L1 SPI – Algorithmique

S03 – La notion de script

Cyril Desjouy

15 juillet 2024

Préambule

Les parties 1 à 3 de ce TP présentent le concept de script et de module. Ces parties seront faîtes avec votre encadrant et constituerons la base du cours à laquelle vous référer pour comprendre et approfondir ces notions. La partie 4 constitue une application pratique des notions vues dans les parties 1 à 3 précédentes.

1 Introduction

Maintenant que vous avez abordé la plupart des notions fondamentales de Python, il convient de passer à la vitesse supérieure et de commencer à coder de réels programmes. Lorsqu'il s'agit d'écrire quelques lignes de code, Jupyter est parfaitement adapté. Par contre, lorsqu'il s'agit de coder une application complète, s'étendant sur plusieurs dizaines (voir centaines ou milliers) de lignes souvent réparties dans plusieurs fichiers, les développeurs utilisent généralement un IDE (Integrated Development Environment). Spyder en est un parmi d'autres, et c'est celui que nous utiliserons pour nos développements ces prochaines années. Il a l'avantage d'être simple, efficace, adapté au calcul scientifique et il est de plus inclus dans la distribution Python *Anaconda*.

- Commencez par lancer le logiciel Spyder

L'interface de Spyder est subdivisée en plusieurs sous fenêtres principales dont :

- une sous fenêtre contenant un éditeur de texte,
- une sous fenêtre regroupant des consoles Python et IPython,
- une sous fenêtre contenant (entre autre) un inspecteur d'objets et un explorateur de variables.

- Commencez par tapez dans l'éditeur de texte une instruction (par exemple `a=1`)
- Sauvegardez ce fichier [a] et exécutez le (touche raccourci F5, et choisissez "exécuter dans la console Python ou IPython active")
- Observez les informations s'affichant dans la console et les modifications dans l'explorateur d'objets

a. Les scripts Python ont l'extension `.py`

Bravo ! Même si ce n'est pas le classique "Hello World", vous venez tout de même de créer votre premier **script** !

2 Interprétation des fichiers sources

2.1 Les scripts

Lors de l'exécution d'un programme, l'interpréteur Python lit un fichier source contenant du code (un script) et interprète de manière linéaire chaque instruction qu'il contient. La première ligne du fichier est lue et interprétée en premier, la seconde est lue et interprétée en deuxième, et ainsi de suite. Il convient donc de bien vérifier qu'à chaque ligne, l'interpréteur Python possède toutes les informations nécessaires pour procéder à son interprétation !

1. Créez un nouveau script que vous appellerez `FirstScript.py` dans lequel vous :
   — définirez une fonction `myfct()` comme suit :

```
def myfct(x):
    print("Le carré de {} est {}".format(x, x**2))
```

   — utiliserez cette fonction pour afficher le carré de la variable `a = 5` que vous aurez préalablement définie
2. Exécutez le script et observez le résultat dans la console Python (ou IPython)

Un script permet finalement de regrouper un ensemble d'instructions dans un même fichier. Comme cela a été précisé avant, l'exécution d'un script consiste simplement à interpréter successivement chaque ligne du fichier.

2.2 Modules et notion de *namespace*

1. Créez un nouveau script que vous appellerez `SecondScript.py` dans lequel vous :
   — importerez `FirstScript` en tant que module à l'aide de la fonction *built in* `import`,
   — définirez une fonction `myfct()` comme suit :

```
def myfct(x):
    print("Le cube de {} est {}".format(x, x**3))
```

   — utiliserez cette fonction pour afficher le cube de la variable `a = 3` que vous aurez préalablement définie.
2. Exécutez le script et observez le résultat dans la console Python (ou IPython)

N'y a-t-il pas quelques lignes inattendues ? En effet, lorsque l'instruction `import` est utilisée [1], l'interpréteur Python lit et exécute les instructions contenues dans le fichier source importé. Toutes les lignes de code interagissant avec la console sont donc interprétées classiquement et leurs résultats s'affichent sur la sortie standard. La commande `import` crée un nouvel objet dont le nom est ici `FirstScript` disponible dans le *namespace* de `SecondScript` et qui contient ses propres objets. Cet objet est un objet de type `module`.

La notion de *namespace* (parfois traduit en français par *espace de nommage*) est une notion fondamentale sous Python. Un *namespace* est simplement un conteneur de noms. Il est utilisé pour permettre la distinction entre deux éléments ayant le même nom. Considérons l'exemple suivant :

```
In  : import numpy          # On importe numpy dans le namespace courant
In  : numpy.abs             # le module numpy propose la fonction abs
Out : <ufunc 'absolute'>
In  : abs                   # abs est également définie dans le namespace courant...
Out : <function abs>        # ... et est différent de numpy.abs
```

Dans cet exemple, nous avons accès à deux objets différents portant le même nom :

— la fonction `abs()` du module `numpy` dont le nom est préfixé par le *namespace* appelé *numpy*,

— la fonction `abs()` dont le nom n'est pas préfixé et qui est donc dans le *namespace* courant.

Le nom du module est ainsi utilisé comme *namespace* ce qui permet de ne pas écraser des objets déjà définis dans le *namespace* courant. C'est pour cette raison qu'il est fortement conseillé d'éviter de copier directement une référence d'un namespace à l'autre à moins de savoir exactement ce que vous faîtes :

```
In  : from numpy import abs  # On copie abs dans le namespace courant
In  : abs
Out : <ufunc 'absolute'>     # On a donc écrasé la fonction built in abs
```

Chaque *namespace* est complètement isolé des autres et vous pouvez vérifier cela en reprenant vos expérimentations sur vos fichiers `FirstScript.py` et `SecondScript.py` :

1. Pour que l'importation se fasse correctement, il faut que l'interpréteur Python connaisse la localisation du fichier à importer. Si le fichier depuis lequel l'importation est réalisée est dans le même répertoire que le fichier à importer, l'importation se passera sans difficulté. Si ce n'est pas le cas et que Python ne trouve pas le fichier dans `PYTHON_PATH` (c'est la liste des dossiers dans laquelle Python effectue sa recherche de modules, nous y reviendrons plus tard...), l'interpréteur lèvera l'exception `ImportError`.

3. Modifiez le fichier `SecondScript.py` comme suit :
   - utilisez la fonction `print` pour afficher la valeur de l'attribut `a` du module `FirstScript`,
   - utilisez la fonction `myfct` du module `FirstScript` avec comme argument d'entrée la variable `a` dans le *namespace* courant puis la variable `a` dans le *namespace* de `FirstScript`,
   - faîtes la même chose avec la fonction `myfct` du *namespace* courant.
4. Exécutez `SecondScript.py`, observez le résultat dans la console et concluez.

2.3 La variable spéciale __name__

Lorsque l'interpréteur Python lit un fichier source (que ce soit directement dans le cadre de son exécution ou indirectement dans le cadre de son importation), il crée plusieurs variables spéciales. L'une d'entre elles nous intéressera particulièrement ici : il s'agit de la variable `__name__`.

1. Ajoutez les instructions :
   - `print("Lecture FirstScript.py :", __name__)` à la fin du fichier `FirstScript.py`
   - `print("Lecture SecondScript.py :", __name__)` à la fin du fichier `SecondScript.py`
2. Exécutez `FirstScript.py` et observez la valeur de l'objet référencé par la variable `__name__`.
3. Exécutez `SecondScript.py` et observez la valeur des objets référencés par les variables `__name__` dans les différents *namespaces*.

Dans le cas de l'exécution de `FirstScript.py`, l'interpréteur Python exécute votre script en tant que programme principal. Vous devriez voir que l'objet `__name__` est une chaîne de caractères contenant la valeur "___main___" (signifiant *principal* en anglais).

Dans le cas de l'exécution de `SecondScript.py`, l'interpréteur Python exécute également votre script en tant que programme principal. Vous devriez voir que l'objet `__name__` dans le *namespace* de `SecondScript` est une chaîne de caractères contenant également la valeur "___main___". Vous devriez par ailleurs voir s'afficher la valeur de l'objet `__name__` dans le *namespace* de `FirstScript` qui est une chaîne de caractères contenant la valeur "FirstScript". Vous aurez constaté que dans le cas présent, la variable spéciale `__name__` dans le *namespace* de `FirstScript.py` ne référence plus le même objet que lorsque vous avez précédemment exécuté directement le programme `FirstScript.py`.

Conclusions :

- Lorsqu'un fichier .py est exécuté, la variable spéciale `__name__` contient "___main___"
- Lorsqu'un fichier `MyModule.py` est importé depuis un fichier `MyScript.py` et que ce dernier est exécuté, la variable spéciale `__name__` contient
  - "___main___" dans le *namespace* de `MyScript.py`
  - "MyModule" dans le *namespace* de `MyModule.py`

3 Les scripts Python : Les bonnes pratiques

La variable `__name__` est d'une importance capitale sous Python puisqu'elle facilite et favorise la réutilisation de code. Comme vous l'avez constaté précédemment, l'importation d'un fichier source entraîne l'interprétation de toutes les lignes de ce fichier, ce qui peut poser problème lorsqu'on souhaite uniquement accéder aux fonctions qui y sont définies et non au corps du programme. C'est là que la variable `__name__` joue un rôle essentiel puisqu'elle permet la définition d'un bloc de code qui sera interprété **uniquement lors de l'exécution** grâce à l'instruction :

```
if __name__ == "__main__":
    instructions
```

Cette instruction permet d'exécuter le script intégralement s'il est lu en tant que programme principale et de n'exécuter que la partie avant l'instruction `if __name__ == "__main__"` s'il fait l'objet d'une importation.

La Fig. 1 suivante présente un *template* classique de script que vous pourrez réutiliser à volonté pour vos futurs développements.

script.py

```
#!/usr/bin/env python                -> Nécessaire pour l'exécution "directe"
# -*- coding: utf-8 -*-              -> Si absent, ASCII par défaut

'''
  La doc. de mon programme...        -> Documentation
  ... sur plusieurs lignes
'''

import mymodule1 as myshortcut1      -> Modules utilisés dans le programme
...
import mymoduleN as myshortcutN

def mygreatfct(a):                   -> Les fonctions définies par l'utilisateur
    '''Docstring de la fonction'''
    return a

...

if __name__ == "__main__":           -> Lit la suite seulement si fichier exécuté !
    first_line = "Yeah! It begins"   -> Début du programme principal
    ...
    ...
    last_line = "Yeah! It ends"      -> Fin du programme principal
```

FIGURE 1 – Trame conseillée pour l'écriture de scripts Python

En utilisant ce type de structure, vous vous assurez que chaque code que vous produirez pourra être réutilisé simplement dans le cadre d'autres applications en important vos scripts sous forme de module afin d'accéder aux fonctions que vous aurez précédemment développées. Il s'agit maintenant de tester ce *template* dans le cas des fichiers que vous avez crées précédemment :

1. Ajoutez la condition sur la variable `__name__` à un endroit judicieux dans votre fichier `FirstScript`
2. Exécutez `FirstScript.py` et observez le résultat dans la console
3. Exécutez `SecondScript.py` et observez le résultat dans la console

4 Application : Création d'un module pour l'acoustique

L'objectif de ce TP est de développer sous Python des outils facilitant la manipulation des niveaux acoustiques. Ces derniers sont généralement exprimés :

— en échelle linéaire, avec pour unité le Pascal (Pa),

— ou en échelle logarithmique, avec pour unité le décibel (dB SPL pour Sound Pressure Level).

4.1 Rappel : conversions dB SPL $\Longleftrightarrow$ Pa

La conversion en décibels d'un niveau acoustique en pascals se fait à l'aide de la formule :

$$L_{dB} = 20\log_{10}(\frac{p}{p_0}) \tag{1}$$

où L_{dB} est le niveau acoustique en décibels, p est le niveau acoustique en pascals et p_0 est la pression de référence, égale à 2.10^{-5} Pa. L'opérateur $\log_{10}$ utilisé ici est le logarithme en base 10 qui peut être calculé à l'aide de la fonction `log10()` fournie par le module `numpy`. La conversion en pascals d'un niveau en décibels se fait quant à elle à l'aide de la formule inverse, à savoir :

$$p = p_0 \cdot 10^{L_{dB}/20}. \tag{2}$$

4.2 Rappel : addition de niveaux en décibels

On ne peut pas directement additionner des niveaux en décibels. Pour calculer le niveau équivalent de N sources incohérentes, il convient de faire la somme logarithme de leurs intensités comme suit :

$$L_T = 10\log_{10}\left(\sum_{i=1}^{N} 10^{L_i/10}\right) \tag{3}$$

où L_i est le niveau de pression de la source i en décibel.

4.3 Le module `dButils`

Le but de cette première partie est de créer un module appelé `dButils` fournissant des fonctions de base pour la manipulation des niveaux acoustiques. Ce module fournira 3 fonctions, à savoir :

— la fonction `dB2Pa` prenant en argument d'entrée un niveau acoustique en dB SPL et retournant le niveau équivalent en Pa,

— la fonction `Pa2dB` prenant en argument d'entrée un niveau acoustique en Pa et retournant le niveau équivalent en dB SPL,

— la fonction `dBsum` prenant en argument d'entrée une **liste** de niveaux acoustiques (en dB SPL) et retournant le résultat de la somme des niveaux (en dB SPL) regroupés dans cette liste.

1. Créer un module `dButils` implémentant les 3 fonctions introduites ci-dessus, à savoir `dB2Pa`, `Pa2dB`, et `dBsum`.
2. Le module `dButils` contiendra des tests [a] permettant de valider les fonctions développées précédemment. Ces tests prendront la forme de 4 petites applications utilisant ces fonctions et permettant d'afficher les 4 lignes présentées dans l'encadré de la figure 2 :
   — convertir en Pa le niveau 94 dB SPL,
   — convertir en dB SPL le niveau 1 Pa,
   — sommer deux sources de niveau 94 dB SPL,
   — sommer trois sources de niveau 94, 94 et 97 dB SPL.

a. Ces tests **ne seront pas exécutés** lorsque `dButils` fait l'objet d'une procédure d'importation !

```
94.0 dB SPL = 1.0 pa
1.0 Pa = 94.0 dB SPL
[94, 94] dB SPL => 97.0 dB SPL
[94, 94, 97] dB SPL => 100.0 dB SPL
```

FIGURE 2 – Exemple d'affichage lors de l'exécution du module `dButils`

4.4 La classe dB

L'objectif de cette seconde partie est de développer un nouveau type d'objet sous Python. Il s'agit du type `dB`. Ce type implémentera l'affichage, l'addition et la conversion. Vous pourrez utilisez le module `dButils` écrit dans la partie 1 précédente pour mener à bien ce développement.

1. Créer un module `dB` implémentant une classe `dB` dont chaque instance sera initialisée avec l'attribut d'instance `value` de type `float` qui aura pour valeur par défaut 94.
2. Ajouter à ce nouveau type la méthode spéciale `__str__()` permettant l'affichage de chaque instance de la classe `dB` comme suit :

```
» L1 = dB()
» print(L1)
94.0 dB SPL
```

Note : *affichage à 1 chiffre après la virgule !*

3. Ajouter à ce nouveau type la méthode spéciale `__add__()` permettant l'addition de deux objets de type `dB`. Vous pouvez pour ce faire utiliser la fonction `dBsum` du module `dButils` précédemment développé.

```
» L1 = dB(94)
» L2 = dB(100)
» L3 = L1 + L2
» print("{} + {} = {}".format(L1, L2, L3))
94.0 dB SPL + 100.0 dB SPL = 101.0 dB SPL
```

4. Ajouter à cette classe une méthode appelée `to_pascals()` fonctionnant comme suit :

```
» L1 = dB(94)
» L1.to_pascals()
94.0 dB SPL correponds to 1.0 Pa
```

Note : *Vous pouvez pour ce faire utiliser la fonction `dB2Pa` du module `dButils` développé précédemment.*