14 Création de modules

14.1 Pourquoi créer ses propres modules ?

Dans le chapitre 8 *Modules*, nous avons découvert quelques modules existants dans Python comme *random*, *math*, etc. Nous avons vu par ailleurs dans les chapitres 9 *Fonctions* et 12 *Plus sur les fonctions* que les fonctions sont utiles pour réutiliser une fraction de code plusieurs fois au sein d'un même programme sans avoir à dupliquer ce code. On peut imaginer qu'une fonction bien écrite pourrait être judicieusement réutilisée dans un autre programme Python. C'est justement l'intérêt de créer un module. On y met un ensemble de fonctions que l'on peut être amené à utiliser souvent. En général, les modules sont regroupés autour d'un thème précis. Par exemple, on pourrait concevoir un module d'analyse de séquences biologiques ou encore de gestion de fichiers PDB.

14.2 Création d'un module

En Python, la création d'un module est très simple. Il suffit d'écrire un ensemble de fonctions (et/ou de constantes) dans un fichier, puis d'enregistrer ce dernier avec une extension `.py` (comme n'importe quel script Python). À titre d'exemple, nous allons créer un module simple que nous enregistrerons sous le nom `message.py` :

```
"""Module inutile qui affiche des messages :-)."""

DATE = 16092008


def bonjour(nom):
    """Dit Bonjour."""
    return "Bonjour " + nom


def ciao(nom):
    """Dit Ciao."""
    return "Ciao " + nom


def hello(nom):
    """Dit Hello."""
    return "Hello " + nom
```

Les chaînes de caractères entre triple guillemets en tête du module et en tête de chaque fonction sont facultatives mais elles jouent néanmoins un rôle essentiel dans la documentation du code.

Remarque

Une constante est, par définition, une variable dont la valeur n'est pas modifiée. Par convention en Python, le nom des constantes est écrit en majuscules (comme `DATE` dans notre exemple).

14.3 Utilisation de son propre module

Pour appeler une fonction ou une variable de ce module, il faut que le fichier `message.py` soit dans le répertoire courant (dans lequel on travaille) ou bien dans un répertoire listé par la variable d'environnement `PYTHONPATH` de votre système d'exploitation. Ensuite, il suffit d'importer le module et toutes ses fonctions (et constantes) vous sont alors accessibles.

Remarque

Avec Mac OS X et Linux, il faut taper la commande suivante depuis un *shell* Bash pour modifier la variable d'environnement `PYTHONPATH` :
`export PYTHONPATH=$PYTHONPATH:/chemin/vers/mon/super/module`

Avec Windows, mais depuis un *shell* PowerShell, il faut taper la commande suivante :
`$env:PYTHONPATH += ";C:\chemin\vers\mon\super\module"`

Une fois cette manipulation effectuée, vous pouvez contrôler que le chemin vers le répertoire contenant vos modules a bien été ajouté à la variable d'environnement `PYTHONPATH` :

- sous Mac OS X et Linux : `echo $PYTHONPATH`
- sous Windows : `echo $env:PYTHONPATH`

Le chargement du module se fait avec la commande `import message`. Notez que le fichier est bien enregistré avec une extension `.py` et pourtant on ne la précise pas lorsqu'on importe le module. Ensuite, on peut utiliser les fonctions comme avec un module classique.

```
>>> import message
>>> message.hello("Joe")
'Hello Joe'
>>> message.ciao("Bill")
'Ciao Bill'
>>> message.bonjour("Monsieur")
'Bonjour Monsieur'
>>> message.DATE
16092008
```

Remarque

La première fois qu'un module est importé, Python crée un répertoire nommé `__pycache__` contenant un fichier avec une extension `.pyc` qui contient le bytecode, c'est-à-dire le code précompilé du module.

14.4 Les *docstrings*

Lorsqu'on écrit un module, il est important de créer de la documentation pour expliquer ce que fait le module et comment utiliser chaque fonction. Les chaînes de caractères entre triple guillemets situées en début du module et de chaque fonction sont là pour cela, on les appelle *docstrings* (« chaînes de documentation » en français). Ces *docstrings* permettent notamment de fournir de l'aide lorsqu'on invoque la commande `help()` :

```
>>> help(message)

Help on module message:

NAME
    message - Module inutile qui affiche des messages :-).

FUNCTIONS
    bonjour(nom)
        Dit Bonjour.

    ciao(nom)
        Dit Ciao.

    hello(nom)
        Dit Hello.

DATA
    DATE = 16092008

FILE
    /home/pierre/message.py
```

Remarque

Pour quitter l'aide, pressez la touche `Q`.

Vous remarquez que Python a généré automatiquement cette page d'aide, tout comme il est capable de le faire pour les modules internes à Python (*random*, *math*, etc.) et ce grâce aux *docstrings*. Notez que l'on peut aussi appeler l'aide pour une seule fonction :

```
>>> help(message.ciao)

Help on function ciao in module message:

```

```
ciao(nom)
    Dit Ciao.
```

En résumé, les *docstrings* sont destinés aux utilisateurs du module. Leur but est différent des commentaires qui, eux, sont destinés à celui qui lit le code (pour en comprendre les subtilités). Une bonne *docstring* de fonction doit contenir tout ce dont un utilisateur a besoin pour utiliser cette fonction. Une liste minimale et non exhaustive serait :

- ce que fait la fonction,
- ce qu'elle prend en argument,
- ce qu'elle renvoie.

Pour en savoir plus sur les *docstrings* et comment les écrire, nous vous recommandons de lire le chapitre 15 *Bonnes pratiques en programmation Python*.

14.5 Visibilité des fonctions dans un module

La visibilité des fonctions au sein des modules suit des règles simples :

- Les fonctions dans un même module peuvent s'appeler les unes les autres.
- Les fonctions dans un module peuvent appeler des fonctions situées dans un autre module s'il a été préalablement importé. Par exemple, si la commande `import autremodule` est utilisée dans un module, il est possible d'appeler une fonction avec `autremodule.fonction()`.

Toutes ces règles viennent de la manière dont Python gère les **espaces de noms**. De plus amples explications sont données sur ce concept dans le chapitre 19 *Avoir la classe avec les objets*.

14.6 Module ou script ?

Vous avez remarqué que notre module `message` ne contient que des fonctions et une constante. Si on l'exécutait comme un script classique, cela n'afficherait rien :

```
$ python message.py
$
```

Cela s'explique par l'absence de programme principal, c'est-à-dire, de lignes de code que l'interpréteur exécute lorsqu'on lance le script.

À l'inverse, que se passe-t-il alors si on importe un script en tant que module alors qu'il contient un programme principal avec des lignes de code ? Prenons par exemple le script `message2.py` suivant :

```
"""Script de test."""


def bonjour(nom):
    """Dit Bonjour."""
    return "Bonjour " + nom


# programme principal
print(bonjour("Joe"))
```

Si on l'importe dans l'interpréteur, on obtient :

```
>>> import message2
Bonjour Joe
```

Ceci n'est pas le comportement voulu pour un module car on n'attend pas d'affichage particulier (par exemple la commande `import math` n'affiche rien dans l'interpréteur).

Afin de pouvoir utiliser un code Python en tant que module ou en tant que script, nous vous conseillons la structure suivante :

```
"""Script de test."""


def bonjour(nom):
    """Dit Bonjour."""
    return "Bonjour " + nom


if __name__ == "__main__":
    print(bonjour("Joe"))
```

À la ligne 9, l'instruction `if __name__ == "__main__":` indique à Python :

- Si le programme `message2.py` est exécuté en tant que script dans un *shell*, le résultat du test `if` sera alors `True` et le bloc d'instructions correspondant (ligne 10) sera exécuté :

```
$ python message2.py
Bonjour Joe
```

- Si le programme `message2.py` est importé en tant que module, le résultat du test `if` sera alors `False` et le bloc d'instructions correspondant ne sera pas exécuté :

```
>>> import message2
>>>
```

À nouveau, ce comportement est possible grâce à la gestion des espaces de noms par Python (pour plus détails, consultez le chapitre 19 *Avoir la classe avec les objets*).

Au delà de la commodité de pouvoir utiliser votre script en tant que programme ou en tant que module, cela présente l'avantage de signaler clairement où se situe le programme principal quand on lit le code. Ainsi, plus besoin d'ajouter un commentaire `# programme principal` comme nous vous l'avions suggéré dans les chapitres 9 *Fonctions* et 12 *Plus sur les fonctions*. L'utilisation de la ligne `if __name__ == "__main__":` est une bonne pratique que nous vous recommandons !

14.7 Exercice

14.7.1 Module ADN

Dans le script `adn.py`, construisez un module qui va contenir les fonctions et constantes suivantes.

- Fonction `lit_fasta()` : prend en argument un nom de fichier sous forme d'une chaîne de caractères et renvoie la séquence d'ADN lue dans le fichier sous forme d'une chaîne de caractères.
- Fonction `seq_alea()` : prend en argument une taille de séquence sous forme d'un entier et renvoie une séquence aléatoire d'ADN de la taille correspondante sous forme d'une chaîne de caractères.
- Fonction `comp_inv()` : prend en argument une séquence d'ADN sous forme d'une chaîne de caractères et renvoie la séquence complémentaire inverse (aussi sous forme d'une chaîne de caractères).
- Fonction `prop_gc()` : prend en argument une séquence d'ADN sous forme d'une chaîne de caractères et renvoie la proportion en GC de la séquence sous forme d'un *float*. Nous vous rappelons que la proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases (A, T, C, G).
- Constante `BASE_COMP` : dictionnaire qui contient la complémentarité des bases d'ADN (`A` $\rightarrow$ `T`, `T` $\rightarrow$ `C`, `G` $\rightarrow$ `C` et `C` $\rightarrow$ `G`). Ce dictionnaire sera utilisé par la fonction `comp_inv()`.

À la fin de votre script, proposez des exemples d'utilisation des fonctions que vous aurez créées. Ces exemples d'utilisation ne devront pas être exécutés lorsque le script est chargé comme un module.

Conseils :

- Dans cet exercice, on supposera que toutes les séquences sont manipulées comme des chaînes de caractères en majuscules.

- Pour les fonctions `seq_alea()` et `comp_inv()`, n'hésitez pas à jeter un œil aux exercices correspondants dans le chapitre 11 *Plus sur les listes*.
- Voici un exemple de fichier FASTA `adn.fasta` pour tester la fonction `lit_fasta()`.