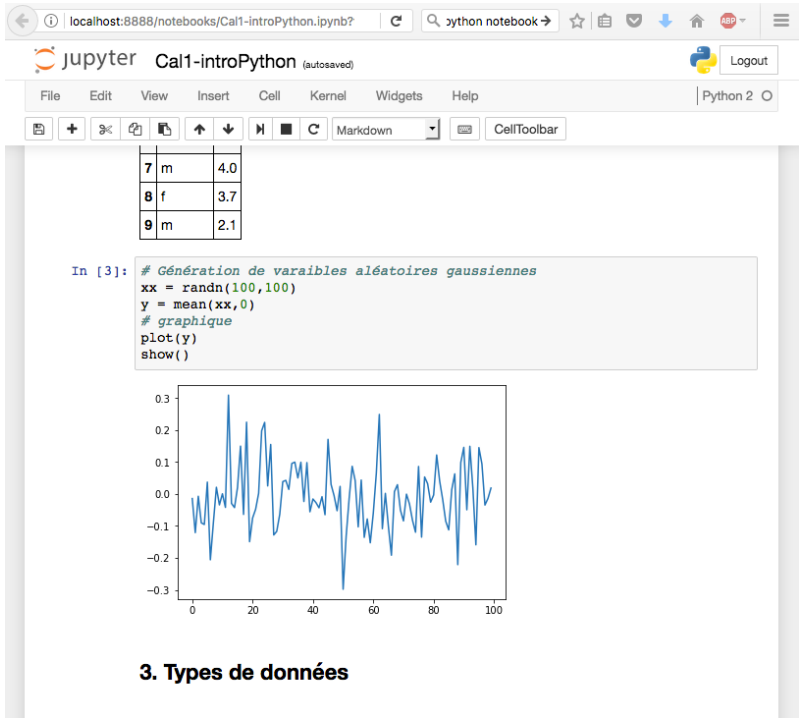
`print("Bonjour")`, `raise IOError("file error")`, `range()` , ... fonctionnent sous les versions récentes de 2.7.

→ Autant les utiliser dès maintenant.

7) Création de notebooks

Jupyter Notebooks (anciennement IPython Notebook) :

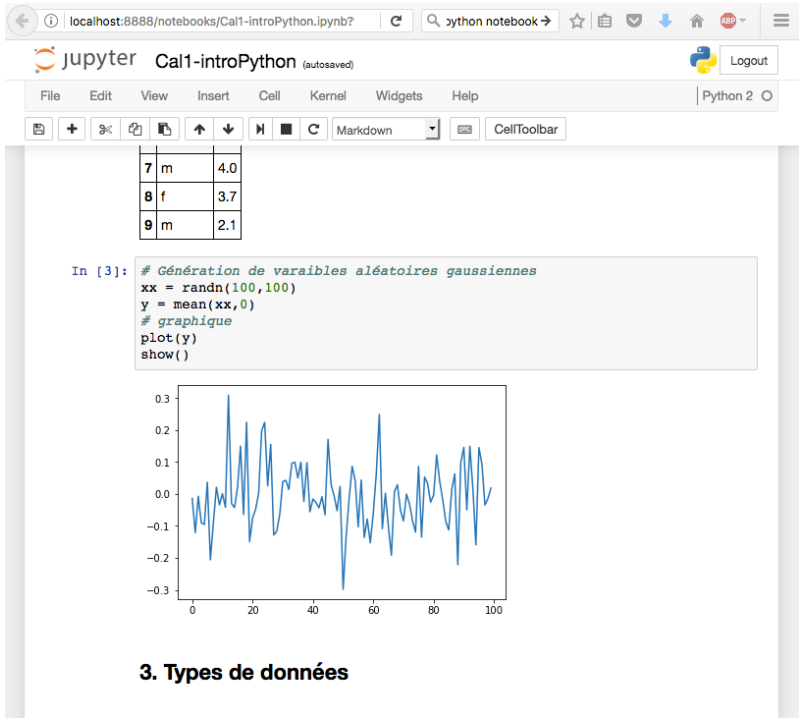
- Environnement *interactif* qui peut combiner du texte enrichi, l'exécution de code python, et des graphiques.
- Idéal pour la communication de résultats.
- Idéal pour l'enseignement des sciences numérique par la pratique.
- Intégré dans Anaconda et dans iPython



7) Création de notebooks

Jupyter Notebooks (anciennement IPython Notebook) :

- Environnement *interactif* qui peut combiner du texte enrichi, l'exécution de code python, et des graphiques.
- Idéal pour la communication de résultats.
- Idéal pour l'enseignement des sciences numérique par la pratique.
- Intégré dans Anaconda et dans iPython



Lecture d'un notebook à partir d'un fichier toto.ipynb :

→ Dans un terminal taper la commande

```
jupyter notebook toto.ipynb
```

ou bien :

```
ipython notebook toto.ipynb
```

→ Le fichier s'ouvre dans le navigateur internet

Anaconda navigator (si installé) peut aussi ouvrir le fichier toto.ipynb via l'explorateur Jupyter

7) Création de notebooks

Jupyter Notebooks (anciennement IPython Notebook) :

- Environnement *interactif* qui peut combiner du texte enrichi, l'exécution de code python, et des graphiques.
- Idéal pour la communication de résultats.
- Idéal pour l'enseignement des sciences numérique par la pratique.
- Intégré dans Anaconda et dans iPython

Créer un nouveau notebook :

→ Dans un terminal taper la commande

```
jupyter notebook
```

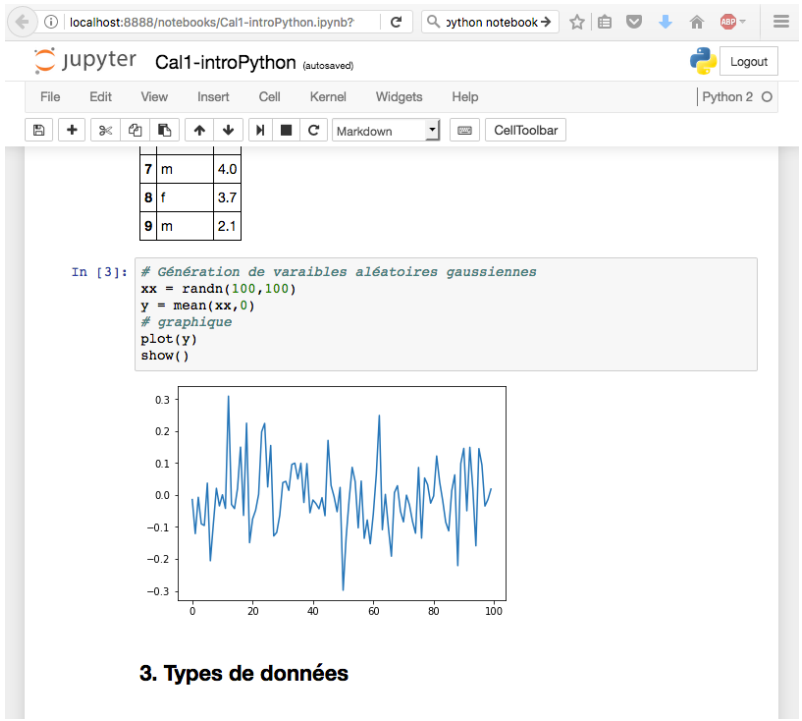
ou bien :

```
ipython notebook
```

→ Aller dans *New* et sélectionner *Python 2*

→ Le code s'écrit en mode *Code* et le texte en mode *Markdown*

→ On créé de nouvelles zones avec les *Cells*



Merci

Références :

- https://docs.python.org/2/tutorial/modules.html
- http://apprendre-python.com/page-syntaxe-differente-python2-python3-python-differences
- https://ipython.org/ipython-doc/3/notebook/notebook.html#introduction
- https://github.com/wikistat

Remerciements :

- Etienne Gondet
- Philippe Besse
- Vous

MERCI !!!