N° 23 FÉVRIER/MARS 2012

GNU LINUX
DÉCOUVRIR, COMPRENDRE ET UTILISER LINUX

PRATIQUE
HORS-SÉRIE

+ PYTHON EN PRATIQUE

- Rédigez et envoyez un e-mail
- Créez votre jeu de bataille navale !

L 12225 - 23 H - F: 6,50 € - RD

ENVIE DE DÉVELOPPER VOS PROPRES OUTILS ?

LA PROGRAMMATION AVEC PYTHON

NIVEAU : DÉBUTANT ET INTERMÉDIAIRE

TOUTES LES BASES DU LANGAGE POUR CRÉER RAPIDEMENT VOS PREMIERS PROGRAMMES !

1 PREMIERS PAS

- Historique et philosophie de Python
- Bien choisir son interpréteur et son éditeur
- Se familiariser avec les différents types de données
- La syntaxe de base

2 S'APPROPRIER UN LANGAGE

- Structures complexes et « slicing »
- Les fonctions et modules pour « alléger » vos programmes
- La bonne utilisation des expressions régulières

3 GÉRER LES FICHIERS

- Manipuler les fichiers en Python
- Créer et exploiter des archives...

4 LA PROGRAMMATION ORIENTEE OBJET

FRANCE MÉTRO. 6,50 € / DOM : 7,50 € / TOM Surface : 950 XPF / TOM AVION : 1200 XPF / BEL/LUX/PORT. CONT. : 7,50 € / CH : 13,80 CHF / CAN : 13 $CAD / MAR : 75 DH

ÉDITO

Dixit Wikipédia : « *Python est un langage de programmation multi-paradigme. Il favorise la programmation impérative structurée, et orientée objet.* » Euh... pardon ? Je vous l'accorde : quiconque n'a jamais écrit la moindre ligne de programmation pourra prendre peur rien qu'en lisant cette phrase... Toutefois, si vous tenez ce magazine entre les mains, c'est que l'envie d'apprendre à programmer vous titille et c'est déjà ça !

Comment se manifeste cette envie ? Dans la plupart des cas, elle est issue du bon vieil adage « on n'est jamais si bien servi que par soi-même ». Un programme, un logiciel vous convient, mais voilà : selon vous, il faudrait un petit peu plus de ceci ou un petit peu moins de cela pour l'améliorer. Qu'à cela ne tienne, pourquoi ne pas y remédier vous-même !?

Pourquoi s'initier à Python plutôt qu'à tout autre langage ? Eh bien Python est en effet bien souvent recommandé aux novices, de par la simplicité de sa syntaxe, sa rapidité, mais aussi grâce à la large documentation que l'on peut trouver sur Internet, sur laquelle vous pourrez vous appuyer en cas de problème. Pas de miracle : on ne devient pas un programmeur hors pair du jour au lendemain. Un (long) apprentissage est nécessaire avant de savoir pratiquer un peu de magie avec ses 10 doigts et son clavier.

Ce numéro hors-série de *Linux Pratique* a été conçu pour vous guider et vous soutenir dans votre démarche et par conséquent, s'avère tout à fait adapté au programmeur débutant. Cette sélection d'articles vous amènera ainsi petit à petit à la conception de vos propres programmes. De l'historique et des notions de base du langage, aux principes de la POO (programmation orientée objet), en passant par les fonctions à connaître et autres bonnes pratiques, la lecture de ce numéro spécial vous permettra d'assimiler progressivement les principes et techniques du langage Python. Pour terminer, vous pourrez asseoir et vérifier vos compétences via les quelques cas pratiques qui vous sont proposés en fin du magazine.

Car évidemment, on ne programme pas juste pour programmer, un projet concret se cache forcément derrière tout ça. À vous de déterminer l'objectif à atteindre...

Fleur Brosseau

SOMMAIRE

www.linux-pratique.com

Linux Pratique Hors-Série
est édité par **Les Éditions Diamond**

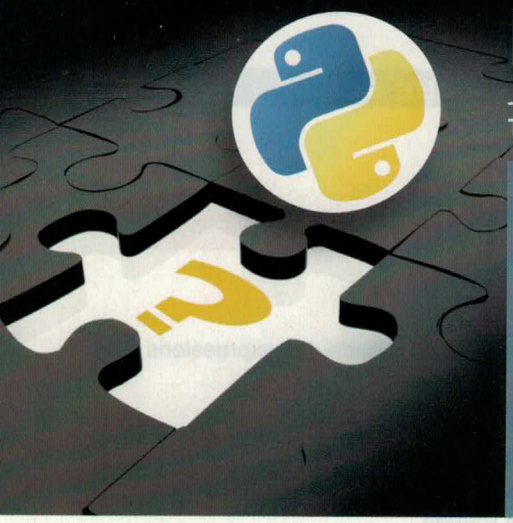
B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21
E-mail : cial@ed-diamond.com
lecteurs@linux-pratique.com
Service commercial : abo@linux-pratique.com
Sites : www.linux-pratique.com
www.ed-diamond.com

LES ÉDITIONS DIAMOND

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédactrice en chef : Fleur Brosseau
Secrétaire de rédaction : Véronique Sittler
Conception graphique : Kathrin Troeger
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : VPM Druck Rastatt / Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01

IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution
N° ISSN : 2101-6836
Commission Paritaire : K78 990
Périodicité : Bimestrielle
Prix de vente : 6,50 Euros

Membre April
www.april.org



PYTHON, C'EST QUOI ?

Tristan Colombo

Vous avez entendu parler de Python depuis de nombreuses années maintenant... mais vous ne connaissez toujours rien de ce langage ? Pourquoi un nouveau langage est-il apparu, à quels usages est-il destiné et surtout qu'est-ce que vous allez pouvoir en faire ? Dans cet article ouvrant ce hors-série consacré à Python, ni code, ni syntaxe, uniquement une présentation des fondamentaux historiques et philosophiques de Python.

La première version de Python a été écrite par Guido van Rossum en décembre 1989. De nombreux articles mettent en exergue le fait que cette version a été écrite en une semaine, mais il faut quelque peu relativiser les choses. Guido van Rossum travaillait déjà depuis quelques années à l'élaboration d'un autre langage de programmation (l'ABC) pour le compte du CWI (*Centrum voor Wiskunde en Informatica*) aux Pays-Bas. C'est de l'expérience acquise lors de ce projet que l'idée de créer Python lui est venue et au moment de se mettre au travail, il savait précisément ce qu'il voulait faire : d'où l'importance de l'étape de réflexion dans n'importe quel projet informatique ! Il a ainsi pu, en une semaine, poser les bases d'un langage utilisé aujourd'hui par des milliers de développeurs.

En 1991, la première version publique est publiée. Les versions s'enchaînent ensuite jusqu'en 2000 avec la sortie de Python 1.6.1 qui est la première version sous licence compatible GPL (*GNU Public Licence*).

En 2001 est créée la *Python Software Foundation* (PSF). Il s'agit d'une association à but non lucratif dont la mission est de promouvoir, protéger et étendre la communauté des utilisateurs du langage. C'est elle qui est en charge du développement du langage. Les propositions d'amélioration du langage sont effectuées par le biais de « Propositions d'Amélioration de Python » (ou PEP pour *Python Enhancement Proposal*). Il peut s'agir d'une proposition d'amélioration du langage d'un point de vue technique, ou d'une simple recommandation. Parmi ces recommandations, celle publiée par Tim Peters en 2004 [1] illustre bien la philosophie de Python. En voici une traduction :

« Préfère : la beauté à la laideur,
l'explicite à l'implicite,
le simple au complexe
et le complexe au compliqué,
le déroulé à l'imbriqué,
l'aéré au compact.
Prends en compte la lisibilité.
Les cas particuliers ne le sont jamais assez pour violer les règles.
Mais, à la pureté, privilégie l'aspect pratique.
Ne passe pas les erreurs sous silence,
... ou bâillonne-les explicitement.
Face à l'ambiguïté, à deviner ne te laisse pas aller.
Sache qu'il ne devrait y avoir qu'une et une seule façon de procéder,
même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.
Mieux vaut maintenant que jamais.
Cependant jamais est souvent mieux qu'immédiatement.
Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.
Les espaces de nommage ! Sacrée bonne idée ! Faisons plus de trucs comme ça. »

Cette PEP a même été enfouie dans le langage sous la forme d'un œuf de Pâques (*easter egg*) ou fonctionnalité cachée.

Au moment où ces lignes sont écrites, les dernières versions de Python publiées sont Python 2.7.2 le 11 juin 2011 et Python 3.2.2 le 14 septembre 2011. Nous verrons dans la suite pourquoi deux versions de Python coexistent.

Nous avons pu voir un historique rapide de la création de Python, mais nous ne savons toujours pas pourquoi son créateur a décidé de l'appeler comme ça. Guido van Rossum serait-il un grand amateur de reptiles ? Se passionnerait-il pour l'espèce des serpents les plus longs du monde ? En fait, non, Guido van Rossum est simplement fan des Monthy Pythons ! S'il avait préféré regarder les aventures de Simon Templar, nous pourrions programmer en « Saint » à l'heure actuelle...

Il serait maintenant intéressant de savoir ce que l'on peut faire avec ce langage...

1 Les usages de Python

Python est de plus en plus employé dans le cadre de l'apprentissage d'un premier langage de programmation à l'université. En effet, Python est un langage dit de haut niveau :

vous n'aurez par exemple pas à vous préoccuper de la gestion de l'espace mémoire pour y stocker des données. De plus, sa syntaxe est simple et il est très rapide d'apprendre à programmer en Python. L'objectif de ce langage est de diminuer les temps de développement... quitte à ce que les temps d'exécution soient plus longs qu'avec d'autres langages et qu'il faille procéder à quelques ajustements par la suite. Le développeur doit pouvoir se concentrer sur son objectif et non sur les subtilités du langage.

Python est également très utilisé pour le calcul scientifique. Il est en effet possible d'étendre le langage à l'aide de modules dédiés à des fonctions spécifiques. Il existe ainsi un module permettant d'effectuer simplement des calculs matriciels (numpy), de tracer des courbes (matplotlib), etc.

Avec l'évolution du langage, de nombreux *frameworks* sont apparus permettant d'utiliser Python dans de multiples contextes : pour les jeux avec PyGame [2][3], pour le développement web avec Django [4], etc.

Pour résumer, on peut donc tout faire avec Python à partir du moment où l'on inclut le ou les bon(s) module(s).

2 Les versions de Python

Nous avons vu en introduction que deux versions de Python coexistaient : la version 2.7.2 et la version 3.2.2. Vous vous demandez sans doute pourquoi deux versions d'un même langage sont toujours maintenues et pourquoi ne pas utiliser simplement la version dont le numéro est le plus élevé ?

Pour continuer à faire évoluer le langage, il a été décidé de casser la rétro-compatibilité des versions à partir de Python 3.x. Ainsi, un programme écrit en Python 2.x ne pourra pas forcément (en fonction du code employé) être exécuté en Python 3.x. Cette décision peut paraître très contraignante de prime abord, mais elle est nécessaire pour gommer certaines imperfections de jeunesse. D'autres langages n'ont pas pris ce risque et n'ont pas pu mener à terme les développements planifiés (voir par exemple le développement de PHP 6).

Quelles sont les conséquences pour le développeur ? De nombreux modules Python ont été écrits pour Python 2.x et n'ont pas encore été portés en Python 3.x, il est donc nécessaire de savoir coder en Python 2.x. D'un autre côté, tôt ou tard ces modules devront passer à Python 3.x (la maintenance de la branche 2.7 de Python est assurée pour l'instant, mais s'arrêtera un jour ou l'autre) et vous devrez donc connaître la syntaxe de Python 3.x. En cette période charnière, il paraît donc essentiel de connaître à la fois Python 2.7 et les différences existant par rapport à Python 3 pour pouvoir porter vos programmes par la suite (à moins bien entendu que vous n'utilisiez que des modules Python 3.x et il serait dans ce cas absurde de ne pas passer directement à Python 3.x).

Le cycle de maintenance de Python 2.7 sera plus long que la normale pour permettre un passage en douceur à Python 3.x. Il faut savoir également que certaines améliorations de Python 3.x sont tout de même portées en version 2.7.

On peut donc dire qu'il y a en fait trois versions : Python 2.6 que l'on retrouve sur les systèmes embarqués, Python 2.7 et Python 3.2. Dans ce hors-série nous utiliserons principalement Python 2.7 et, lorsqu'il existera des différences de syntaxe avec Python 3.2, nous les signalerons.

3 Un langage dynamique

Python est un langage à typage dynamique. Cela signifie que lorsque l'on définit une variable, nous n'avons pas à spécifier le type de donnée qu'elle contiendra (entier, chaîne de caractères, etc).

Parmi les langage à typage dynamique on distingue les langages faiblement typés et les langages fortement typés. Python fait partie de la seconde catégorie : le type d'une variable est fixé en fonction du contexte dans lequel elle est utilisée et des méthodes que l'on peut lui appliquer. Cette technique de typage s'appelle le *duck typing* pour « typage canard » qui doit son nom à l'analogie avec la détermination du type d'un oiseau dans la phrase suivante : « Si je vois un animal qui vole comme un canard, cancane comme un canard et nage comme un canard, alors j'appelle cet oiseau un canard ».

Avec un typage fort, l'interpréteur peut signaler des erreurs de type, ce qui n'est pas le cas avec un langage faiblement typé comme le PHP. En PHP, il est par exemple possible d'ajouter des entiers et des chaînes de caractères. L'opération `2 + "3 petits cochons"` renverra `5` comme résultat : PHP force l'interprétation numérique de la chaîne de caractères. Était-ce réellement le résultat attendu ou une erreur de codage du développeur qui n'était pas au courant de ce mécanisme ? En Python la question ne se posera pas et vous obtiendrez une erreur de typage.

4 Un langage interprété

Python est un langage interprété, c'est-à-dire que son code n'est pas compilé pour une architecture spécifique. Cela lui permet d'être multiplateforme : vous écrirez le même code, que vous souhaitiez exécuter votre programme sous Linux, Windows ou Mac OS X. Pour rappel, voici les différences entre un langage compilé et un langage interprété.

4.1. Langage compilé

Dans le cas d'un langage compilé, le code du programme, appelé « source » ou « code source », sera stocké dans des fichiers textes, puis passera par une étape de compilation : le code source sera transformé en instructions directement comprises par la machine sous la forme d'un fichier binaire exécutable. L'étape de compilation sera plus ou moins longue en fonction de l'architecture et de la longueur du programme, mais elle produira toujours un

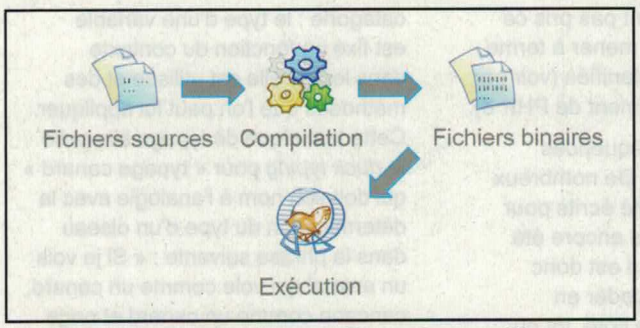


Fig. 1 : Processus d'exécution d'un code compilé

code adapté à la machine sur laquelle elle aura été exécutée. La figure 1 illustre les différentes étapes d'exécution d'un code issu d'un langage compilé. Le C et le C++ sont des exemples de langages compilés : ils sont très performants, mais le développement est plus long qu'avec un langage interprété.

4.2. Langage interprété

Dans le cas d'un langage interprété, le code source ne sera pas compilé mais sera interprété au fur et à mesure de son exécution. Cette méthode permet de s'affranchir de l'étape de compilation et ne provoque pas de crash (bien sûr des erreurs sont toujours possibles...). Par contre, suivant le code, on pourra remarquer des performances moins bonnes qu'avec un langage compilé. La figure 2 illustre les étapes de l'exécution d'un code issu d'un langage interprété. Perl est un exemple de langage interprété.

4.3. Langage semi-interprété

Dans le cas d'un langage semi-interprété, on va passer par une étape de compilation qui ne produira pas un code binaire mais un code intermédiaire, souvent appelé *byte code*, qui sera interprété dans un environnement spécifique : une machine virtuelle. Le langage modèle pour cette méthode est le Java. La figure 3 illustre les étapes de l'exécution d'un code issu d'un langage semi-interprété.

Python appartient en fait à cette sous-catégorie des langages interprétés. Contrairement à Java où l'étape de compilation est distincte de l'exécution, avec Python la compilation en byte code et l'interprétation sont effectuées à la volée. Vous pouvez retrouver le byte code des fichiers sources Python : le fichier source porte l'extension `.py` et après exécution, vous pourrez voir qu'un fichier de même nom mais portant l'extension `.pyc` est apparu : c'est le fichier de byte code.

Conclusion

En conclusion, je vous propose de voir en huit points les avantages et les inconvénients de Python.

1. Python permet un codage plus rapide : donc vous laisse plus de temps pour les tests et la documentation (je suppose ici que le temps de réflexion précédant le projet est incompressible et identique quel que soit le langage utilisé... en tout cas il devrait l'être).
2. Il existe de nombreux modules (plus de 10000) permettant d'étendre le langage.
3. Les crashes ne sont pas possibles puisqu'il s'agit d'un langage interprété.
4. Pour la même raison, il n'y a pas de perte de temps en compilations successives.
5. Venant toujours du fait que le langage est interprété, Python est multiplateforme.
6. Python est bien sûr gratuit et open source, distribué sous licence PSF – *Python Software Foundation* (compatible GPL).
7. Au niveau de la vitesse d'exécution, Python peut être dépassé par des langages compilés. Il existe toutefois une solution grâce à l'interopérabilité des langages : porter en C le morceau de code ralentissant l'exécution du programme et l'utiliser ensuite dans Python.
8. Python n'a pas été créé nativement pour intégrer la programmation parallèle... mais il existe une extension le permettant.

Vous savez maintenant ce qu'est Python et vous pouvez vous lancer dans la syntaxe de ce langage en ayant connaissance des mécanismes internes essentiels. ■

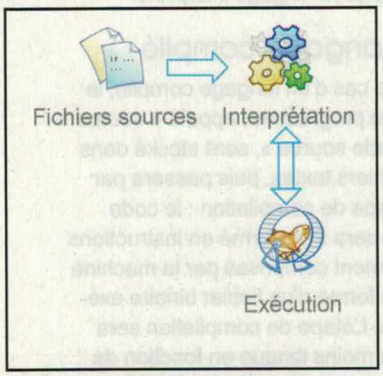


Fig. 2 : Processus d'exécution d'un code interprété

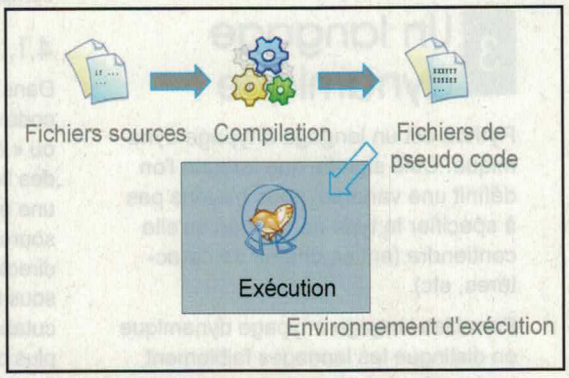


Fig. 3 : Processus d'exécution d'un code semi-interprété

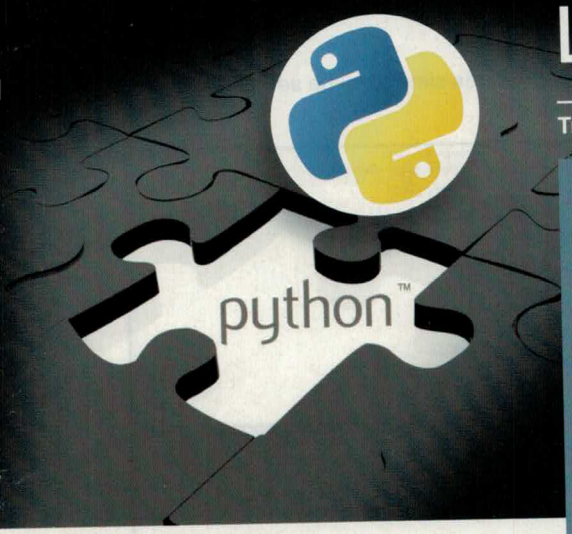
Références

[1] PEP 20 de Tim Peters sur le Zen de Python : http://www.python.org/dev/peps/pep-0020/

[2] Site officiel de PyGame : http://www.pygame.org

[3] COLOMBO (Tristan), « Créez un jeu en Python avec Pygame », GNU Linux Magazine n°117, juin 2009, p. 64 à 71.

[4] Site officiel de Django : https://www.djangoproject.com/



LES BASES DE PYTHON

Tristan Colombo

Après une introduction purement théorique, passons à la pratique et voyons quelle est la syntaxe de base utilisée en Python. Cet article constitue une introduction généraliste à la syntaxe de Python. Nous aborderons de nombreux sujets sans rentrer dans les détails, ces détails étant abordés dans les articles suivants. Nous commencerons par configurer un environnement de développement en installant des versions de Python (au moins un Python 2.x et un Python 3.x) et un éditeur de code avant d'appréhender les commandes Python en partant du type des données jusqu'à la structure d'un programme complet. Mais pour débuter, nous avons besoin de loger un Python dans notre ordinateur...

1 Installation des Pythons

Nous avons pu voir qu'il existait plusieurs versions de Python. Ces versions sont présentes dans les dépôts des distributions Linux majeures. Ainsi, si l'on utilise une distribution basée sur Debian, l'installation de Python 2.x se fera par :

```
sudo aptitude install python
```

Pour Python 3.x la commande sera :

```
sudo aptitude install python3
```

En utilisant ce mécanisme, rien ne nous assure que nous utilisons la dernière version. Et si nous voulons tester une ancienne version comme Python 2.6, cela sera impossible. Il faut alors passer par une installation manuelle en téléchargeant et compilant Python :

```
wget http://python.org/ftp/python/2.7.3/Python-2.7.2.tgz
tar -zxvf Python-2.7.2.tgz
cd Python-2.7.2
./configure
make
sudo make install
cd ..; rm -R Python-2.7.2
```

Lorsque vous voudrez sélectionner une version de Python particulière, vous utiliserez le nom complet de son exécutable. Par exemple, en tapant seulement `python` dans une console, vous passerez en mode interactif et vous obtiendrez l'invite de commande suivante :

```
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:13:53)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

On peut voir que la version installée par le gestionnaire de paquetages est ici la 2.7.1 (tapez `exit()` ou [Ctrl]+[d] pour sortir). Pour accéder à Python 3, tapez `python3`.

Si vous avez installé des versions de Python manuellement, celles-ci seront disponibles de la même manière mais en utilisant également le numéro de version complet (par exemple `python3.2` pour lancer Python 3.2).

2 Les différents interpréteurs

En tant que langage interprété, lorsque nous installons Python, nous installons un interpréteur. Il existe de nombreux interpréteurs Python écrits dans différents langages. L'interpréteur « classique » est écrit en C et s'appelle CPython. On peut aussi trouver un interpréteur écrit en Java, Jython, et un interpréteur écrit en... Python, avec PyPy. Nous ne rentrerons pas ici dans le détail de l'utilisation de ces interpréteurs, mais sachez que chacun d'eux dispose d'un mode interactif permettant de tester des commandes et qu'il existe des « sur-couches » à ce mode. On peut essentiellement distinguer trois interpréteurs interactifs :

- python : l'interpréteur interactif classique et basique intégré à Python ;
- IPython : un interpréteur interactif adapté à l'affichage en temps réel de courbes dessinées avec le module matplotlib (Fig. 1). Pour l'instant cet interpréteur n'existe qu'en version 2.7 et il est disponible dans les dépôts de toutes les distributions.

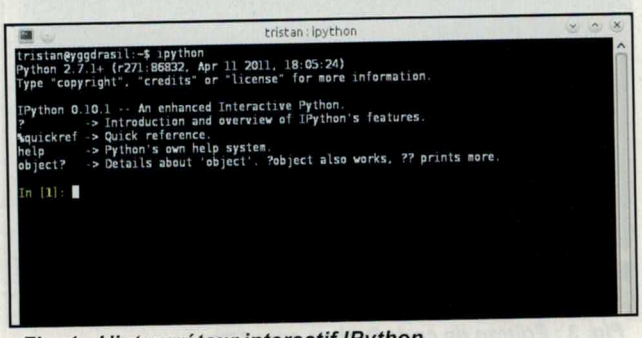


Fig. 1 : L'interpréteur interactif IPython

- BPython : un interpréteur interactif amélioré grâce à l'utilisation de la coloration syntaxique, la mise à disposition d'un historique des commandes, des propositions de complétion, etc. Pour l'instant cet interpréteur n'existe, lui aussi, qu'en version 2.7 et l'installation se fera simplement depuis les dépôts.

Fig. 2 : L'interpréteur interactif BPython

3 Quel éditeur choisir ?

Qui dit langage de programmation, dit production de code source... et un code source n'est jamais qu'un fichier texte contenant des instructions. Il va donc falloir utiliser un éditeur pour taper notre programme. Si vous avez déjà vos habitudes avec un éditeur, à partir du moment où il dispose de la coloration syntaxique, conservez-le ! S'il fallait changer d'éditeur à chaque nouveau langage, la perte de temps serait énorme ! Pour les autres, voici une liste de quelques éditeurs libres adaptés au développement en Python :

- l'incontournable Vim [1], un éditeur en mode console pour lequel il faut accepter de « perdre » du temps en apprentissage pour être très efficace par la suite (Fig. 3). Vim n'est pas spécifiquement adapté au développement sous Python.

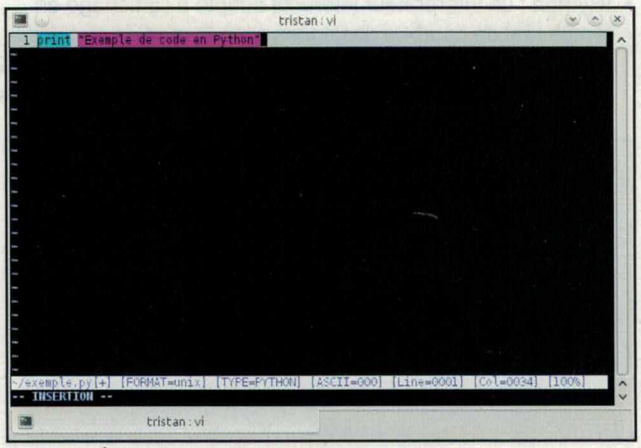


Fig. 3 : Édition de code Python avec Vim

- Geany, un petit éditeur très simple qui ne se focalise pas sur Python (Fig. 4).

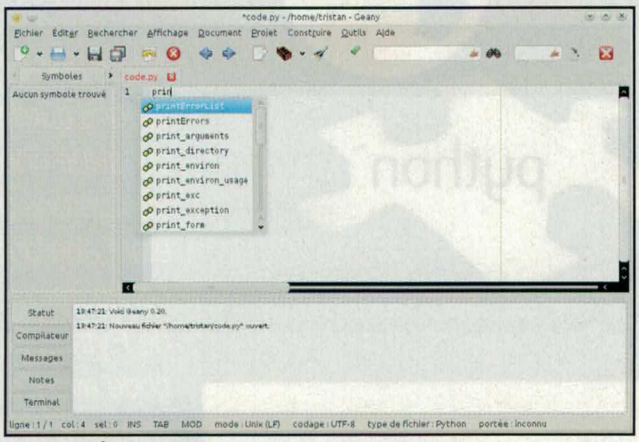


Fig. 4 : Édition de code Python avec Geany

- Spyder, un éditeur créé pour développer du Python avec, comme le montre la figure 5, un accès direct à l'interpréteur interactif IPython. Spyder intègre parfaitement les bibliothèques scientifiques de Python dans un environnement proche de Matlab et permet d'ailleurs d'échanger des données avec ce dernier. Cet éditeur est intégré au projet PythonXY [2] visant à fournir simplement un environnement complet de développement scientifique sous Python.

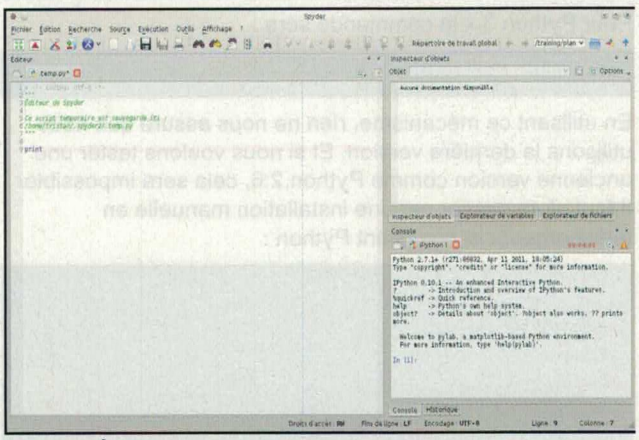


Fig. 5 : Édition de code Python avec Spyder

- Eclipse avec l'extension PyDev (Fig. 6, page suivante). On ne présente plus cet éditeur dont les fonctionnalités peuvent être améliorées à l'aide d'extensions et qui est utilisé par des milliers de développeurs. De mon point de vue, ce programme est bien trop lourd pour un éditeur (d'autant plus après avoir ajouté quelques extensions) et je préfère nettement Vim... mais rien ne vous empêche de le tester et éventuellement de l'adopter.

Maintenant que votre environnement de développement est prêt avec un Python fonctionnel et un éditeur de texte à disposition, nous allons pouvoir rentrer dans les entrailles du langage.

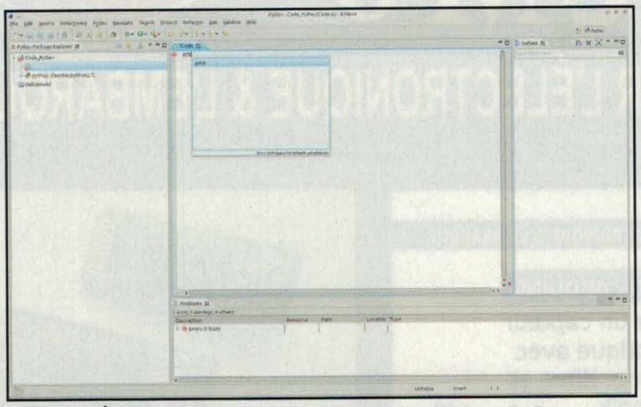


Fig. 6 : Édition de code Python avec Eclipse et PyDev

4 Hello World

Depuis quelques pages nous ne parlons que théorie et installation de Python... mais peut-être souhaitez-vous maintenant voir à quoi peut ressembler un programme en Python ? Voici le plus simple des programmes, le « Hello World ». Nous allons profiter de cet exemple pour mettre en valeur la concision de l'écriture de code en Python. Nous souhaitons donc simplement afficher dans une console le message « Hello World ! ». En C, nous aurons besoin de six lignes :

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n ");
  return 0;
}
```

Il faudra bien sûr ensuite compiler et exécuter le programme :

```
login@serveur:~$ gcc -Wall hello.c
login@serveur:~$ a.out
Hello World !
```

Cela représente beaucoup de travail pour un petit résultat ! Et encore, on peut faire encore plus long avec du Java :

```
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!\n");
  }
}
```

Ici nous devons créer une classe et donc utiliser la programmation orientée objet pour afficher un simple message ! Et nous devons bien sûr passer par une étape de pré-compilation :

```
login@serveur:~$ javac Hello.java
login@serveur:~$ java Hello
Hello World !
```

C'est toujours très long pour afficher un simple message... Voyons maintenant le code Python permettant d'obtenir le même résultat. À ce stade, nous utiliserons l'éditeur interactif en lançant `python` dans un terminal, puis en tapant nos commandes Python. Pour commencer, en Python 2.7 nous entrerons le code :

```
print "Hello World !"
```

Oui, c'est tout et ça paraît logique puisque tout ce que je demande c'est l'affichage de ce message ! Le résultat de l'interprétation de cette commande dans l'interpréteur interactif est montré en figure 7.

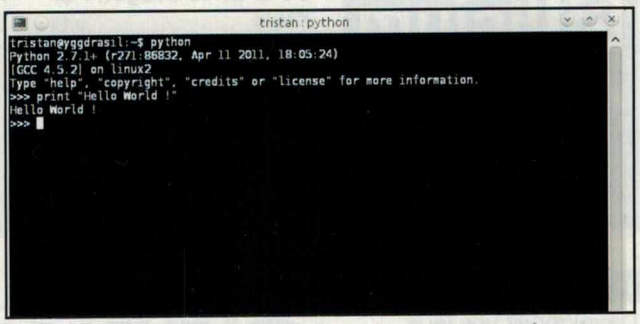


Fig. 7 : « Hello World » en Python dans un interpréteur interactif

En Python 3.2 cette commande sera légèrement différente et prendra la forme d'une fonction. Nous reviendrons sur le pourquoi de cette différence plus loin dans ce hors-série. Pour l'instant, sachez simplement que pour réaliser la même action que précédemment mais cette fois en Python 3.2, vous devrez taper :

```
python("Hello World !")
```

Remarquez au passage l'absence de point-virgule en fin de ligne, contrairement au C, à Java et à bon nombre de langages.

Maintenant que vous avez vu votre premier programme - ou script - en Python, certes court, mais programme quand même, je vous propose d'explorer les différents types de données manipulables en Python.

5 Les différents types de données

Nous avons vu que Python était un langage fortement typé utilisant le *duck typing* pour déterminer le type des données. Pour tester les types présentés dans cette partie, vous pourrez utiliser l'interpréteur interactif. C'est d'ailleurs sous cette forme que je présenterai les différents exemples en faisant précéder les commandes des caractères `>>>` correspondant au prompt de l'interpréteur.

Pour pouvoir travailler sur les types, nous aurons besoin de faire appel à une fonction : la fonction `type()`. Le rôle de cette fonction est d'afficher le type de l'élément qui lui est passé en paramètre.

5.1. Les données numériques

Parmi les données manipulables, on va bien entendu pouvoir utiliser des nombres. Ceux-ci sont classés en quatre types numériques :

- Entier : permet de représenter les entiers sur 32 bits (de - 2 147 483 648 à 2 147 483 647).

```
>>> type(2)
<type 'int'>
>>> type(2147483647)
<type 'int'>
```

Des préfixes particuliers permettent d'utiliser la notation octale (en base huit) ou hexadécimale (en base seize). Pour la notation octale, il faudra préfixer les entiers par le chiffre `0` en Python 2.7 et par `0o` (le chiffre zéro suivi de la lettre « o » en minuscule) en Python 3.2. Par exemple, `12` en notation octale vaut `8+2=10` :

```
>>> 012
10
```

Pour la notation hexadécimale, le préfixe sera le même que l'on soit sous Python 2.7 ou Python 3.2, ce sera `0x` (le chiffre zéro suivi de la lettre « x » en minuscule). Ainsi, `12` en notation hexadécimale vaut `16+2=18` :

```
>>> 0x12
18
```

- Entier long : tout entier ne pouvant être représenté sur 32 bits. On peut forcer un entier en entier long en le faisant suivre de la lettre `L` ou `l`.

```
>>> type(2147483647+1)
<type 'long'>
>>> type(2L)
<type 'long'>
```

- Flottant : pour les nombres flottants. Le symbole utilisé pour déterminer la partie décimale est le point. On peut également utiliser la lettre `E` ou `e` pour signaler un exposant en base 10 (101, 102, etc.).

```
>>> type(3.14)
<class 'float'>
>>> 2e1
20.0
>>> type(2e1)
<class 'float'>
```

- Complexe : pour les nombres complexes qui sont représentés par leur partie réelle et leur partie imaginaire. En Python, la partie imaginaire sera logiquement suivie de la lettre `J` ou `j` puisqu'en anglais imaginaire se dit... non l'explication n'est pas là... En informatique, on utilise souvent la variable `i` comme variable de boucle (dont nous parlerons par la suite). Pour éviter toute confusion, la partie imaginaire des complexes se note donc `j`. Voici comment écrire le complexe 2+3i :

```
>>> 2+3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
```

Revenons maintenant sur les flottants. L'interpréteur interactif peut être utilisé comme une calculette. Pour l'instant nous ne parlerons que des opérations élémentaires `+`, `-`, `*`, et `/` auxquelles nous ajouterons le quotient de la division euclidienne noté `//`. Voici quelques exemples de calculs :

```
>>> 3.14*2+1
7.28
>>> 3.14+5+2j
(8.14+2j)
>>> (15-5)*(3+4)
70
>>> 2+3j+2+5j
(4+8j)
```

Vous pouvez voir que la priorité des opérateurs est respectée et que les opérations entre nombres de types différents sont gérées (par exemple l'ajout d'un flottant et d'un complexe produit un complexe).

Au niveau du traitement de la division et de la division euclidienne, nous allons pouvoir noter des différences intéressantes entre Python 2.7 et Python 3.2. En effet, voici un exemple de tests de divisions en Python 2.7 :

```
>>> 7/6
1
>>> 7.0/6.0
1.1666666666666667
>>> 7//6
1
```

Les résultats dépendent du contexte ! Dans un contexte entier, la division donne pour résultat le quotient de la division euclidienne alors qu'elle se comporte normalement dans un contexte flottant. Effectuons les mêmes tests en Python 3.2 :

```
>>> 7/6
1.1666666666666667
>>> 7.0/6.0
1.1666666666666667
>>> 7//6
1
```

Les résultats paraissent ici logiques : `7/6` et `7.0/6.0` donnent le même résultat sous la forme d'un flottant et `7//6` a pour résultat `1` qui est bien le quotient de la division euclidienne de `7` par `6`.

Si vous utilisez les calculs en Python, l'usage de la division peut donc être source d'erreur et il faudra donc bien se souvenir de la version de Python employée. Mais il y a pire... et ce n'est pas l'apanage de Python. Le phénomène que je m'apprête à souligner est vrai pour absolument tous les langages, mais il est bien souvent oublié. Quel est le résultat d'une opération simple telle que `((0.7 + 0.1) x 10)` ?

Vous pensez que le résultat est 8 ? Testons donc en Python :

```
>>> ((0.7+0.1)*10)
7.999999999999999
```

On dirait qu'il y a un problème... En fait cela provient de la représentation des flottants en machine : ils sont arrondis [3] ! En Python, la solution consistera à utiliser un module spécial, le module `Decimal` [4]. Comme nous n'avons pas encore vu comment utiliser les modules, je ne m'étendrai pas sur ce problème, mais si vous manipulez des flottants pour des calculs précis, méfiez-vous !

En dehors des opérateurs élémentaires, les éléments numériques peuvent être assignés à des variables à l'aide du signe = (`variable = valeur`) et il est possible d'effectuer une opération en utilisant l'ancienne valeur d'une variable à l'aide de +=, -=, *= et /= (`variable += valeur` équivaut à `variable = variable + valeur`). Voici un petit exemple :

```
>>> a = 2
>>> a
2
>>> a += 3
>>> a
5
```

Les opérateurs de pré- et de post-incrémentation ou décrémentation ++ ou -- n'existent pas en Python. Par contre, vous avez la possibilité d'effectuer des affectations de variables multiples avec une même valeur :

```
>>> a = b = 1
>>> a
1
>>> b
1
```

Et vous pouvez également effectuer des affectations multiples de variables ayant des valeurs différentes en séparant les noms de variables et les valeurs par des virgules :

```
>>> a, b = 2, 3
>>> a
2
>>> b
3
```

Ce mécanisme peut paraître anodin, mais il permet de réaliser très simplement une permutation de valeurs sans avoir recours à une variable intermédiaire :

```
>>> a = 2
>>> b = 3
>>> a, b = b, a
>>> a
3
>>> b
2
```

Enfin, un opérateur intéressant pour élever un nombre à une puissance quelconque : l'opérateur **.

```
>>> 2**3
8
```

Après avoir vu les types numériques, voyons maintenant comment stocker des chaînes de caractères.

5.2. Les chaînes de caractères

Les chaînes de caractères sont encadrées par des apostrophes ou des guillemets. Le choix de l'un ou de l'autre n'a aucun impact au niveau des performances. La concaténation (assemblage de plusieurs chaînes pour n'en produire qu'une seule) se fait à l'aide de l'opérateur +. Enfin, l'opérateur * permet de répéter une chaîne. Voyons cela sous forme d'exemples :

```
>>> a = "Hello "
>>> b = 'World !'
>>> a+b
'Hello World !'
>>> 3*a
'Hello Hello Hello '
```

L'accès à un caractère particulier de la chaîne se fait en indiquant sa position entre crochets (le premier caractère est à la position 0) :

```
>>> a[0]
'H'
```

En Python, on peut faire beaucoup de choses avec les chaînes de caractères... nous verrons cela plus en détail dans l'article consacré aux chaînes et aux listes.

5.3 Les booléens

Les valeurs booléennes sont notées `True` et `False` (attention à la majuscule). Dans le cadre d'un test, les valeurs 0, "" (la chaîne vide) et `None` sont considérées comme étant égales à `False`. Ces valeurs sont retournées comme résultats des tests réalisés à l'aide des opérateurs de comparaison : == pour l'égalité, != pour la différence, puis <, >, <=, et >=. Exemple :

```
>>> 2 == 3
False
>>> 2 <= 3
True
```

Pour combiner les tests, on utilise les opérateurs booléens : `and` (et), `or` (ou) et `not` (non). Voici des exemples de combinaisons de tests :

```
>>> 0 and True
0
>>> 1 and True
True
>>> not (0 or True)
False
```

5.4. Les listes, les tuples et les dictionnaires

Nous allons maintenant voir trois types particuliers présentant de fortes similitudes : les listes, les tuples et les dictionnaires. Ces types étant un peu complexes, il ne s'agit ici que d'un premier aperçu, nous les étudierons plus en détail dans l'article leur étant consacré.

5.4.1. Les listes

Une liste est un ensemble d'éléments éventuellement de différents types. Une liste se définit à l'aide des crochets et une liste vide est symbolisée par `[]`. L'accès à un élément de la liste se fait en donnant son index, de la même manière qu'avec les chaînes de caractères :

```
>>> maListe = [ 3.14, "Linux Pratique",
2+3j, 5 ]
>>> maListe[1]
'Linux Pratique'
```

5.4.2. Les tuples

Les tuples sont des listes particulières dont nous verrons l'intérêt plus tard. Ils

sont définis par des parenthèses et leurs éléments sont accessibles de la même manière que pour les listes. Notez que pour lever toute ambiguïté, un tuple ne contenant qu'un seul élément sera noté `(élément,)`. Si vous omettez la virgule, Python pensera qu'il s'agit d'un parenthésage superflu et vous n'aurez donc pas créé un tuple. L'exemple des listes peut tout à fait être rapporté aux tuples :

```
>>> monTuple = ( 3.14, "Linux Pratique", 2+3j, 5 )
>>> monTuple[1]
'Linux Pratique'
```

5.4.3. Les dictionnaires

Pour en finir avec notre aperçu des types complexes, voici les dictionnaires. Un dictionnaire est une liste où l'accès aux éléments se fait à l'aide d'une clé alphanumérique ou purement numérique. Il s'agit d'une association clé/valeur sous la forme `clé:valeur`. Les dictionnaires sont définis à l'aide des accolades. Dans d'autres langages les dictionnaires sont aussi appelés tableaux associatifs ou tables de hachage. Voici un exemple d'utilisation d'un dictionnaire :

```
>>> t = { 'nom' : 'toto', 'prenom' : 'titi' }
>>> t['nom']
'toto'
```

Ce type de données conclut notre tour des types de données en Python. Comment articuler maintenant plusieurs instructions pour écrire un programme ?

6 La syntaxe générale

Dans tous les langages, l'indentation du code (c'est-à-dire la manière d'écrire le code en laissant des espaces de décalage en début des lignes) est laissée au choix éclairé du développeur. Mais, force est de le constater, parfois le développeur est un sombre idiot... Qui n'a jamais vu de code ressemblant au programme suivant (exemple en PHP) ?

```
if ($a == 1) {
  echo 'Salut !'; }
else
{ echo 'Au revoir !'; }
```

Ce code peut aussi être écrit différemment :

```
if ($a==1)
echo 'Salut !';
else echo 'Au revoir !';
```

Python oblige donc le développeur à structurer son code à l'aide des indentations : ce sont elles qui détermineront les blocs (séquences d'instructions liées) et non les accolades comme dans la majorité des langages. Les blocs de code sont déterminés par :

- la présence du caractère `:` en fin de ligne ;
- une indentation des lignes suivantes à l'aide de tabulations ou d'espaces (attention à ne pas mélanger les deux, Python n'aime pas ça du tout : votre code ne fonctionnera pas et vous n'obtiendrez pas de message d'erreur explicite).

La figure 8 illustre le mécanisme des blocs en Python.

Enfin, il n'y a pas de point-virgule en fin de ligne. Même si ce caractère peut-être utilisé pour séparer des instructions sur une même ligne, son usage est déconseillé. Le caractère `#` permet d'indiquer que tous les caractères suivants sur la même ligne ne doivent pas être interprétés : il s'agit de commentaires.

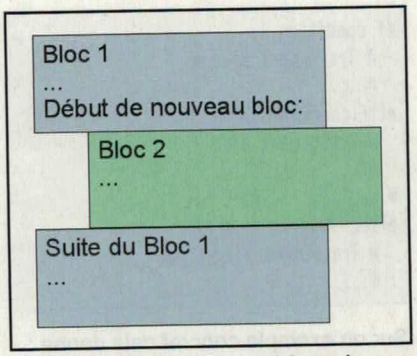


Fig. 8 : Structuration en blocs dans un script Python

```
>>> a = 12 # Affectation d'un entier
```

Pour pouvoir créer réellement un script, il nous faut des structures permettant d'effectuer des tests et des traitements répétitifs. C'est ce que nous allons voir dans la suite.

7 Les structures de boucle et de test

Nous séparerons ici les tests et les boucles.

7.1. Les structures de test

Il n'existe en fait qu'une seule structure de test en Python : le test « si ... alors ... » utilisant les instructions `if` et `else`. Au niveau de la construction du test, il faudra vérifier une condition et si cette dernière est vraie exécuter un bloc (et donc penser aux deux-points en fin de ligne et à l'indentation). La structure d'un test est donc :

```
if condition:
  # Traitement bloc 1
  # ...
else:
  # Traitement bloc 2
  # ...
```

Pour les tests multiples, il faudra enchaîner les `if` en cascade grâce à l'instruction `elif`, contraction de *else if* :

```
if condition_1:
  # Traitement bloc 1
  # ...
elif condition_2:
  # Traitement bloc 2
  # ...
# ...
else:
  # Traitement bloc final
  # ...
```

Sur un exemple concret cela donne :

```
>>> a=3
>>> if a>1:
...   print 'a>1'
... else:
...   print 'a<=1'
...
a>1
```

Notez qu'en mode interactif, Python affiche des points de suspension pour indiquer que votre commande est incomplète car un bloc a été ouvert mais pas fermé. Il faut appuyer sur <Entrée> dans la dernière ligne pour fermer le bloc.

7.2. Les structures de boucle

Pour les boucles il existe deux structures. La première est le `for` qui prend ses valeurs dans une liste : `for variable in liste_valeurs`. À chaque itération la variable prendra pour valeur un élément de la liste. Voici un exemple :

```
>>> for e in ['pomme', 'poire', 'kiwi']:
...   print e
...
pomme
poire
kiwi
```

Pour créer une boucle équivalente aux boucles traditionnelles « pour i variant de n à m, faire... », nous utiliserons une astuce. En Python 2.7, la fonction `range(a, b)` crée une liste d'entiers de `a` à `b-1` :

```
>>> range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

En Python 3.2 le type n'est pas le même mais, en première approche, tout fonctionne de la même manière (vous ne pourrez simplement pas afficher la liste dans l'exemple ci-dessus). En utilisant cette fonction, on peut donc mimer le comportement d'une boucle `for` traditionnelle avec une variable qui prendra ses valeurs entre deux entiers donnés :

```
>>> for i in range(0,4):
...   print i
...
0
1
2
3
```

Le mot-clé `continue` permet de passer à l'itération suivante et le mot-clé `break` permet de sortir de la boucle :

```
>>> for i in range(0,10):
...   if i==2:
...     continue
...   if i==4:
...     break
...   print i
...
0
1
3
```

La seconde structure de boucle est le « tant que » : tant qu'une condition est vraie, on boucle. Attention : dans ce type de boucle on utilise une variable de boucle dont la valeur doit changer pour pouvoir sortir de la boucle :

```
>>> i = 0
>>> while i<4:
...   print i
...   i += 1 # équivalent à i = i + 1
...
0
1
2
3
```

La boucle `while` accepte elle aussi les mots-clés `continue` et `break`.

8 La structure d'un programme

Avec l'ensemble des instructions que nous avons vu, vous pouvez écrire des programmes plus longs que les quelques lignes testées dans l'interpréteur interactif. Vous allez donc avoir besoin de stocker vos scripts dans des fichiers d'extension `.py`. Pour exécuter ces fichiers de code vous aurez alors deux possibilités :

- soit lancer simplement dans un terminal :

```
python monFichier.py
```

- soit ajouter à votre code source une ligne de shebang indiquant où se trouve l'interpréteur Python à utiliser :

```
#!/usr/bin/python
```

Cette ligne doit être la première ligne de votre fichier. Ensuite vous n'avez plus qu'à rendre votre fichier exécutable (`chmod u+x monFichier.py`) et vous pourrez lancer votre script par :

```
monFichier.py
```

Conclusion

Cet article a permis d'avoir un aperçu global de ce qu'était Python. Avec les connaissances acquises vous devriez être capable de réaliser déjà de petits scripts. Les articles suivants vont vous permettre d'approfondir vos connaissances sur des points précis du langage. ■

Références

[1] COLOMBO (Tristan), « Préparer son système : configurer l'éditeur Vim », Linux Pratique HS n°22, Octobre 2011, p. 12 à 17.

[2] Site officiel de PythonXY : http://www.pythonxy.org

[3] COLOMBO (Tristan), « Au-delà des Réels, l'aventure continue... », Linux Magazine n°113, Février 2009, p. 60 à 63.

[4] Documentation du module Decimal : http://docs.python.org/library/decimal.html

LE SLICING ET LES STRUCTURES DE LISTE

Tristan Colombo

Cet article constitue un approfondissement sur le fonctionnement des listes, des tuples et des dictionnaires. Nous apporterons une attention particulière à la technique du « slicing » ou « découpage en tranches ».

Dans les articles précédents vous avez pu vous familiariser avec différents types de données Python. Nous avons notamment vu des structures complexes permettant de stocker d'autres variables : les listes et les tuples. Je vous propose d'approfondir vos connaissances sur ces structures et d'étudier la technique du « slicing » qui permet de spécifier de manière simple des plages de données que l'on souhaite utiliser. Pour commencer, revenons sur une liste particulière : la chaîne de caractères.

1 Les spécificités des chaînes de caractères

Une chaîne de caractères est une liste particulière ne contenant que des caractères. Comme pour une liste, on peut accéder à ses éléments (les caractères) en spécifiant un indice :

```
>>> chaine = 'Linux Pratique'
>>> chaine[0]
'L'
```

Par contre, il est impossible de modifier une chaîne de caractères ! On dit alors qu'il s'agit d'une liste non modifiable :

```
>>> chaine[1] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Vous pourriez avoir la sensation de modifier une chaîne de caractères si vous décidiez de lui ajouter des caractères en fin de chaîne à l'aide d'une concaténation du type suivant :

```
>>> chaine = chaine + '!'
```

En fait, ici, nous réalisons une nouvelle affectation : la variable `chaine` est écrasée (effacée puis remplacée) par la valeur `chaine + '!'`. Il ne s'agit pas d'une modification au sens propre du terme.

Les chaînes de caractères admettent une écriture particulière appelée « formatage ». La syntaxe à employer est légèrement différente qu'il s'agisse de Python 2.7 ou Python 3.2. Le formatage consiste à écrire une chaîne de caractères en y insérant des variables dont nous donnerons la valeur par la suite. On peut voir ce mécanisme comme dans un texte à trous pour lequel la réponse serait donnée a posteriori. En Python 2.7 on utilisera les expressions de formatage issues du C :

- `%d` pour désigner un entier ;
- `%f` pour un flottant ;
- `%s` pour une chaîne de caractères ;
- `%.4f` pour forcer l'affichage de 4 chiffres après la virgule ;
- `%02d` pour forcer l'affichage d'un entier sur deux caractères ;
- etc.

Vous trouverez une liste complète des expressions reconnues sur la page : http://docs.python.org/library/string.html. Pour utiliser ce formatage, il faudra donner une chaîne de caractères contenant des expressions de formatage et faire suivre cette chaîne du caractère `%` et d'un tuple contenant les valeurs de remplacement :

```
>>> c = 2
>>> d = 3
>>> '%d * %d = %d' % (c, d, c*d)
'2 * 3 = 6'
```

En Python 3.2, les indications de formatage sont données par `{}` sans indication de type. Il est possible d'utiliser les expressions vues précédemment mais en utilisant le caractère `:` à la place de `%`. Cette fois, le formatage est réalisé par la méthode `format()` qui s'applique à une chaîne de caractères :

```
>>> '{} * {} = {}'.format(c, d, c*d)
'2 * 3 = 6'
>>> chaine = 'Un entier sur deux chiffres : {:02d}'
>>> chaine.format(6)
'Un entier sur deux chiffres : 06'
```

2 Les spécificités des tuples

Les tuples sont également des listes non modifiables (les chaînes de caractères sont donc en fait des tuples particuliers) :

```
>>> t = (1, 2, 3)
>>> t[0]
1
>>> t[0] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Attention de ne pas utiliser le mot-clé `tuple` comme nom de variable : ce dernier permet de créer un tuple de manière explicite. Il vous faudra utiliser en paramètre... une liste. Il permet donc d'effectuer une conversion :

```
>>> l = [1, 2, 3]
>>> t = tuple(l)
>>> t
(1, 2, 3)
>>> l
[1, 2, 3]
>>> t = tuple('1234')
>>> t
('1', '2', '3', '4')
```

Tout comme avec les chaînes, il sera possible de concaténer des tuples et avoir la sensation d'avoir modifié un tuple alors que nous aurons simplement réaffecté une variable :

```
>>> t = (1, 2, 3)
>>> t = t + (4, 5, 6)
>>> t
(1, 2, 3, 4, 5, 6)
```

Pour rappel, un tuple ne contenant qu'un seul élément est noté entre parenthèses, mais avec une virgule précédant la dernière parenthèse :

```
>>> t = (1,)
>>> t
(1,)
```

Quel peut être l'intérêt d'utiliser ce type plutôt qu'une liste ? Tout d'abord, les données ne peuvent pas être modifiées par erreur mais surtout, leur implémentation en machine fait qu'elles occupent moins d'espace mémoire et que leur traitement par l'interpréteur est plus rapide que pour des valeurs identiques mais stockées sous forme de listes. De plus, de par leur aspect non modifiable, les valeurs contenues dans un tuple peuvent être utilisées en tant que clé pour accéder à une valeur dans un dictionnaire (alors que c'est beaucoup plus dangereux avec les valeurs contenues dans une liste) :

```
>>> l = ['cle1', 'cle2', 'cle3']
>>> t = ('cle1', 'cle2', 'cle3')
>>> d = { 'cle1' : 1, 'cle2' : 2 }
>>> d[l[0]]
1
>>> l[0] = 'cle_modifiee'
>>> d[l[0]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cle_modifiee'
```

3 Les spécificités des listes

Nous avons vu que les listes étaient des éléments modifiables pouvant contenir différents types de données. Il est bien sûr possible de modifier les éléments d'une liste :

```
>>> l = [ 1, 2, 3, 4]
>>> l
[1, 2, 3, 4]
>>> l[0] = 0
>>> l
[0, 2, 3, 4]
```

Plusieurs méthodes peuvent être employées pour ajouter des éléments à une liste existante (sans réaffectation). La méthode `append()` ajoute un élément en fin de liste et la méthode `insert()` permet de spécifier l'indice où insérer un élément :

```
>>> l.append(5)
>>> l
[0, 2, 3, 4, 5]
>>> l.insert(1, 1)
>>> l
[0, 1, 2, 3, 4, 5]
```

La méthode `extend()` permet de réaliser la concaténation de deux listes sans réaffectation. Cette opération est réalisable avec réaffectation en utilisant l'opérateur `+` :

```
>>> l1 = [ 1, 2, 3 ]
>>> l2 = [ 4, 5, 6 ]
>>> l1.extend(l2)
>>> l1
[1, 2, 3, 4, 5, 6]
>>> l1 = [ 1, 2, 3 ]
>>> l1 = l1 + l2
>>> l1
[1, 2, 3, 4, 5, 6]
```

Pour supprimer un élément, on peut utiliser la méthode `remove()` qui supprime la première occurrence de la valeur passée en paramètre ou la commande `del` qui supprime un élément précis en fonction de son indice :

```
>>> l = [ 1, 2, 3, 1 ]
>>> l.remove(1)
>>> l
[2, 3, 1]
>>> l.remove(1)
>>> l
[2, 3]
>>> l.remove(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> l = [ 1, 2, 3, 1 ]
>>> del l[3]
>>> l
[1, 2, 3]
```

Pour savoir si un élément appartient bien à une liste on utilise le mot-clé `in`. Si l'élément testé est dans la liste, la valeur retournée sera `True` et sinon ce sera `False` :

```
>>> distrib = [ 'debian', 'ubuntu', 'fedora' ]
>>> 'debian' in distrib
True
>>> 'mandriva' in distrib
False
```

Il est également possible d'obtenir l'indice de la première occurrence d'un élément par la méthode `index()` :

```
>>> distrib.index('ubuntu')
1
```

Pour connaître la taille d'une liste (nombre d'éléments qu'elle contient), on pourra utiliser la fonction `len()` :

```
>>> l = [ 1, 2, 3 ]
>>> len(l)
3
```

Mais attention : cette fonction ne compte que les éléments de « premier niveau » ! Si vous avez une structure de liste complexe contenant des sous-listes, chaque sous-liste ne comptera que comme un seul élément :

```
>>> l = [ 1, 2, [ 'a', 'b', [ '000', '001', '010' ] ] ]
>>> len(l)
3
```

Ce problème du décompte des sous-listes nous amènera à un autre problème : celui de la copie des listes que nous aborderons après le slicing.

L'accès aux éléments d'une sous-liste s'effectue en spécifiant l'indice de la sous-liste suivi de l'indice de l'élément souhaité dans la sous-liste, etc. En reprenant la liste utilisée dans l'exemple précédent :

```
>>> l[0]
1
>>> l[2]
['a', 'b', ['000', '001', '010']]
>>> l[2][0]
'a'
>>> l[2][2]
['000', '001', '010']
>>> l[2][2][1]
'001'
```

Pour compléter cet approfondissement sur les listes, sachez que Python implémente un mécanisme appelé « compréhension de listes », permettant d'utiliser une fonction qui sera appliquée sur chacun des éléments d'une liste. Voici un exemple :

```
>>> l = [ 0, 1, 2, 3, 4, 5 ]
>>> carre = [ x**2 for x in l ]
>>> carre
[0, 1, 4, 9, 16, 25]
```

À l'aide de l'instruction `for x in l`, on récupère un à un les éléments contenus dans la liste `l` et on les place dans la variable `x`. On élève ensuite chacune des valeurs au carré (`x**2`) dans un contexte de liste, ce qui produit une nouvelle liste. Ce mécanisme peut produire des résultats plus complexes. Il peut également être utilisé avec les dictionnaires.

4 Les spécificités des dictionnaires

Comme nous l'avons vu, les dictionnaires sont des listes d'éléments indicés par des clés. Comme pour les listes, la commande `del` permet de supprimer un élément et le mot-clé `in` permet de vérifier l'existence d'une clé :

```
>>> d = { 'cle_1' : 1, 'cle_2' : 2, 'cle_3' : 3 }
>>> del d['cle_2']
>>> d
{'cle_3': 3, 'cle_1': 1}
>>> 'cle_1' in d
True
>>> 'cle_2' in d
False
```

Vous aurez pu remarquer lors de l'affichage du dictionnaire `d` que les couples clé/valeur n'étaient pas affichés dans l'ordre de création. C'est tout à fait normal car les dictionnaires ne sont pas ordonnés !

Plusieurs méthodes permettent de récupérer des listes de clés, de valeurs et de couples clé/valeur :

```
>>> courses = { 'pommes' : 3, 'poires' : 5, 'kiwis' : 7 }
>>> courses.keys()
['poires', 'kiwis', 'pommes']
>>> courses.values()
[5, 7, 3]
>>> courses.items()
[('poires', 5), ('kiwis', 7), ('pommes', 3)]
```

Toutes ces méthodes renvoient des listes et la dernière méthode renvoie une liste de tuples contenant la clé et la valeur. Toutefois attention : les résultats présentés ci-dessus sont ceux obtenus en Ptyhon 2.7. En Python 3.2 les valeurs de retour sont légèrement différentes :

```
>>> courses = { 'pommes' : 3, 'poires' : 5, 'kiwis' : 7 }
>>> courses.keys()
dict_keys(['poires', 'kiwis', 'pommes'])
>>> courses.values()
dict_values([5, 7, 3])
>>> courses.items()
dict_items([('poires', 5), ('kiwis', 7), ('pommes', 3)])
```

Ces méthodes renvoient des objets itérables : ce ne sont pas des listes et ils ne consomment donc pas autant de place mémoire que pour stocker une liste, on ne stocke qu'un pointeur vers l'élément courant. Avec ces structures vous pourrez toujours utiliser les boucles `for` classiques, par contre vous n'aurez plus un accès direct aux valeurs en spécifiant leur indice.

Dans un dictionnaire, pour obtenir la valeur correspondant à une clé on peut bien entendu utiliser la notation classique `dictionnaire[clé]`, mais on peut aussi utiliser la méthode `get()` qui renvoie la valeur associée à la clé passée en premier paramètre ou, si elle n'existe pas, la valeur passée en second paramètre :

```
>>> courses = { 'pommes' : 3, 'poires' : 5, 'kiwis' : 7 }
>>> courses.get('pommes', 'stock epuise')
3
>>> courses.get('salades', 'stock epuise')
'stock epuise'
```

On peut fusionner deux dictionnaires avec la méthode `update()`. Notez que si une clé existe déjà, la valeur stockée sera écrasée :

```
>>> courses = { 'pommes' : 3, 'poires' : 5, 'kiwis' : 7 }
>>> courses_2 = { 'kiwis' : 3, 'salades' : 2 }
>>> courses.update(courses_2)
>>> courses
{'poires': 5, 'salades': 2, 'kiwis': 3, 'pommes': 3}
```

Enfin, en ce qui concerne la copie de dictionnaires, le problème sera le même qu'avec les listes et il sera expliqué plus loin dans cet article.

5 Le slicing

Le slicing est une méthode applicable à tous les objets de type liste ordonnée (donc pas aux dictionnaires). Il s'agit d'un « découpage en tranches » des éléments d'une liste de manière à récupérer des objets respectant cette découpe. Pour cela, nous devrons spécifier l'indice de l'élément de départ, l'indice de l'élément d'arrivée (qui ne sera pas compris dans la plage) et le pas de déplacement. Pour une variable `v` donnée, l'écriture se fera en utilisant la notation entre crochets et en séparant chacun des paramètres par le caractère deux-points : `v[début:fin:pas]`. Cette écriture peut se traduire par : « les caractères de la variable `v` depuis l'indice `début` jusqu'à l'indice `fin` non compris avec un déplacement de `pas` caractère(s) ».

Pour bien comprendre le fonctionnement du slicing, nous commencerons par l'appliquer aux chaînes de caractères avant de voir les listes et les tuples.

5.1. Le slicing sur chaînes de caractères

Voici un premier exemple simple, illustré par la figure 1 :

```
>>> c = 'Linux Pratique'
>>> c[0:5]
'Linux'
>>> c[6:14]
'Pratique'
```

Vous avez remarqué que dans ce code aucune indication de pas n'a été donnée : c'est la valeur par défaut qui est alors utilisée, c'est-à-dire 1. De même, si la valeur de début est omise, la valeur par défaut utilisée sera 0 et si la valeur de fin est omise, la valeur par défaut utilisée sera la taille de la chaîne + 1 (il ne faut pas oublier que l'indice de fin indique la première lettre qui ne sera pas affichée).

```
>>> c[:5] # équivaut à c[0:5], équivaut à c[0:5:1]
'Linux'
>>> c[6:] # équivaut à c[6:14], équivaut à c[6:14:1]
'Pratique'
```

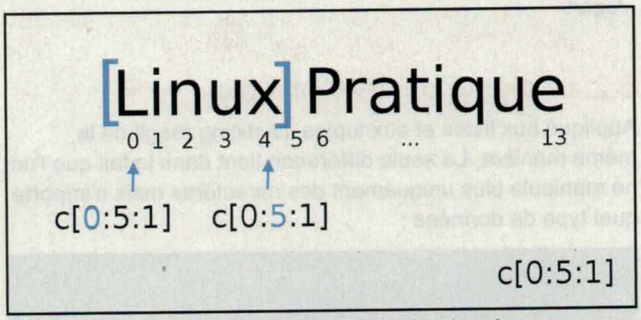


Fig. 1 : Slicing sur la chaîne c='Linux Pratique'

Du fait de ces valeurs par défaut, que pensez-vous que l'on sélectionne en tapant : `c[:]` ou encore `c[::]` ? La chaîne entière, bien sûr :

```
>>> c[:]
'Linux Pratique'
```

Il devient alors très simple d'inverser une chaîne en utilisant le pas :

```
>>> c[::-1]
'euqitarP xuniL'
```

On sélectionne toute la chaîne et on la parcourt avec un pas de -1 (donc à l'envers).

Il faut savoir également qu'en Python les listes sont indicées en nombres positifs de la gauche vers la droite et en nombres négatifs de la droite vers la gauche (voir figure 2). On peut ainsi très facilement accéder aux derniers caractères d'une chaîne sans connaître sa taille :

```
>>> c[-8:]
'Pratique'
```

Attention : si vous donnez comme intervalle [-8:-1] vous n'aurez pas la dernière lettre, et si vous donnez [-8:0] vous n'obtiendrez rien puisqu'il est impossible d'aller de -8 à 0 avec un pas de 1 (la dernière lettre étant indicée -1).

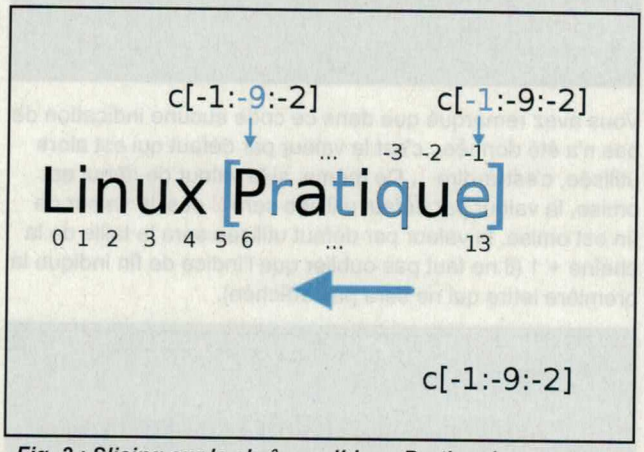


Fig. 2 : Slicing sur la chaîne c='Linux Pratique' en partant de la droite (indice -1 jusqu'à -9) avec un pas de -2 : 'eqtr'.

5.2. Le slicing sur listes et tuples

Appliqué aux listes et aux tuples, le slicing réagit de la même manière. La seule différence tient dans le fait que l'on ne manipule plus uniquement des caractères mais n'importe quel type de données :

```
>>> distrib = [ 'suse', 'debian', 'mandriva', 'ubuntu' ]
>>> distrib[1:3]
['debian', 'mandriva']
>>> distrib[-1::-2]
['ubuntu', 'debian']
>>> t_distrib = ('suse', 'debian', 'mandriva', 'ubuntu')
>>> t_distrib[::-1]
('ubuntu', 'mandriva', 'debian', 'suse')
```

Pour les listes, en tant qu'élément modifiable, nous pouvons utiliser le slicing pour ajouter ou retirer des éléments : il suffit de déterminer une tranche et d'indiquer par quelle liste la remplacer. Pour insérer un élément à un indice précis, il faudra donner cet indice en tant que paramètre de début et de fin :

La figure 3 illustre l'utilisation du slicing sur les listes.

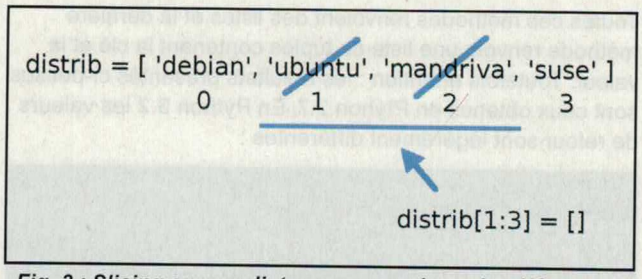


Fig. 3 : Slicing sur une liste pour supprimer des éléments.

Pour conclure, vous avez pu voir que le slicing permettait de réaliser bon nombre d'opérations sur les chaînes, les listes et les tuples. De premier abord un peu étrange, cette méthode est vraiment très pratique et simple à l'usage. Exercez-vous à l'utiliser et vous l'adopterez sans l'ombre d'un doute.

6 La copie de listes

Lorsque l'on copie naïvement une liste en Python, on peut s'exposer à des surprises :

```
>>> liste_a = [ 1, 2, 3 ]
>>> liste_b = liste_a
>>> liste_a
[1, 2, 3]
>>> liste_b
[1, 2, 3]
>>> liste_b[0] = 0
>>> liste_b
[0, 2, 3]
>>> liste_a
[0, 2, 3]
```

Le fait de modifier l'une des listes affecte les deux listes ! Ceci est dû au fait qu'il n'y a pas de véritable copie de liste, mais une copie de l'adresse mémoire où sont stockées les données : on a en fait défini un alias permettant de manipuler les mêmes données ! La figure 4 illustre ce phénomène.

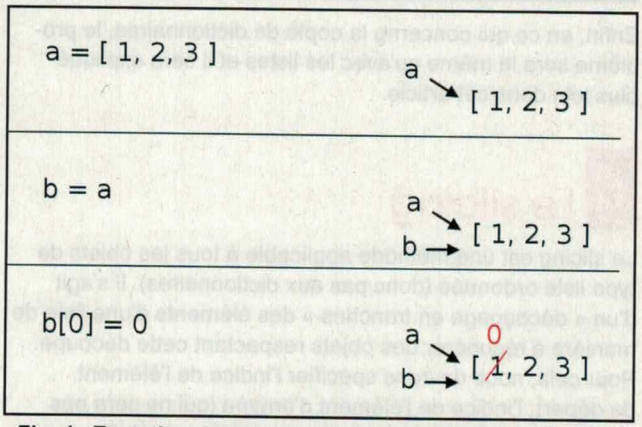


Fig. 4 : Tentative de copie de listes par affectation simple

Une première solution pour effectuer une copie peut être d'utiliser le slicing. En effet, l'opération `[:]` renvoie une nouvelle liste, ce qui résout le problème des pointeurs vers la même zone mémoire, comme le montrent la figure 5 et l'exemple suivant :

```
>>> liste_a = [ 1, 2, 3 ]
>>> liste_b = liste_a[:]
>>> liste_a
[1, 2, 3]
>>> liste_b
[1, 2, 3]
>>> liste_b[0] = 0
>>> liste_b
[0, 2, 3]
>>> liste_a
[1, 2, 3]
```

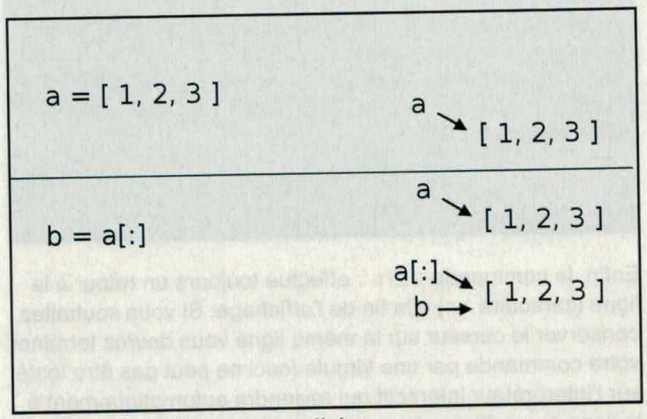


Fig. 5 : Copie de listes par slicing

Cette copie peut paraître satisfaisante... et elle l'est, mais à condition de ne manipuler que des listes de premier niveau ne comportant aucune sous-liste :

```
>>> liste_a = [ 1, 2, 3, [ 4, 5, 6 ] ]
>>> liste_b = liste_a[:]
>>> liste_a
[1, 2, 3, [4, 5, 6]]
>>> liste_b
[1, 2, 3, [4, 5, 6]]
>>> liste_b[3][0] = 0
>>> liste_b
[1, 2, 3, [0, 5, 6]]
>>> liste_a
[1, 2, 3, [0, 5, 6]]
```

En effet, le slicing n'effectue pas de copie récursive : en cas de sous-liste, on retombe dans la problématique des pointeurs mémoire (voir figure 6). La solution est alors d'utiliser un module spécifique, le module `copy`, qui permet d'effectuer une copie récursive. Bien que n'ayant pas encore approfondi la manipulation des modules, voici comment réaliser une copie de liste par cette méthode :

```
>>> from copy import *
>>> liste_a = [ 1, 2, 3, [ 4, 5, 6 ] ]
>>> liste_b = deepcopy(liste_a)
>>> liste_a
[1, 2, 3, [4, 5, 6]]
>>> liste_b
[1, 2, 3, [4, 5, 6]]
>>> liste_b[3][0] = 0
>>> liste_b
[1, 2, 3, [0, 5, 6]]
>>> liste_a
[1, 2, 3, [4, 5, 6]]
```

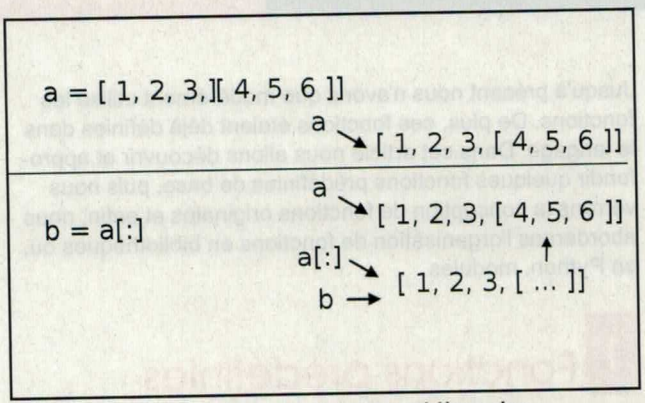


Fig. 6 : Copie de listes par slicing : problème des sous-listes

Le problème est exactement le même avec la copie de dictionnaires. Il faudra utiliser ici la méthode `copy()` :

```
>>> dico_a = {'cle_1' : 1, 'cle_2' : 2, 'cle_3' : 3 }
>>> dico_b = dico_a.copy()
>>> dico_a
{'cle_2': 2, 'cle_3': 3, 'cle_1': 1}
>>> dico_b
{'cle_2': 2, 'cle_3': 3, 'cle_1': 1}
>>> dico_b['cle_1'] = 0
>>> dico_b
{'cle_2': 2, 'cle_3': 3, 'cle_1': 0}
>>> dico_a
{'cle_2': 2, 'cle_3': 3, 'cle_1': 1}
```

Conclusion

Les structures de liste en Python sont particulièrement fournies et il existe pour chacune d'elles de nombreuses méthodes permettant de manipuler leurs éléments. Le slicing représente un raccourci efficace pour récupérer ou modifier des éléments d'une liste : il masque la complexité des opérations sous une syntaxe claire et lisible... pour peu que l'on ait pris le temps de se familiariser avec elle. ■

LES FONCTIONS ET LES MODULES

Tristan Colombo

Un code sans fonctions comporte beaucoup trop de redondances, de copier/coller. Les fonctions permettent d'écrire des programmes beaucoup plus synthétiques et modulaires. Comment écrire des fonctions en Python et comment les organiser en bibliothèques ? C'est ce que nous allons voir dans cet article.

Jusqu'à présent nous n'avons que modérément utilisé les fonctions. De plus, ces fonctions étaient déjà définies dans le langage. Dans cet article nous allons découvrir et approfondir quelques fonctions prédéfinies de base, puis nous verrons la conception de fonctions originales et enfin, nous aborderons l'organisation de fonctions en bibliothèques ou, en Python, modules.

1 Fonctions prédéfinies

La toute première fonction que nous avons utilisée est la fonction `print()` permettant d'afficher du texte à l'écran.

1.1. Afficher un texte avec print

Rappelez-vous que cette fonction est présente en Python 2.7 (sans les parenthèses) sous forme de commande et sous Python 3.2 sous forme de fonction.

1.1.1. Python 2.7 : la commande print

Plutôt que de concaténer des chaînes de caractères par l'opérateur +, la commande `print` admet le passage de plusieurs éléments à afficher, séparés par des virgules :

```
>>> print 'Linux ' + 'Pratique'
Linux Pratique
>>> print 'Linux', 'Pratique'
Linux Pratique
```

Dans le cas de l'utilisation de la virgule, les chaînes sont automatiquement séparées par un caractère espace alors qu'en utilisant la concaténation c'était au développeur de gérer ce problème.

Un autre intérêt de l'utilisation des virgules par rapport à la concaténation est le typage des données : pour effectuer une concaténation vous devez manipuler deux chaînes de caractères ! Si vous concaténez des variables non chaînées, il vous faudra les convertir avant de pouvoir les afficher. En passant les variables sous forme de paramètres à la commande `print` la conversion est implicite :

```
>>> c = 3
>>> print 'Valeur : ' + c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> print 'Valeur : ' + str(c)
Valeur : 3
>>> print 'Valeur :', c
Valeur : 3
```

Enfin, la commande `print` effectue toujours un retour à la ligne (caractère `\n`) à la fin de l'affichage. Si vous souhaitez conserver le curseur sur la même ligne vous devrez terminer votre commande par une virgule (ceci ne peut pas être testé sur l'interpréteur interactif qui reviendra automatiquement à la ligne pour afficher le prompt). La commande sera du type suivant :

```
>>> print 'Pas de retour à la ligne',
```

Cette commande est pratique mais pourrait l'être plus encore s'il y avait moyen de la configurer plus finement. C'est ce qui a été modifié et introduit dans Python 3.2 sous forme de fonction.

1.1.2. Python 3.2 : la fonction print()

La fonction `print()` se comporte globalement comme la commande `print` à la différence près qu'il faut encadrer les paramètres par des parenthèses :

```
>>> print('Linux', 'Pratique')
Linux Pratique
```

Ce qui devient beaucoup plus intéressant, c'est l'apparition des paramètres `sep` et `end` qui permettent de définir le caractère utilisé pour séparer les chaînes et de définir le caractère de fin de ligne. Ces paramètres sont utilisés d'une manière un peu particulière en spécifiant leur nom suivi du signe égal et de leur valeur lors de l'appel de la fonction.

Le mécanisme mis en jeu lors de cet appel sera analysé dans la partie consacrée aux fonctions originales. Voici quelques exemples utilisant ces paramètres qui doivent être placés en dernière position :

```
>>> print('Linux', 'Pratique', sep='**')
Linux**Pratique
 >>> print('a', 'b', 'c', 'd', sep=',')

a,b,c,d


>>> print('Linux', 'Pratique', sep='-', end='!')
Linux-Pratique!>>>
```

Après avoir vu comment afficher du texte, il peut être intéressant de voir la fonction complémentaire permettant à l'utilisateur de saisir des données.

1.2. Interagir avec l'utilisateur : la fonction input

Le dialogue entre l'utilisateur et le programme est important car il permet d'obtenir une interface permettant de paramétrer le comportement du programme (abstraction faite du passage de paramètres en lignes de commandes que nous verrons dans un autre article). Ici encore il existe une forte différence entre les fonctions proposées pour Python 2.7 et la fonction pour Python 3.2.

1.2.1. Python 2.7 : les fonctions input() et raw_input()

Les fonctions de la famille `input` permettent d'afficher un message à l'écran et de donner la main à l'utilisateur pour qu'il saisisse des caractères au clavier et qu'il appuie sur la touche [Entrée] pour valider. En Python 2.7 il existe une distinction importante entre la fonction `input()` et la fonction `raw_input()` : la première récupère un entier ou un flottant et la seconde récupère une chaîne de caractères. Avec la fonction `input()`, la saisie d'une chaîne de caractères entraînera la tentative d'interprétation de celle-ci en tant que variable :

```
>>> c = input('Veuillez saisir un entier : ')
Veuillez saisir un entier : 3
>>> c
3
>>> c = input('Veuillez saisir un entier : ')
Veuillez saisir un entier : coucou
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'coucou' is not defined
>>> a = 2
>>> c = input('Veuillez saisir un entier : ')
Veuillez saisir un entier : a
>>> c
2
```

Le comportement de la fonction `raw_input()` est plus simple puisque toute saisie sera considérée comme chaîne de caractères :

```
>>> c = raw_input('Veuillez saisir une chaine :')
Veuillez saisir une chaine :coucou
>>> c
'coucou'
>>> c = raw_input('Veuillez saisir une chaine :')
Veuillez saisir une chaine :2
>>> c
'2'
```

Pour utiliser un entier saisi sous forme de chaîne de caractères dans un contexte entier (pour une opération par exemple), il faudra le convertir avec la fonction `int()`, le plus simple étant de conserver la variable sous sa forme convertie :

```
>>> int(c) + 2
4
>>> c = int(c)
>>> c
2
```

1.2.2. Python 3.2 : la fonction input()

En Python 3.2, récupérer une saisie utilisateur sera beaucoup plus simple : il n'y a plus qu'une seule fonction `input()` qui renvoie une chaîne de caractères, libre ensuite au développeur d'effectuer la conversion nécessaire.

```
>>> c = input('Veuillez saisir un entier : ')
Veuillez saisir un entier : 2
>>> c
'2'
>>> c = input('Veuillez saisir une chaine : ')
Veuillez saisir une chaine : coucou
>>> c
'coucou'
```

Les principales fonctions de conversion sont :

- `int()` pour convertir en entier ;
- `float()` pour convertir en flottant ;
- `complex()` pour convertir en complexe ;
- `str()` pour convertir sous forme de chaîne de caractères (inutile dans le cadre d'une utilisation avec `input()`).

1.3. Rechercher de l'aide avec help()

On ne peut pas connaître par cœur la syntaxe de toutes les commandes ! Il est donc nécessaire de savoir où chercher des informations. La fonction `help()` permet d'avoir accès à une aide (minimaliste) relative à la fonction passée en paramètre. De manière générale, le nom de la fonction pour laquelle on recherche de l'aide doit être donné sans guillemets alors que les commandes doivent être

sous forme de chaînes de caractères (encadrées par des guillemets ou des apostrophes). Après affichage de l'aide, tapez sur la touche <q> pour revenir au prompt de l'éditeur interactif :

```
>>> help(input)
Help on built-in function input in module __builtin__:

input(...)
    input([prompt]) -> value

    Equivalent to eval(raw_input(prompt)).
```

Vous aurez accès à cette aide depuis le shell en utilisant la commande `pydoc` :

```
login@server:~$ pydoc input
```

2 Fonctions originales

Comme en mathématiques, une fonction effectue un calcul, un traitement, puis renvoie un résultat. Une fonction a un nom et accepte ou non des paramètres. Prenons l'exemple de la fonction mathématique f(x)=x2. En Python, cela donnerait :

```
01: def f(x):
02:   return x**2
```

En ligne 1 on définit l'en-tête de la fonction à l'aide du mot-clé `def` : la fonction s'appelle `f` et elle admet un paramètre noté `x`. La ligne se termine par le caractère deux-points : on ouvre un nouveau bloc, toutes les lignes suivantes qui seront indentées feront partie des instructions exécutées lors de l'appel à la fonction. La ligne 2 indique la valeur à renvoyer à l'aide du mot-clé `return`. Cette fonction pourra alors être appelée :

```
>>> print f(2)
4
```

Si vous omettez de préciser une valeur de retour pour une fonction, Python renverra automatiquement la valeur `None`.

Vous avez bien sûr la possibilité de créer des fonctions acceptant aucun ou de nombreux paramètre(s) :

```
>>> def hello():
...   print 'bonjour!'
...
>>> hello()
bonjour!
>>> def maFct(a, b, c):
...   print 'Parametres : ', a, ',', b, 'et', c
...
>>> maFct(1, 2, 'coucou')
Parametres :  1 , 2 et coucou
```

La documentation relative à une fonction s'écrit à l'aide d'une chaîne simple située sous l'en-tête de la fonction. Si vous avez besoin de plus d'une ligne, vous pouvez utiliser les commentaires multi-lignes qui sont encadrés par des triples guillemets. Les commentaires ajoutés ainsi à une fonction sont ensuite accessibles par la fonction `help()` :

```
>>> def ma_fonction():
...   """Fonction affichant un message
...
...      Valeur de retour : None"""
...   print "coucou"
...
>>> help(ma_fonction)
Help on function ma_fonction in module __main__:

ma_fonction()
    Fonction affichant un message

    Valeur de retour : None
```

De plus, Python dispose de mécanismes permettant de manipuler très simplement les paramètres des fonctions.

2.1. Définir des paramètres par défaut

Il est possible de définir des paramètres ayant une valeur par défaut. Si ces paramètres ne sont pas spécifiés lors de l'appel à la fonction, ce seront les valeurs par défaut qui seront utilisées. La seule contrainte est que les paramètres ayant une valeur par défaut doivent être positionnés à la fin de la liste des paramètres :

```
def ma_fonction(a, b, c=2, d=3):
  ...
```

Cette fonction pourra être appelée de trois façons différentes :

```
>>> ma_fonction(0, 1) # a=0, b=1, c=2, d=3
>>> ma_fonction(0, 1, 4) # a=0, b=1, c=4, d=3
>>> ma_fonction(0, 1, 4, 5) # a=0, b=1, c=4, d=5
```

2.2. Ordre des paramètres

En Python, l'ordre des paramètres d'une fonction n'est pas fixé si l'on est capable de les nommer. Nous avons vu un exemple de nommage de paramètres précédemment avec la fonction `print()` de Python 3.2 et les paramètres `sep` et `end`. Voici un exemple d'application de ce concept :

```
>>> def premier(a, b):
...  return a
...
>>> premier(b=5, a=2)
2
```

Bien entendu, il est encore plus recommandé qu'à l'accoutumée d'utiliser des noms de paramètres « parlants » et pas seulement des lettres comme dans l'exemple précédent.

2.3. Nombre de paramètres non fixé

Un mécanisme très utile de Python est basé sur le fait de pouvoir créer une fonction sans fixer au préalable le nombre de ses paramètres. On utilise pour cela les `*args` et les `**kwargs` :

- `*args` : tuple contenant la liste des paramètres passés par l'utilisateur ;
- `**kwargs` : dictionnaire des paramètres.

La seule différence entre les deux syntaxes (* ou ** devant le nom du paramètre) est le type de la variable de retour : un tuple dans le premier cas et un dictionnaire dans le deuxième cas (les paramètres doivent alors être saisis sous la forme `nom = valeur`). Voici un exemple d'application :

```
>>> def exemple(*args):
...   print args
...
>>> exemple(1, 2, "coucou")
(1, 2, 'coucou')
>>> def exemple_2(**kwargs):
...   print kwargs
...
>>> exemple_2(i=1, j=2, text="coucou")
{'i': 1, 'text': 'coucou', 'j': 2}
```

Il est possible de mixer les écritures de paramètres fixés et non fixés. Ceci permet d'imposer des paramètres et d'en rendre d'autres optionnels :

```
def affiche_style(text, **options):
  style = ''
  if 'color' in options:
    style += '  color : ' + options['color'] + ';\n'
  if 'font' in options:
    style += '  font : ' + options['font'] + ';\n'
  # ...
  print 'style\n{\n' + style + '}\n\n' + 'Texte : ' + text

affiche_style('Linux Pratique', color='#f00', font='Arial 12pt')
```

L'exécution de ce code produira l'affichage suivant :

```
style
{
  color : #f00;
  font : Arial 12pt;
}

Texte : Linux Pratique
```

2.4. Type de passage des paramètres : valeur ou adresse ?

Pour finir sur les fonctions, intéressons-nous à la manière dont sont transmises les variables passées en paramètres. Sont-elles passées par valeur, c'est-à-dire qu'une copie de la variable sera effectuée et que toute modification de la valeur de la variable à l'intérieur de la fonction ne sera pas conservée à la sortie de la fonction ? Ou bien sont-elles passées par adresse (ou encore par référence), c'est-à-dire que l'on travaille directement sur la variable et que toute modification de la variable dans la fonction est valable sur l'ensemble du programme ?

En Python, tous les paramètres sont passés par adresse ! Par contre, comme certains types ne sont pas modifiables, on aura alors l'impression qu'ils sont passés par valeur. Les types non modifiables sont les types simples (entiers, flottants, complexes, etc.), les chaînes de caractères et les tuples.

Voici un exemple utilisant un paramètre non modifiable :

```
>>> def incremente(a):
...  a = a + 1
...
>>> valeur = 5
>>> valeur
5
>>> incremente(valeur)
>>> valeur
5
```

Avec un paramètre modifiable, tel qu'une liste, les modifications seront visibles en sortie de la fonction :

```
>>> def ajoute(liste, val):
...   liste.append(val)
...
>>> l = []
>>> l
[]
>>> ajoute(l, 5)
>>> l
[5]
```

Ayant bien étudié la création des fonctions, voyons maintenant comment créer des ensembles cohérents de fonctions.

3 Les modules

En Python, les fonctions peuvent être organisées en bibliothèques appelées modules. On regroupe dans ces modules des fonctions gravitant autour du même centre d'intérêt. Par exemple, les fonctions mathématiques se trouvent dans le module `math`. Ce module est installé par défaut et contient par exemple les fonctions `sin()` et `cos()`.

Pour avoir accès aux fonctions contenues dans un module, il faut charger ledit module grâce à la commande `import`. Il existe plusieurs manières d'utiliser l'import :

- `from nom_module import *` : ici on demande à Python de charger en mémoire l'ensemble des fonctions (représentées par le caractère *) du module `nom_module`. Voici un exemple d'utilisation avec le module `math` :

```
>>> from math import *
>>> pi
3.141592653589793
>>> sin(pi/2)
1.0
```

On a directement accès à l'ensemble des fonctions et « constantes » du module.

- `from nom_module import fct_1, fct_2, ..., fct_n` : dans cet import, seules les fonctions `fct_1` à `fct_n` sont chargées en mémoire, toutes les autres fonctions du module `nom_module` restant inaccessibles :

```
>>> from math import pi, sin
>>> sin(pi/2)
1.0
>>> cos(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
```

- `import nom_module` : toutes les fonctions du module sont chargées mais ne seront accessibles qu'en les préfixant par le nom du module :

```
>>> import math
>>> sin(pi/2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> math.sin(math.pi/2)
1.0
```

Les imports peuvent être vus comme de gigantesques copier/coller. Imaginez donc ce qui pourrait se passer si un module nommé `A` définit une fonction `f()` qui affiche « bonjour » et si un module `B` définit une fonction de même nom qui affiche « au revoir »... Que va afficher le code suivant ?

```
>>> from A import *
>>> from B import *
>>> f()
```

Il affichera « au revoir » car en cas de noms identiques c'est le dernier import qui écrase les définitions précédentes. D'où l'intérêt de la syntaxe `import nom_module` :

```
>>> import A
>>> import B
>>> A.f()
bonjour
>>> B.f()
au revoir
```

Si vous manipulez des modules ayant un nom très long, vous pourrez définir un alias grâce au mot-clé `as`. Prenons par exemple un module nommé `monModuleAuNomTresLong` et contenant la fonction `f()`. À chaque appel à `f()` vous n'allez quand même pas écrire : `monModuleAuNomTresLong.f()`...

```
>>> import monModuleAuNomTresLong as monModule
>>> monModule.f()
```

Mais peut-être vous demandez-vous maintenant comment créer vos propres modules ?

3.1. Modules originaux

Créer un module en Python n'a rien de très compliqué... tout fichier Python peut-être appelé comme un module. Si ce fichier contient des instructions à exécution immédiate (qui ne sont pas contenues dans des fonctions), ces dernières seront exécutées au moment de l'import. Prenons l'exemple du fichier `monModule.py` suivant :

```
01: def lp():
02:   print 'Linux Pratique'
03:
04: print 'Test de la fonction lp()'
05: lp()
```

Dans un autre fichier, `monScript.py`, je pourrai faire appel à ce module en le nommant sans l'extension `.py` :

```
01: import monModule
02:
03: lp()
```

Le résultat obtenu ne sera certainement pas celui attendu : la ligne 1 va provoquer l'exécution des lignes 4 et 5 du fichier `monModule.py`, soit :

```
Test de la fonction lp()
Linux Pratique
```

Ensuite, l'appel à la fonction `lp()` de la ligne 3 effectuera un second appel à `lp()` et affichera à nouveau :

```
Linux Pratique
```

C'était le seul affichage que nous attendions. Les deux lignes précédentes étaient des lignes permettant de tester le module lors de son développement. Pour différencier le comportement de l'interpréteur lors d'une exécution directe d'un module ou lors du chargement de celui-ci, il existe un test spécifique à ajouter qui permet de déterminer une sorte de programme principal dans le module :

```
01: def lp():
02:   print 'Linux Pratique'
03:
04: if __name__ == '__main__':
05:   print 'Test de la fonction lp()'
06:   lp()
```

Ainsi, les lignes 5 et 6 seront exécutées lors d'un appel direct au script (`python monScript.py` par exemple) et seront ignorées lors d'un import (`import monScript`).

3.2. Chemin de recherche des modules

Lors d'un import, où Python va-t-il rechercher les modules ? Par défaut, il commence par rechercher dans le répertoire courant, puis dans le ou les répertoire(s) spécifié(s) par la variable d'environnement `PYTHONPATH` si celle-ci est définie, et enfin dans le répertoire des bibliothèques de Python : `/usr/lib/python<version>`.

Faites donc très attention en nommant vos fichiers : s'il porte le nom d'un module Python existant, il sera importé en lieu et place de ce dernier puisque l'import recherche d'abord les modules dans le répertoire courant.

Pour utiliser la variable `PYTHONPATH`, elle se définit comme une variable d'environnement classique (`PATH` par exemple) :

```
export PYTHONPATH=${PYTHONPATH}:/home/login/python/mes_libs
```

Pour que la modification soit permanente il faudra ajouter cette ligne à votre fichier de configuration du shell (`~/.bashrc` pour le shell bash).

3.3. Modules et sous-modules

Vous avez la possibilité d'organiser vos modules en répertoires et créer ainsi des sous-modules. Prenons l'exemple de trois modules `A1`, `A2`, et `A3` qui sont des sous-modules d'un module `A`. Au niveau architectural nous créerons un répertoire `A` contenant trois fichiers `A1.py`, `A2.py`, et `A3.py`.

Pour indiquer que A est un « répertoire-module », nous créerons un fichier supplémentaire `__init__.py`. Ce fichier spécial sera lu lors de l'exécution de l'instruction `import A` (il peut donc contenir l'équivalent de ce qu'aurait contenu le fichier `A.py`). De plus ce fichier, qui peut être vide si `A` n'est qu'une coquille permettant de classer les modules, autorisera l'accès aux sous-modules. Pour cela, il faudra décrire le chemin depuis le module `A` jusqu'à eux :

```
>>> import A.A1
```

Le module `A` est ici appelé paquetage.

Conclusion

Avec les fonctions et les modules vous pouvez commencer à développer des scripts intéressants. Nous pouvons résumer la structure générale d'un programme Python de la manière suivante :

1. Shebang et déclaration d'encodage :

```
#!/usr/bin/python
# -*- coding:utf-8 -*-
```

Jusqu'à présent nous n'avons pas parlé de l'encodage des caractères, mais vous avez dû vous rendre compte que les caractères accentués posaient problème avec Python 2.7. Cette ligne permet de spécifier l'encodage à utiliser (ici `utf-8`).

2. Les imports :

```
import module_1
...
import module_n
```

Les imports peuvent être placés n'importe où dans le code et peuvent être intégrés à des boucles conditionnelles... mais il est beaucoup plus simple de les retrouver s'ils sont placés en tête de fichier (voir la recommandation PEP8 : http://www.python.org/dev/peps/pep-0008/).

3. Les définitions de fonctions :

```
def fonction_1:
  ...
def fonction_n:
  ...
```

4. Le corps du programme principal :

```
if __name__ == '__main__':
  ...
```

Et maintenant... à vos claviers ! ■

ANALYSER DES CHAÎNES DE CARACTÈRES À L'AIDE DES EXPRESSIONS RÉGULIÈRES

Tristan Colombo

C'est une problématique que l'on retrouve fréquemment : comment analyser des chaînes de caractères ? Comment vérifier qu'elles correspondent au format attendu ? Comment récupérer des informations précises mais non connues à l'avance ? Les expressions régulières sont là pour répondre à ce problème.

Python est tout à fait adapté au traitement de données textuelles. Pour cela, il fournit deux modules : le module `string` permettant d'effectuer de nombreuses opérations simples et le module `re` prenant en charge les expressions régulières.

Dans cet article, nous verrons comment et quand utiliser ces modules et, pour ceux d'entre vous qui ne le sauraient pas, nous définirons ce qu'est une expression régulière.

1 Le module string : opérations simples

Le module `string` est installé par défaut avec Python. Il fournit de nombreuses méthodes permettant de rechercher du texte et de remplacer des chaînes de caractères. La méconnaissance de ce module conduit bien souvent à utiliser des méthodes beaucoup trop complexes et moins efficaces. Je vous propose de voir ici les fonctions les plus utiles.

1.1. Fonctions de base

La fonction `split()` permet de découper la chaîne de caractères qui lui est passée en paramètre suivant un ou des caractère(s) de séparation et renvoie une liste des chaînes découpées. Les caractères de séparation lui sont également passés en paramètre et, si ce n'est pas le cas, ce sera le caractère espace qui sera utilisé :

```
>>> import string
>>> s = 'login:password:userId:groupId:name:home:shell'
>>> string.split(s, ':')
['login', 'password', 'userId', 'groupId', 'name', 'home', 'shell']
>>> p = 'Une phrase d\'exemple'
>>> string.split(p)
['Une', 'phrase', "d'exemple"]
```

L'opération inverse s'appelle `join()`. Elle consiste à prendre une liste de chaînes de caractères pour former une chaîne en concaténant tous les éléments et en les assemblant à l'aide d'un caractère.

Cette méthode prend en paramètre une liste de chaînes de caractères et s'applique à une chaîne de caractères désignant le ou les caractère(s) de liaison :

```
>>> l = [ '1', '2', '3', '4' ]
>>> ' -> '.join(l)
'1 -> 2 -> 3 -> 4'
```

Cette méthode est disponible en standard (nous ne l'avons pas préfixée par `string` après l'import du module).

Deux autres méthodes standards peuvent être utiles : `lower()` et `upper()` permettant respectivement de convertir les caractères d'une chaîne en minuscules ou en majuscules.

Attention, bien que parlant de « conversion », ces méthodes ne modifient pas la chaîne de départ mais renvoient une nouvelle chaîne :

```
>>> s = 'Linux Pratique'
>>> s.lower()
'linux pratique'
>>> s
'Linux Pratique'
>>> s.upper()
'LINUX PRATIQUE'
```

La fonction `capitalize()` permet de ne mettre en majuscule que la première lettre d'une chaîne :

```
>>> s = 'linux pratique'
>>> string.capitalize(s)
'Linux pratique'
```

Si maintenant nous souhaitons que la première lettre de chaque mot soit une majuscule, que pouvons-nous faire ?

En utilisant les quelques fonctions vues précédemment, cela est tout à fait réalisable :

```
>>> p = 'une petite phrase de test'
>>> l = string.split(p)
>>> i = 0
>>> for s in l:
...     l[i] = string.capitalize(s)
...     i += 1
...
>>> ' '.join(l)
'Une Petite Phrase De Test'
```

On peut même le faire beaucoup plus « simplement », en utilisant la compréhension de listes :

```
>>> ' '.join([string.capitalize(s) for s in string.split(p)])
'Une Petite Phrase De Test'
```

Mais en fait il existe une fonction qui va tout faire pour nous : `capwords()`. Cette fonction effectue exactement le même traitement que ce que nous avons fait précédemment :

```
>>> string.capwords(p)
'Une Petite Phrase De Test'
```

La fonction `count()` permet de compter le nombre d'occurrences d'une sous-chaîne dans une chaîne de caractères. Le premier paramètre est la chaîne dans laquelle effectuer la recherche et le second paramètre est la sous-chaîne :

```
>>> p = 'un pingouin, deux pingouins, trois pingouins, ...'
>>> string.count(p, 'pingouin')
3
>>> string.count(p, 'pingouins')
2
```

La fonction `find()` permet de trouver l'indice de la première occurrence d'une sous-chaîne. Les paramètres sont les mêmes que pour la fonction `count()`.

En cas d'échec, `find()` renvoie la valeur `-1` (`0` correspond à l'indice du premier caractère) :

```
>>> string.find(p, 'pingouin')
3
>>> string.find(p, 'pomme')
-1
```

Enfin, la fonction `replace()` permet, comme son nom l'indique, de remplacer une sous-chaîne par une autre à l'intérieur d'une chaîne de caractères. Les paramètres sont, dans l'ordre : la chaîne de caractères à modifier, la sous-chaîne à remplacer, la sous-chaîne de remplacement, et, éventuellement, le nombre maximum d'occurrences à remplacer (si non spécifié, toutes les occurrences seront remplacées).

```
>>> p = 'un pingouin, deux pingouins, trois pingouins, ...'
>>> string.replace(p, 'pingouin', 'singe')
'un singe, deux singes, trois singes, ...'
>>> string.replace(p, 'pingouin', 'singe', 1)
'un singe, deux pingouins, trois pingouins, ...'
```

Penchons-nous maintenant sur des fonctions permettant d'effectuer des traitements plus complexes.

1.2. Fonctions « évoluées »

Il est possible de créer des tables de traduction permettant de traduire automatiquement un ensemble de caractères. Ceci est réalisable grâce à la fonction `maketrans()` qui crée la table de traduction et grâce à la fonction `translate()` qui applique les traductions sur une chaîne de caractères. Voici un exemple :

```
>>> trad = string.maketrans('abeio', '463!0')
>>> p = 'ma chaine a traduire'
>>> p.translate(trad)
'm4 ch4!n3 4 tr4du!r3'
```

Lors de la création de la table avec `maketrans()`, chaque caractère de position `i` dans la première chaîne passée en paramètre est traduit par le caractère en position `i` dans la seconde chaîne. À `a` correspond `4`, à `b` correspond `6`, etc.

Nous avons également la possibilité d'utiliser des modèles pour formater un affichage (un peu le même mécanisme qu'avec le formatage de chaînes de caractères). L'intérêt ici est que l'on peut nommer les variables dans le modèle en les préfixant du caractère `$` et que leur valeur sera donnée depuis un dictionnaire. Le code est alors beaucoup plus clair :

```
>>> t = string.Template("""
... Affichage d'une variable : $var
... Et une autre variable    : $var_2
... Et dans un texte         : ${var}iable
... Nom des variables        : $$var et $$var_2
... """)
>>> values = { 'var' : 'variable', 'var_2' : '123' }
>>> t.substitute(values)
"\nAffichage d'une variable : variable\nEt une autre variable    :
123\nEt dans un texte         : variableiable\nNom des variables
      : $var et $var_2\n"
```

Notez que l'emploi de `$$` permet d'afficher le caractère `$` et que `${nom_variable}` permet d'isoler une variable comprise dans un mot.

Si l'une des variables du modèle est absente du dictionnaire de remplacement, vous obtiendrez un message d'erreur :

```
>>> values = { 'var' : 'variable'}
>>> t.substitute(values)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.7/string.py", line 172, in substitute
    return self.pattern.sub(convert, self.template)
  File "/usr/lib/python2.7/string.py", line 162, in convert
    val = mapping[named]
KeyError: 'var_2'
```

La solution consistera alors à utiliser la méthode `safe_substitute()` qui ne provoque pas d'erreur en cas de non-substitution :

```
>>> t.safe_substitute(values)
"\nAffichage d'une variable : variable\nEt une autre variable
: $var_2\nEt dans un texte        : variableiable\nNom des
variables        : $var et $var_2\n"
```

Lorsque cela est suffisant, inutile d'aller chercher à complexifier votre code avec d'autres modules, utilisez le module `string`. En effet, les méthodes qu'il propose sont très efficaces et simples à lire.

Bien sûr, si les traitements sont plus complexes et que leur réalisation avec le module `string` implique de nombreuses opérations, il faudra passer aux expressions régulières.

2 Qu'est-ce qu'une expression régulière ?

Une expression régulière, notée « regex » ou « re » en anglais pour *regular expression*, ou encore ER en français, est une chaîne de caractères décrivant à l'aide d'une syntaxe précise un ensemble de chaînes de caractères. Cette chaîne descriptive est encore appelée « motif » ou *pattern*.

Quelle est l'utilité des expressions régulières ? Supposons que vous cherchiez à savoir si une chaîne est bien un code postal. Que devrez-vous vérifier ? Si cette chaîne est composée de cinq chiffres alors oui, c'est un code postal. Si dans la chaîne « Classement : 1 – Debian, 2 – Ubuntu, ... » vous voulez récupérer le nom du premier, vous utiliserez une expression régulière...

Voyons donc la syntaxe permettant de décrire un motif. Le tableau ci-contre indique pour chaque opérateur sa signification ainsi qu'un exemple d'application.

Si nous revenons à nos deux exemples du code postal et de l'obtention du nom de la première distribution d'un classement, il nous faut créer deux motifs :

- code postal à cinq chiffres : `\d\d\d\d\d` ou `\d{5}` ou encore `[0-9]{5}` ;
- nom de la première distribution dans la chaîne « Classement : 1 – Debian, 2 – Ubuntu, ... » : `1\s-\s(\w+)` ou `1\ -\ ([a-zA-Z]+)`.

Pour utiliser les expressions régulières en Python, il faudra importer le module `re`.

3 Le module re : manipulation des expressions régulières en Python

Pour rechercher un motif dans un texte, le module `re` propose la fonction `search()` qui prend en paramètres un motif et une chaîne de caractères dans laquelle rechercher le motif. Elle retourne un élément qui contient le motif recherché, la chaîne dans laquelle la recherche a été effectuée, et les indices de début et de fin de la sous-chaîne correspondant au motif peuvent être obtenus à l'aide des méthodes `start()` et `end()` :

```
>>> pattern = '\d{5}'
>>> text    = 'Code postal : 13013'
>>> match = re.search(pattern, text)
>>> print 'Motif', match.re.pattern, 'trouve en position :',
match.start(), ',', match.end()
Motif \d{4} trouve en position : 14 , 18
```

Si le motif n'est pas trouvé, la fonction `search()` renvoie la valeur `None` :

```
>>> text_2  = 'Pas de code postal'
>>> match_2 = re.search(pattern, text_2)
>>> if match_2 == None:
...   print 'Pas de resultat!'
...
Pas de resultat!
```

Si vous utilisez fréquemment le même motif de recherche, plutôt que de le conserver sous forme de chaîne de caractères, vous pourrez le compiler de manière à avoir un code plus efficace :

```
>>> pattern = re.compile('\d{5}')
>>> match = pattern.search(text)
```

Lorsque vous utilisez des motifs complexes, il est recommandé de les commenter et d'écrire donc une expression régulière « verbeuse ». Pour utiliser une telle expression, il faudra indiquer à la fonction `search()` que l'on travaille en mode verbeux et qu'il faut donc ignorer les espaces multiples et les commentaires Python. Ceci se fait grâce à la « constante » `VERBOSE` :

```
>>> pattern="""
... ^               # Debut de chaine
... Linux\s         # Nom commun du magazine
... (Pratique|Mag)  # Differenciation du nom
... $               # Fin de la chaine
... """
>>> match = re.search(pattern, 'Linux Pratique', re.VERBOSE)
```

Attention toutefois : la fonction `search()` ne permet de retrouver que la première sous-chaîne correspondant au motif recherché !

```
>>> text_3  = 'Plusieurs codes : 13001, 13002, 13003'
>>> match_3 = re.search(pattern, text_3)
>>> print 'Motif', match_3.re.pattern, 'trouve en position :',
match_3.start(), ',', match_3.end()
Motif \d{4} trouve en position : 18 , 22
```

Opérateur	Signification	Exemple
^	Début de ligne	^Un dans « Un matin, ... »
$	Fin de ligne	fin$ dans « C'est la fin »
a, ..., z, A, ..., Z	Un caractère précisé	Linux dans « Paru dans **Linux** Pratique »
.	Un caractère quelconque	.a dans « Et toi, **ça va** ou **pas**? »
\w	Un caractère alphanumérique (0, ...9, a, ..., z, A, ..., Z)	\win dans « **Lin**ux »
\W	Un caractère non alphanumérique (tout caractère qui n'est pas reconnu par \w)	c\W dans « **c**'est ça »
\d	Un chiffre (0, ... 9)	\d\d dans « Le résultat est **10** »
\D	Un caractère qui n'est pas un chiffre (tout caractère qui n'est pas reconnu par \d)	\Dinux dans « **Linux** Pratique »
\s	Un caractère d'espacement (espace, \t, \n, \r, ou \f)	le\spetit dans « **le petit** pingouin »
\S	Tout caractère autre qu'un caractère d'espacement (tout caractère qui n'est pas reconnu par \s)	\Sinux dans « **Linux** Pratique »
\	Un caractère servant d'opérateur auquel on retire sa fonction (caractère d'échappement)	fini\. dans « C'est **fini.** »
+	Répétition du caractère précédent au moins une fois (entre 1 et n fois)	a+ dans « **a** ou **aa**b ou **aaaa** »
*	Répétition éventuelle du caractère précédent (entre 0 et n fois)	a*b dans « **b** ou **aaab** »
?	Apparition éventuelle du caractère précédent (entre 0 et 1 fois)	a?b dans « **b** ou **ab** ou aaab »
{m}	Répétition exactement m fois du caractère précédent	a{4} dans « aa ou **aaaa** »
{m,}	Répétition au moins m fois du caractère précédent	a{2,} dans « a ou **aa** ou **aaa** »
{, n}	Répétition entre 0 et n fois du caractère précédent	a{,2} dans « **a** ou **aa** ou aaa »
{m, n}	Répétition entre m et n fois du caractère précédent	a{2, 3} dans « a ou **aa** ou **aaa** ou aaaa »
[]	Délimite un ensemble de caractères. Soit ils sont tous énumérés, soit ils sont déterminés à l'aide d'intervalles (ex : [a-z] pour tous les caractères minuscules).	[A-Z][0-9] dans « le code **C5** »
[^]	Délimite un ensemble de caractères indésirables (équivalent du not booléen).	[^a-z][a-z] dans « **Majuscule** »
()	Délimite un ensemble de caractères qui seront capturés pour une utilisation postérieure	(linux) dans « distribution **linux** »
|	Spécifie une alternative : soit ce qui précède, soit ce qui suit. Cet opérateur est utilisé en général dans ()	(chaine|chaîne) dans « Écrit-on une **chaine** ou une **chaîne** ? »

3.1. Une fonction pour les retrouver tous

Si vous souhaitez retrouver toutes les occurrences d'une chaîne correspondant à un motif, c'est la fonction `findall()` qu'il faudra utiliser. Elle fonctionne de la même manière que `search()` mais renvoie une liste de chaînes de caractères correspondant au motif recherché :

```
>>> pattern = "\d{5}"
>>> text    = "Plusieurs codes : 13001, 13002, 13003"
>>> for m in re.findall(pattern, text):
...   print "Code :", m
...
Code : 13001
Code : 13002
Code : 13003
```

Si vous souhaitez obtenir un élément vous fournissant les mêmes informations que `search()`, c'est la fonction `finditer()` qu'il faudra utiliser :

```
>>> for m in re.finditer(pattern, text):
...   print 'Motif', m.re.pattern, 'trouve en position :',
m.start(), ',', m.end()
...
Motif \d{5} trouve en position : 18 , 23
Motif \d{5} trouve en position : 25 , 30
Motif \d{5} trouve en position : 32 , 37
```

3.2. Faire référence à une chaîne trouvée

Les expressions régulières permettent de faire référence à un élément noté entre parenthèses et donc capturé. Ceci peut être effectué directement à l'intérieur d'une expression régulière en désignant la capture par un anti-slash suivi du numéro de capture (commence à 1). Voici un exemple :

```
>>> pattern = r'^Nom : ([A-Z][a-z]+) Prenom : ([A-Z][a-z]+) Mail :
\2\.\1@linux.org$'
>>> match = re.search(pattern, 'Nom : Torvald Prenom : Linus Mail :
Linus.Torvald@linux.org')
```

Nous avons utilisé ici une chaîne de caractères sous la forme `r'...'`. Ceci permet de déterminer une chaîne brute (*raw string*) qui interprétera différemment les anti-slashes (ils ne protègent aucun caractère et doivent donc être considérés en tant que caractères à part entière).

Pour une meilleure lisibilité, il est possible de nommer les éléments capturés à l'aide de la syntaxe suivante :

- `(?P<nom>motif)` pour nommer un élément ;
- `(?P=nom)` pour utiliser un élément.

```
>>> pattern = r'^Nom : (?P<name>[A-Z][a-z]+) Prenom :
(?P<firstname>[A-Z][a-z]+) Mail : (?P=firstname).(?P=name)@linux.org$'
>>> match = re.search(pattern, 'Nom : Torvald Prenom : Linus Mail :
Linus.Torvald@linux.org')
```

On peut également récupérer la valeur capturée après un appel à une fonction de la famille `search()`. Les éléments sont récupérés dans une liste à l'aide de la fonction `groups()` et peuvent être obtenus un à un en utilisant la fonction `group()` qui prend en paramètre le numéro de l'élément (commence à `1`, et pour l'indice `0` la fonction renvoie toute la chaîne).

On peut également les obtenir dans un dictionnaire à l'aide de la fonction `groupdict()` (si vous avez nommé les éléments capturés, c'est cette méthode qu'il faudra choisir).

Voyons tout d'abord un exemple d'application avec les fonctions `groups()` et `group()` :

```
>>> pattern = '^Nom : ([A-Z][a-z]+) Prenom : ([A-Z][a-z]+)'
>>> import re
>>> match = re.search(pattern, 'Nom : Torvald Prenom : Linus$')
>>> match.groups()
('Torvald', 'Linus')
>>> match.group(0)
'Nom : Torvald Prenom : Linus'
>>> match.group(1)
'Torvald'
>>> match.group(2)
'Linus'
```

Avec des éléments nommés, on utilise `groupdict()` :

```
>>> pattern = '^Nom : (?P<name>[A-Z][a-z]+) Prenom :
(?P<firstname>[A-Z][a-z]+)$'
>>> match = re.search(pattern, 'Nom : Torvald Prenom : Linus')
>>> match.groupdict()
{'name': 'Torvald', 'firstname': 'Linus'}
>>> m = match.groupdict()
>>> m['name']
'Torvald'
>>> m['firstname']
'Linus'
```

3.3. Les options de recherche

Des options de recherche peuvent être spécifiées lors de l'appel aux fonctions du module `re` ou directement dans les motifs. Elles vont permettre de modifier la manière d'effectuer la recherche.

3.3.1. Options dans les fonctions de recherche

Toutes les fonctions de recherche acceptant un motif en paramètre permettent de paramétrer la recherche à l'aide d'options. Ces options sont indiquées à l'aide de « constantes » telles que `VERBOSE` vues précédemment. Il existe également :

- `IGNORECASE` pour effectuer une recherche non sensible à la casse (pas de différenciation entre les majuscules et les minuscules).
- `MULTILINE` indique que la chaîne sur laquelle la recherche va être effectuée contient plusieurs lignes séparées par un retour à la ligne (caractère `\n`).

Avec cette option, les opérateurs d'expressions régulières ^ et $ détectent les lignes terminées par un retour à la ligne.

- DOTALL indique que la chaîne sur laquelle la recherche va être effectuée contient plusieurs lignes séparées par un point (fonctionnement identique à MULTILINE mais avec le caractère . au lieu de \n).
- UNICODE permet d'utiliser l'encodage Unicode pour la recherche (utile avec les caractères accentués). Notez que vos chaînes devront être données au format Unicode en les préfixant par u : u'chaîne Unicode'. De plus, Python 3.2 utilisant l'Unicode nativement, cette option sera inutile avec cette version de Python.

Voici un exemple d'application de ces options :

```
>>> pattern = '^linux pratique$'
>>> match = re.search(pattern, 'Linux Pratique', re.IGNORECASE)
>>> match.start()
0
>>> pattern = '^linux$'
>>> match = re.search(pattern, 'linux\npratique', re.MULTILINE)
>>> match.start()
0
```

La combinaison de plusieurs options se fait à l'aide du caractère | :

```
>>> pattern = '^linux$'
>>> match = re.search(pattern, 'LiNuX\npratique', re.MULTILINE |
re.IGNORECASE)
>>> match.start()
0
```

3.3.2. Options dans les motifs

Les options précédentes peuvent être indiquées directement dans les motifs sous la forme (?option) où option correspond à l'une des lettres suivantes :

- i pour IGNORECASE ;
- m pour MULTILINE ;
- s pour DOTALL ;
- u pour UNICODE ;
- x pour VERBOSE.

La combinaison de plusieurs options se fait par juxtaposition d'options :

```
>>> pattern = '(?i)^linux pratique$'
>>> match = re.search(pattern, 'Linux Pratique')
>>> match.start()
0
>>> pattern = '(?m)^linux$'
>>> match = re.search(pattern, 'linux\npratique', re.MULTILINE)
>>> match.start()
0
>>> pattern = '(?i)(?m)^linux$'
>>> match = re.search(pattern, 'LiNuX\npratique')
>>> match.start()
0
```

3.4. Modifier une chaîne de caractères

La substitution se fait à l'aide de la méthode sub() et suit les mêmes règles que les fonctions précédentes avec ajout du paramètre indiquant la chaîne à insérer à la place de l'expression régulière détectée. Cette fonction retourne la chaîne de caractères modifiée :

```
>>> pattern = '(mag|Mag)$'
>>> re.sub(pattern, 'Pratique', 'Linux Mag')
'Linux Pratique'
```

Les références à des éléments capturés sont autorisées :

```
>>> pattern = '(mag|Mag)$'
>>> re.sub(pattern, r'\1azine', 'Linux Mag')
'Linux Magazine'
```

Pour des éléments capturés nommés, la référence se fera par \g<nom> :

```
>>> pattern = '(?P<name>mag|Mag)$'
>>> re.sub(pattern, r'\g<name>azine', 'Linux Mag')
'Linux Magazine'
```

Notez que sub() effectue les substitutions pour toutes les occurrences du motif rencontrées.

Pour limiter les substitutions aux n premières chaînes correspondant au motif, il faudra utiliser la fonction subn() qui accepte les mêmes paramètres que sub() plus un paramètre count indiquant le nombre de substitutions :

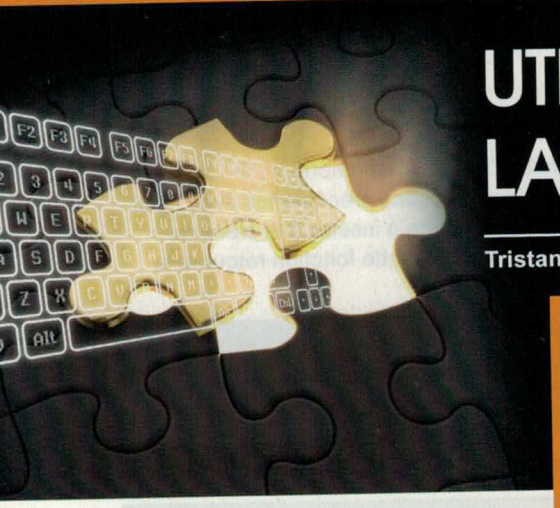
```
>>> pattern = '(mag|Mag)'
>>> re.subn(pattern, r'\1azine', 'Linux Mag Linux Mag Linux
Mag', count=2)
('Linux Magazine Linux Magazine Linux Mag', 2)
```

Pour finir, signalons l'existence d'une fonction split() où les caractères de séparation sont déterminés par un motif :

```
>>> pattern = '-\*-|--|\*'
>>> re.split(pattern, '1-*-2--3--4*5*6-*-7')
['1', '2', '3', '4', '5', '6', '7']
```

Conclusion

Nous avons pu voir de manière très large comment manipuler des chaînes de caractères en Python avec les modules string et re ainsi que la puissance des expressions régulières. Mais que cette puissance ne vous fasse pas oublier le zen de Python : « Préfère : (...) le simple au complexe et le complexe au compliqué, (...) ». Donc si vous pouvez résoudre un problème à l'aide du module string, inutile d'utiliser les expressions régulières ! ■



UTILISER LES ARGUMENTS DE LA LIGNE DE COMMANDES

Tristan Colombo

Dans les systèmes Linux, il est d'usage de pouvoir utiliser une application en mode commande, même si cette dernière dispose d'une interface graphique. Ce mécanisme permet aux utilisateurs confirmés de s'affranchir de la lourdeur inhérente aux interfaces graphiques (déplacements et clics souris répétés) au profit d'une utilisation certes moins élégante visuellement, mais plus pratique et rapide. Parfois l'interface graphique ne représente aucun intérêt et ne sera pas développée. Dans tous les cas, il va falloir être capable de lire les arguments de la ligne de commandes.

Comme tout langage, Python vous permet de récupérer et d'utiliser des arguments passés à un script en ligne de commandes. Les arguments, ce sont ces paramètres, parfois optionnels, que vous ajoutez à une commande. Par exemple, dans la commande `rm fichier1 fichier2`, `fichier1` est le premier argument et `fichier2` est le second argument.

En Python, la façon la plus simple de travailler en prenant en compte la saisie d'arguments sur la ligne de commandes est d'utiliser le module `sys`. En effet, ce dernier propose une variable, `argv`, qui contient l'ensemble des paramètres de l'appel au script sous la forme d'une liste. À l'instar des scripts shell, le premier élément de cette liste sera le nom de votre script (équivalent au `$0` du shell désignant le nom de la commande), le deuxième élément (dont l'index sera 1) correspondra au premier paramètre, etc. Voici un exemple d'application avec un script que nous appellerons `display_args.py` :

```
01: import sys
02:
03: print 'Nombre d\'arguments saisis :', str(len(sys.argv) - 1)
04: print 'Nom du script :', sys.argv[0]
05: print 'Paramètres saisis :'
06: for i in range(1, len(sys.argv)):
07:   print 'sys.argv[' + str(i) + '] = ' + sys.argv[i]
```

En appelant ce script depuis un terminal, ce dernier affichera alors la liste des paramètres qui lui ont été transmis :

```
login@server:~$ python display_args.py 'Linux Pratique' 10 3.14
Nombre d'arguments saisis : 3
Nom du script : display_args.py
Paramètres saisis :
sys.argv[1] = Linux Pratique
sys.argv[2] = 10
sys.argv[3] = 3.14
```

Bien sûr, en appelant le script de cette manière la numérotation des arguments peut prêter à confusion. Vous pouvez alors ajouter à votre script une première ligne indiquant au shell quel interpréteur il doit utiliser lorsqu'il doit exécuter ce fichier et rendre le fichier exécutable (et éventuellement, le renommer sans l'extension `.py`). Cette première ligne s'appelle le « shebang » et pour connaître le chemin menant à votre interpréteur Python, vous pouvez utiliser la commande `which`. Par exemple, si l'on modifie le code précédent pour le rendre directement exécutable :

1. On recherche le chemin absolu menant à notre interpréteur Python :

```
login@server:~$ which python
/usr/bin/python
```

2. On ajoute le shebang en première ligne du script `display_args.py` :

```
01: # !/usr/bin/python
02:
03: print 'Nombre d\'arguments saisis :', len(sys.argv) - 1
04: …
```

3. On rend le fichier exécutable :

```
login@server:~$ chmod ugo+x display_args.py
```

4. Éventuellement on le renomme (considération purement esthétique dans la mesure où l'on considère que le script est en version stable) :

```
login@server:~$ mv display_args.py display_args
```

5. On peut lancer directement la commande :

```
login@server:~$ display_args 'Linux Pratique' 10 3.14
```

Tout cela est bien pratique, mais si l'on souhaite utiliser des options avec un format un peu plus complexe ? Si l'on prend l'exemple de la commande shell `cut` permettant de découper un texte en fonction d'un caractère et de sélectionner des colonnes précises résultant de ce découpage, on peut

lui passer en paramètre le caractère servant à déterminer les colonnes. Il s'agit de l'option `-d` suivie du délimiteur. Mais cette option peut également être appelée dans sa version longue par `--delimiter=` suivie du délimiteur.

Gérer manuellement toutes les options d'un script serait alors très fastidieux... Heureusement, comme bien souvent avec Python, il y a un module pour ça ! Il s'agit du module `argparse` que nous allons étudier dans cet article.

1 Première utilisation du module argparse

Tout d'abord attention : le module `argparse` est la nouvelle version du module `optparse` qui est maintenant déprécié. La page de documentation de http://docs.python.org l'indique clairement. Les deux modules sont toujours disponibles, car `argparse` inclut de nouvelles fonctionnalités qui étaient incompatibles avec l'architecture de l'ancien module `optparse`. Ainsi, bien que le nom des objets varie de l'un à l'autre et que le nom des modules soit différent, `argparse` est bien la nouvelle version de `optparse` que vous pouvez continuer à utiliser, mais qui ne sera plus amélioré.

Le fonctionnement général est très simple et peut se résumer en cinq étapes :

1. Évidemment, importer le module `argparse` pour pouvoir l'utiliser :

```
import argparse
```

2. Créer un objet `ArgumentParser` qui va contenir la liste des arguments. Le paramètre `description` permet d'indiquer par un petit commentaire ce que fait votre script et `epilog` est le texte qui sera affiché à la fin de l'aide. Ces textes seront utilisés lors de l'affichage de l'aide générée automatiquement.

```
parser = argparse.ArgumentParser(description='xxxx',
epilog='yyyy')
```

3. Ajouter des arguments et définir à l'aide de paramètres spéciaux s'il s'agit de simples drapeaux indiquant si une option a été activée ou pas, s'il s'agit d'une ou de plusieurs valeurs à stocker, etc. Nous étudierons plus en détails ces paramètres par la suite. En guise d'exemple, définissons une option `-a` qui ne collecte aucune information et indiquera seulement si l'utilisateur l'a utilisée lors de l'appel à la commande (`True`) ou non (`False`).

```
parser.add_argument('-a', action='store_true', default=False,
help='Activation du drapeau a')
```

4. Récupérer les arguments saisis par l'utilisateur :

```
args = parser.parse_args()
```

5. Utiliser les arguments :

```
# Affichage de la liste des arguments
print args
# Valeur de l'argument -a
print args.a
```

Le script que nous venons de créer dispose des options `-a` et `-h` pour l'aide qui est générée automatiquement. Supposons que le fichier se nomme `command.py`, en lançant la commande `python command.py -h`, nous obtiendrons :

```
usage: command.py [-h] [-a]

Description de ma commande

optional arguments:
  -h, --help  show this help message and exit
  -a          Activation du drapeau a
```

Comme vous pouvez le voir, la mise en place de ce module est vraiment très simple et permet une gestion automatique de l'aide et de la lecture des options. Seule la troisième étape consistant à définir les arguments mérite d'être approfondie.

2 Définition des arguments

Les arguments sont définis à l'aide de la méthode, bien nommée, `add_argument`. Cette méthode admet de nombreux paramètres dont voici la signification accompagnée d'exemples.

2.1. Le nom d'un argument

La définition du nom de l'argument s'effectue dans le ou les premiers paramètres de type chaîne de caractères fournis à la méthode. Pourquoi a-t-on la possibilité de définir plusieurs noms ? Tout simplement pour pouvoir effectuer une même action à l'aide d'alias ou d'arguments spécifiés en écriture raccourcie ou longue. Pour obtenir la version d'une commande avec le paramètre `-v` ou le paramètre `--version` :

```
parser.add_argument('-v', '--version', action='store_true',
help='Afichage de la version de la commande')
```

On pourra alors appeler indifféremment `python command.py -v` et `python command.py --version`.

Attention, lors de la récupération des arguments avec `parser.parse_args()`, le nom d'un argument comportant plusieurs noms est celui de la première occurrence d'un nom commençant par `--`. Par exemple :

```
args = parser.parse_args()
```

`args.v` n'existe pas, par contre `args.version` existe et si l'on ajoute d'autres entrées, ce sera toujours `args.version`. Pour remédier à ce problème, le paramètre `dest` indique le nom qui permettra de retrouver le paramètre :

```
parser.add_argument('-v', '--version', dest='VERSION',
action='store_true', default=False, help='Affichage de la
version de la commande')
```

L'accès à la valeur du paramètre se fera alors par `args.VERSION`.

Enfin, vous n'êtes pas obligé de spécifier vos options en les préfixant par le caractère `-`. Pour indiquer d'autres caractères, il faudra le signaler lors de la création de l'instance d'`ArgumentParser` à l'aide du paramètre `prefix_chars` :

```
parser = argparse.ArgumentParser(description='xxxx', prefix_
chars='-+')
```

Ceci peut être utile pour activer/désactiver une option : `python command.py -active` ou `python command +active`.

2.2. Les actions possibles

Plusieurs actions sont possibles lors du traitement d'un argument (paramètre `action=...`) :

- `store` : action par défaut consistant à sauvegarder la valeur.

```
parser.add_argument('-v', action='store', help='Valeur initiale')
args = parser.parse_args()
```

`args.v` contient la valeur saisie après l'argument `-v`.

Le script pourra être indifféremment appelé par l'une des deux commandes ci-dessous :

```
python command -v123
python command -v=123
```

- `store_const` : sauvegarde une valeur déterminée au préalable par le développeur à l'aide du paramètre `const`. Il s'agit en fait de définir un drapeau qui va initialiser une valeur : l'utilisateur ne saisit aucune valeur.

```
parser.add_argument('-v', action='store_const', const='Ma
Valeur', help='Drapeau paramétrable')
args = parser.parse_args()
```

La variable `args.v` contiendra la valeur `'Ma Valeur'` si l'argument `-v` est spécifié lors de l'appel de la commande et, sinon, elle contiendra `None`.

- `store_true`/`store_false` : sauvegarde la valeur booléenne `True` ou `False` si l'argument est indiqué. Le fonctionnement est identique à `store_const` mais avec des booléens.

Avec un appel à `python command -v`, la variable `args.v` vaudra `True` (et `False` si l'on utilise `store_false`). En l'absence de l'argument, la variable `args.v` prend la valeur booléenne inverse de celle indiquée par l'action `store`.

- `append` : enregistre la valeur dans une liste. Si plusieurs appels au même argument sont spécifiés, alors elles sont ajoutées à la liste.

```
parser.add_argument('-l', action='append', help='Liste de
valeurs')
args = parser.parse_args()
```

Voici des exemples d'appels et le contenu de la variable `args.l` :

```
login@server:~$ python command
None
login@server:~$ python command -l=12
['12']
login@server:~$ python command -l=12 -l=22 -l=5
['12', '22', '5']
```

- `append_const` : enregistre une valeur spécifique dans une liste. Le fonctionnement est identique à celui de l'action `append` où chaque valeur serait sauvegardée par l'action `store_const` :

```
parser.add_argument('-l', action='append_const', const='Ma
Valeur', help='Liste de valeurs')
args = parser.parse_args()
```

Voici des exemples d'appels et le contenu de la variable `args.l` :

```
login@server:~$ python command
None
login@server:~$ python command -l
['Ma Valeur']
login@server:~$ python command -l -l -l
['Ma Valeur', 'Ma Valeur', 'Ma Valeur']
```

- `version` : action spéciale indiquant la version du programme. Le message contenant le numéro de version est contenu dans le paramètre `version` :

```
parser.add_argument('-v', action='version', version='%(prog)s
v1.0.0 Licence GPLv3', help='Version')
args = parser.parse_args()
```

Notez que `%(prog)s` fait référence au nom du script s'exécutant, tout comme `sys.argv[0]`. L'appel de la commande avec l'option `-v` provoquera alors l'affichage de la ligne :

```
command v1.0.0 Licence GPL
```

2.3. Les valeurs

Il est possible de paramétrer plus finement le comportement du programme avec les valeurs récupérées dans la ligne de commandes. Tout d'abord le paramètre `type` permet de spécifier le type de valeur attendu : `int`, `float`, `bool`, etc., et `file` pour les fichiers. Ceci permettra non seulement de convertir le type de donnée stockée dans le type attendu mais en plus, une vérification du type sera effectuée lors de l'appel à la commande (et pour les fichiers un objet-fichier sera créé) :

```
parser.add_argument('-a', action='store', type=int, help='Valeur
initiale')
args = parser.parse_args()
```

L'affichage de la valeur de `args.a` donne :

```
login@server:~$ python command -a=12
12
login@server:~$ python command -a=linux
usage: command [-h] [-a A]
command: error: argument -a: invalid int value: 'linux'
```

Lors de l'appel erroné, `argparse` affiche automatiquement la syntaxe attendue et un message d'erreur. Dans l'affichage de la syntaxe, on peut voir `[-a A]` qui signifie que l'argument `-a` attend un paramètre (nommé ici `A`). Il est possible de modifier cet affichage en utilisant le paramètre `metavar` :

```
parser.add_argument('-a', action='store', type=int, metavar='N',
help='Valeur initiale')
args = parser.parse_args()
```

Lors de la saisie d'arguments invalides, on obtient alors :

```
usage: command [-h] [-a N]
command: error: argument -a: invalid int value: 'linux'
```

Il est également possible de définir une valeur par défaut pour un argument à l'aide du paramètre `default` :

```
parser.add_argument('-a', action='store', type=int, default=10,
help='Valeur initiale')
args = parser.parse_args()
```

Ainsi, un appel à la commande sans spécifier d'argument `-a` initialisera tout de même la variable `args.a` avec la valeur `10`.

Quand un argument est obligatoire, il faudra utiliser `required=True` : en cas d'omission de l'argument une erreur sera signalée :

```
parser.add_argument('-a', action='store', required=True)
```

```
login@server:~$ python command
usage: command [-h] -a A
command: error: argument -a is required
```

Enfin, `argparse` offre la possibilité de préciser le nombre de valeurs attendues pour chaque argument grâce au paramètre `nargs`. Ce paramètre accepte les valeurs suivantes :

- un entier `n` indiquant le nombre fixe de valeurs attendues.

```
parser.add_argument('-a', action='store', nargs=3)
```

```
login@server:~$ python command -a 2
usage: command [-h] [-a A A A]
command: error: argument -a: expected 3 argument(s)
login@server:~$ python command -a  1 2 3
args.a = ['1', '2', '3']
```

- le caractère `?` indiquant la présence éventuelle d'une valeur (entre 0 et 1).

```
parser.add_argument('-b', action='store', nargs='?')
```

```
login@server:~$ python command -b
args.b = None
login@server:~$ python command -b 1
args.b = '1'
login@server:~$ python command -b 1 2
usage: command [-h] [-b [B]]
command: error: unrecognized arguments: 3
```

- le caractère `*` indiquant la présence éventuelle de valeurs (entre 0 et n).

```
parser.add_argument('-c', action='store', nargs='*')
```

```
login@server:~$ python command -c
args.c = []
login@server:~$ python command -c 1
args.c = ['1']
login@server:~$ python command -c 1 2
args.c = ['1', '2']
login@server:~$ python command -c 1 2 3 ...
args.c = ['1', '2', '3', ...]
```

- le caractère `+` indiquant la présence d'au moins une valeur (entre 1 et n).

```
parser.add_argument('-c', action='store', nargs='+')
```

```
login@server:~$ python command -d
usage: command.py [-h] [-d D [D ...]]
command.py: error: argument -d: expected at least one argument
login@server:~$ python command -d 1
args.d = ['1']
login@server:~$ python command -d 1 2 ...
args.d = ['1', '2', ...]
```

2.4. Options conflictuelles

Que se passe-t-il si l'on définit les mêmes options pour des actions différentes ?

```
parser.add_argument('--long_a', '-a', action='store')
parser.add_argument('-a', action='store')
```

```
login@server:~$ python command
argparse.ArgumentError: argument -a: conflicting option
string(s): -a
```

Le problème peut être résolu en appliquant le paramètre `conflict_handler='resolve'` au constructeur `ArgumentParser` :

```
parser = argparse.ArgumentParser(description='xxxx', conflict_
handler='resolve')
```

```
login@server:~$ python command -h
usage: command.py [-h] [--long_a LONG_A] [-a A]

Description de ma commande

optional arguments:
  -h, --help       show this help message and exit
  --long_a LONG_A
  -a A
```

La deuxième option `-a` est venue écraser la première option qui était associée à l'option `--long_a`.

2.5. Regroupement d'options

Pour une meilleure lecture de l'aide, il est possible de regrouper les arguments en grandes catégories. Lors de l'affichage de l'aide, les arguments disponibles seront alors affichés par groupe. La création d'un groupe se fait par la méthode `add_argument_group()` :

```
group = parser.add_argument_group('login')
group.add_argument('-u', action='store', metavar='name',
help='Nom de l\'utilisateur')
group.add_argument('-p', action='store', metavar='password',
help='Mot de passe')

group2 = parser.add_argument_group('io')
group2.add_argument('-i', action='store', metavar='filename',
help='Fichier en entrée')
group2.add_argument('-o', action='store', metavar='filename',
help='Fichier en sortie')
```

Lors de l'affichage de l'aide par `python command -h` nous obtiendrons alors l'affichage suivant :

```
usage: command.py [-h] [-u name] [-p password] [-i filename] [-o filename]

Description de ma commande

optional arguments:
  -h, --help   show this help message and exit

login:
  -u name      Nom de l'utilisateur
  -p password  Mot de passe

io:
  -i filename  Fichier en entrée
  -o filename  Fichier en sortie
```

Lorsqu'aucun groupe n'est indiqué lors de l'ajout d'un argument, c'est le groupe par défaut « optional arguments » qui est utilisé pour les options (précédées par un préfixe - ou définies par le développeur) et le groupe « positional arguments » est utilisé pour les autres.

Les groupes permettent aussi de définir des ensembles d'options « incompatibles » : une option et une seule pourra être sélectionnée dans le groupe. Cette opération est réalisée en créant un groupe par `add_mutually_exclusive_group()` :

```
group = parser.add_mutually_exclusive_group('login')
group.add_argument('-r', action='store', metavar='left',
help='Déplacement vers la gauche en px')
group.add_argument('-l', action='store', metavar='right',
help='Déplacement vers la droite en px')
```

Lors de l'affichage de l'aide de la commande, les ensembles d'options « exclusives » seront notés `[ option1 | option2 | ... ]`.

```
usage: command.py [-h] [-r left | -l right]

Description de ma commande

optional arguments:
  -h, --help  show this help message and exit
  -r left     Déplacement vers la gauche en px
  -l right    Déplacement vers la droite en px
```

3 Un exemple récapitulatif

Pour finir cet article voici un exemple récapitulatif permettant de mettre en place les différents types d'arguments vus précédemment.

```
01: #!/usr/bin/pytho,
02: # -*- coding:utf-8 -*-
03:
04: import argparse
05:
06: parser = argparse.ArgumentParser(description='Commande
affichant ses paramètres',
07:         prefix_chars='+-')
08:
09: parser.add_argument('-v', '--version', action='version',
10:         version='%(prog)s v1.0.0 Licence GPL',
11:         help='Affichage de la version de la commande')
12:
13: """
14:    Arguments positionnels
15: """
16: parser.add_argument('filename', action='store', type=file,
nargs='+',
17:         help='Fichier(s) à traiter')
18: parser.add_argument('-algo', action='store', metavar='algo_
name', nargs=1,
19:         required=True, help='Nom de l\'algorithme à
appliquer')
20: parser.add_argument('-s', action='append', metavar='N',
type=int, dest='sum',
21:         help='Données utilisées pour initialiser
l\'algorithme')
22:
23: """
24:    Arguments du groupe format
25: """
26: group_format = parser.add_argument_group('formatage')
27: subgroup_format = group_format.add_mutually_exclusive_
group()
28: subgroup_format.add_argument('-png', action='store_true',
29:         help='Document au format image png')
30: subgroup_format.add_argument('-pdf', action='store_true',
31:         help='Document au format pdf')
32:
33: """
34:    Arguments du groupe io
35: """
36: group_io = parser.add_argument_group('entrées/sorties')
37: group_io.add_argument('-tmp', action='store', default='/tmp',
dest='tmp_dir',
38:         help='Répertoire temporaire')
39: subgroup_verbose = group_io.add_mutually_exclusive_group()
40: subgroup_verbose.add_argument('+verb', action='store_true',
41:         help='Active le mode verbeux')
42: subgroup_verbose.add_argument('-verb', action='store_true',
43:         help='Désactive le mode verbeux')
44:
45: args = parser.parse_args()
46: print args
```

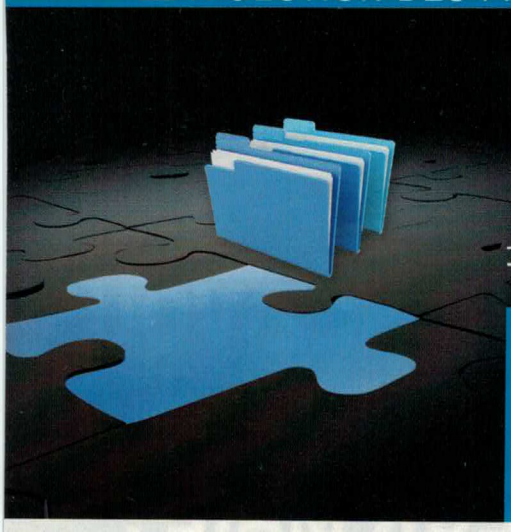
Les arguments ont été ici scindés en trois groupes : arguments positionnels (lignes 13 à 21), arguments du groupe format (lignes 23 à 31) et arguments du groupe io (lignes 33 à 43). L'argument `-v` permettant d'afficher la version de la commande a été isolé dans le code dans les lignes 9 à 11, car il ne requiert aucun traitement. Les arguments qui ont été définis dans ce code produisent l'affichage de l'aide suivante, nous indiquant plus précisément la syntaxe d'utilisation du script :

```
usage: command.py [-h] [-v] -algo algo_name [-s N] [-png | -pdf]
                  [-tmp TMP_DIR] [+verb | -verb]
                  filename [filename ...]

Commande affichant ses paramètres

positional arguments:
  filename         Fichier(s) à traiter

optional arguments:
  -h, --help       show this help message and exit
  -v, --version    Affichage de la version de la commande
  -algo algo_name  Nom de l'algorithme à appliquer
  -s N             Données utilisées pour initialiser
l'algorithme

formatage:
  -png             Document au format image png
  -pdf             Document au format pdf

entrées/sorties:
  -tmp TMP_DIR     Répertoire temporaire
  +verb            Active le mode verbeux
  -verb            Désactive le mode verbeux
```

Ainsi, on voit bien que `+verb`/`-verb` (lignes 40 à 43) et `-png`/`-pdf` (lignes 28 à 31) sont des drapeaux appartenant à deux ensembles d'options incompatibles ([+verb | -verb] et [-png | -pdf]), que l'argument `-algo` est obligatoire, etc.

Conclusion

Les différents usages vus dans cet article sont les plus courants. Le module `argparse`, dans une utilisation plus avancée, permet de définir des actions spécifiques en créant des classes héritant de la classe `argparse.Action`. Si le sujet vous intéresse, je vous conseille de consulter la documentation sur http://docs.python.org/dev/library/argparse.html. Une excellente référence aussi pour tout ce qui touche aux modules standards de Python : le livre de Doug Hellmann « The Python Standard Library by Example » aux éditions Addison Wesley. Attention toutefois, 1300 pages en anglais... âmes sensibles s'abstenir ! ■



MANIPULATION DE FICHIERS EN PYTHON

Tristan Colombo

Les fichiers représentent un élément essentiel du développement informatique, permettant de conserver et retrouver des informations. Tout langage permet d'accéder à des fichiers et Python ne déroge pas à la règle...

Que l'on souhaite stocker des données de manière structurée ou bien exploiter des données issues d'un autre programme, nous aurons bien souvent recours aux fichiers. Avant de se lancer dans la technique proprement dite et de voir comment lire ou écrire un fichier, nous nous attarderons sur une notion fondamentale bien souvent oubliée des développeurs : la représentation des données. En effet, à moins que votre programme ne serve qu'à lire et à écrire des lettres à votre tante Ursule, vous aurez besoin de déterminer une structure pour stocker vos données et cette structure devra respecter deux impératifs : être facilement lisible par un programme et contenir un minimum d'informations dupliquées. Il y a donc une réflexion à mener en amont de la manipulation de fichiers.

1 Représentation des données

Les fichiers permettent de séparer les données des mécanismes de traitement. Lorsque vous manipulerez des fichiers, vous vous retrouverez confronté à l'un des trois cas suivants :

- soit vous créez un fichier qui sera lu par un de vos programmes : vous serez entièrement libre de déterminer la structure de vos données à l'intérieur du fichier.
- soit vous créez un fichier qui sera lu par un programme externe : vous devrez vous conformer strictement au format défini par l'application.
- soit vous lisez un ou des fichier(s) produit(s) par une source externe : vous vous conformerez au format défini par l'application ou bien vous traduirez les informations suivant une structure que vous aurez déterminée.

Pour être analysé simplement, un fichier doit contenir des zones distinctes et respecter strictement un format. Rien n'est plus pénible que d'analyser (ou « parser » pour employer un anglicisme) un fichier censé respecter un format connu mais qui ne le respecte pas tout à fait. Malheureusement, ces fichiers ne sont pas rares et sont parfois mis à disposition par des organismes de grande renommée. J'ai par exemple travaillé sur des fichiers de génomes bactériens du NCBI (*National Center for Biotechnology Information*) qui, dans de nombreux cas, ne respectaient pas le format énoncé.

Dans la structuration des fichiers, on retrouvera toujours plus ou moins les mêmes types d'architectures :

- un en-tête : des lignes indiquant le contenu du fichier.
- une séparation en sections : cette séparation pourra être des caractères spéciaux, le nom de la section en majuscules, etc.
- un mot-clé en début de ligne, indiquant le type des données de la ligne.
- un identifiant, suivi d'un caractère d'affectation et d'une valeur.
- des colonnes de données séparées par un caractère spécial (en général le point-virgule pour correspondre au format CSV). Dans ce cas, la première ligne contient généralement le titre de chaque colonne.

Voici quelques exemples de structures de fichiers :

- Fichier GBK définissant un génome bactérien :

```
LOCUS       NC_002570          4202352 bp    DNA     circular BCT 14-MAY-2010
DEFINITION  Bacillus halodurans C-125, complete genome.
ACCESSION   NC_002570
VERSION     NC_002570.2  GI:57596592
...
```

- Fichier `/etc/passwd` sous Linux :

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
...
```

- Fichier de données produit par analyse des déplacements suivant x, y, et z d'un objet. Les mesures sont prises toutes les 20ms :

```
# Déplacements en (x, y, z)
#Time 0.00
0 1e-05 2e-05 3e-05 4e-05 5e-05 6e-05 7e-05 8e-05 9e-05 0.0001
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0

#Time 0.02
0 1e-05 2e-05 3e-05 4e-05 5e-05 6e-05 7e-05 8e-05 9e-05 0.0001
1.23971e-05 1.24278e-05 1.24583e-05 1.24889e-05 1.25193e-05 1.25497e-05
1.258e-05 1.26103e-05 1.26404e-05 1.26706e-05 1.27006e-05
0 0 0 0 0 0 0 0 0 0 0
...
```

- Fichier CSV contenant les notes d'étudiants lors d'examens :

```
Prénom;Nom;TP 1;TP 2;TP 3;TP 4;TP
5;Examen;Note Finale
Linus;Torvalds;20;20;20;20;20;20;20
...
```

Si les données des fichiers ne peuvent pas être exploitées directement, il faudra passer par une étape de réorganisation des données. Si les données sont statiques et n'évoluent plus, vous pourrez créer une « vue » au sens base de données : depuis un ou plusieurs fichier(s) vous ne récupérez que les données qui vous intéressent et vous les restructurez dans un nouveau fichier. Vous gagnerez ainsi beaucoup de temps...

2 L'accès aux fichiers texte

La lecture ou l'écriture dans un fichier peut être vue comme la lecture dans un livre ou l'écriture dans un cahier. Plusieurs étapes sont nécessaires :

1. Choisir le livre ou le cahier, l'ouvrir et déterminer si l'on va lire, écrire ou ajouter des informations.
2. Lire le livre ou écrire dans le cahier.
3. Refermer le livre ou le cahier.

Ces étapes se retrouvent dans la manipulation de fichiers en Python.

2.1. Ouverture du fichier

Pour ouvrir un fichier il va falloir être en possession de deux informations : son nom et la méthode d'accès que l'on souhaite employer. Pour les fichiers texte, il existe trois méthodes d'accès :

- `r` (pour « *read* ») : ouverture en lecture.
- `w` (pour « *write* ») : ouverture en écriture. Si le fichier existe, il sera « écrasé » (détruit, puis remplacé par un nouveau fichier vide), sinon il sera créé.
- `a` (pour « *append* ») : ouverture en ajout. Si le fichier existe et qu'il contient des données, les anciennes données seront conservées et les nouvelles données seront ajoutées à la fin. Si le fichier n'existe pas, il sera créé et les données seront ajoutées comme avec une ouverture en écriture classique.

Pour connaître à tout moment à quel fichier on fait référence lorsque l'on désire lire ou écrire des données, nous allons créer un « objet-fichier ». Cet objet-fichier est un descripteur de fichier, c'est-à-dire une variable identifiant le fichier. La syntaxe de création de cette variable et donc d'ouverture d'un fichier est :

```
>>> fic = open('nom_fichier.txt', 'mode')
```

Sur un exemple réel, si l'on souhaite ouvrir le fichier `data.txt` en lecture, il faudra écrire :

```
>>> fic = open('data.txt', 'r')
```

Il faut noter que les fichiers sont toujours recherchés dans le répertoire courant. Si vous souhaitez ouvrir le fichier `/home/login/repertoire/data.txt` et que votre script se trouve dans `/home/login/bin`... qu'allez-vous faire ? Copier le fichier `data.txt` dans le répertoire du script ? Ce n'est pas très propre ! En fait, il suffit d'indiquer à Python que l'on souhaite changer de répertoire grâce à la fonction `chdir()` du module `os` :

```
>>> from os import chdir
>>> chdir('/home/login/repertoire')
>>> fic = open('data.txt', 'r')
```

Ici, nous avons bien ouvert le fichier `/home/login/repertoire/data.txt` en lecture.

2.2. Manipulation des données du fichier

2.2.1. Lecture

Il existe plusieurs méthodes de lecture. Pour bien illustrer les différences, nous allons partir d'un fichier texte simple contenant les données suivantes :

```
1 Première ligne
2 Deuxième ligne
3 Troisième ligne
```

Ce fichier, nommé `data.txt`, a été ouvert en lecture en utilisant la commande `open()` vue précédemment :

```
>>> fic = open('data.txt', 'r')
```

Voici donc les différentes méthodes de lecture.

2.2.1.1. Lecture intégrale du fichier

Il est possible de lire tout le fichier en une seule commande et de traiter les données par la suite. Cette méthode s'applique préférentiellement aux petits fichiers, car la consommation mémoire sera proportionnelle à la taille du fichier lu.

On peut obtenir le contenu du fichier dans une chaîne de caractères à l'aide de la méthode `read()` :

```
>>> texte = fic.read()
>>> texte
'1 Premi\xc3\xa8re ligne\n2 Deuxi\xc3\
xa8me ligne\n3 Troisi\xc3\xa8me ligne\n'
```

Vous pouvez constater que la chaîne obtenue est « brute » : on peut voir les caractères de saut de ligne `\n` et l'encodage des caractères accentués. Nous avons affiché ici le contenu de la variable texte sans contexte particulier. En utilisant la commande `print`, ces différents caractères seront interprétés pour l'affichage :

```
>>> print texte
1 Première ligne
2 Deuxième ligne
3 Troisième ligne
```

On peut également lire intégralement un fichier à l'aide de la méthode `readlines()`. Cette fois-ci, les données seront stockées dans une liste où chaque élément correspondra à une ligne du fichier (une ligne se termine par un caractère de retour à la ligne `\n`).

```
>>> texte = fic.readlines()
>>> texte
[]
```

Mais que se passe-t-il ? Nous obtenons une liste vide alors que nous avions réussi à lire le fichier précédemment ? C'est justement le problème ! Lorsque vous lisez un fichier, il y a un pointeur qui se déplace sur le fichier pour indiquer où devra commencer la prochaine lecture. Comme nous avions lu entièrement le fichier avec la méthode `read()`, le pointeur se trouvait à la fin du fichier et une nouvelle lecture démarrait à cet endroit... produisant un résultat vide. Si vous devez accéder plusieurs fois aux données d'un même fichier, pensez donc à le fermer et à le rouvrir :

```
>>> fic.close()
>>> fic = open('data.txt', 'r')
>>> texte = fic.readlines()
>>> texte
['1 Premi\xc3\xa8re ligne\n', '2 Deuxi\xc3\
xa8me ligne\n', '3 Troisi\xc3\xa8me ligne\n']
>>> for ligne in texte:
...   print ligne
...
1 Première ligne

2 Deuxième ligne

3 Troisième ligne
```

Notez ici la ligne vide apparaissant entre chaque ligne du fichier : la commande `print` affiche `texte[0]`, puis `texte[1]`, etc., puis effectue un retour à la ligne. Or, dans chaque variable `texte[i]` il y a aussi un retour à la ligne et on obtient donc un double retour à la ligne et une ligne vide. Nous verrons dans la section « Traitement des données » comment résoudre ce problème.

2.2.1.2. Lecture par tranches de caractères

Lorsque la fonction `read()` est utilisée en lui passant en paramètre un entier `n`, elle ne renvoie que les `n` caractères lus depuis le pointeur de fichier. On peut ainsi lire un fichier par tranches de `n` caractères, ce qui peut être très utile suivant les formats (par exemple, si chaque ligne représente des colonnes d'un nombre de caractères fixé).

```
>>> fic = open('data.txt', 'r')
>>> texte = fic.read(11)
>>> texte
'1 Premi\xc3\xa8re'
>>> print texte
1 Première
>>> texte = fic.read(8)
>>> texte
' ligne\n2'
>>> print texte
 ligne
2
```

Ici, on commence par lire les 11 premiers caractères du fichier, puis les 8 suivants.

2.2.1.3. Lecture ligne à ligne

Enfin, la méthode `readline()` permet de lire un fichier ligne à ligne : le pointeur de fichier s'arrêtera dès qu'il rencontrera un caractère `\n`.

```
>>> fic = open('data.txt', 'r')
>>> texte = fic.readline()
>>> texte
'1 Premi\xc3\xa8re ligne\n'
>>> print texte
1 Première ligne

>>> texte = fic.readline()
>>> print texte
2 Deuxième ligne
```

2.2.2. Écriture

Pour écrire dans un fichier, la commande est très simple : il s'agit de `write()`. Cette méthode prend en paramètre une chaîne de caractères qui sera inscrite dans le fichier :

```
>>> fic.write('Phrase à écrire dans le
fichier référencé par l'objet-fichier fic')
```

2.3. Fermeture du fichier

Pour fermer un fichier, on utilisera la méthode `close()` :

```
>>> fic.close()
```

Le nombre maximum de descripteurs de fichiers ouverts par processus est fixé par défaut à 1024 (tapez dans une console la commande `ulimit -n` pour vous en convaincre). Lorsque vous quittez votre programme Python, l'ensemble des descripteurs ouverts sont automatiquement fermés. Il est toutefois important de refermer un fichier ouvert pour plusieurs raisons :

- Le nombre maximum de descripteurs de fichiers est fixé à 1024, mais l'administrateur a pu diminuer ce nombre.
- Si vous travaillez sur plusieurs fichiers dans un même script, votre code sera plus lisible et vous éviterez de nombreuses erreurs.
- C'est plus propre : quand on ouvre quelque chose, on le referme...

Nous avons vu comment accéder aux données, mais comment les traiter ? On ne peut pas toujours manipuler directement les informations qui ont été lues...

3 Traitement des données

Pour utiliser les données lues depuis un fichier, il existe de nombreuses fonctions qui sont des fonctions de traitement des chaînes de caractères. Voici trois fonctions essentielles :

- `strip()` : supprime les caractères non visibles (espaces, tabulations, retours à la ligne) du début et de la fin d'une chaîne de caractères. Cette fonction fait partie du module `string` et comporte deux variantes : `lstrip()` qui ne supprime les caractères qu'au début de la chaîne (« l » pour *left* - gauche) et `rstrip()` qui ne supprime les caractères qu'à la fin de la chaîne (« r » pour *right* - droite). Voici un exemple d'utilisation :

```
>>> texte = '   Linux Pratique\n   '
>>> from string import strip, lstrip, rstrip
>>> strip(texte)
'Linux Pratique'
>>> lstrip(texte)
'Linux Pratique\n   '
>>> rstrip(texte)
'   Linux Pratique'
```

- `split()` : découpe une chaîne de caractères suivant un ou des caractère(s) donné(s) et crée une liste contenant les différents éléments coupés. Cette fonction sera très utile si vous avez structuré vos données sous forme de colonnes séparées par un caractère spécifique. Voici un exemple utilisant un format simulant le fichier de mots de passe Linux :

```
>>> texte = 'login:ae234fG4eZE:/home/login:bash'
>>> texte.split(':')
['login', 'ae234fG4eZE', '/home/login', 'bash']
```

- `join()` : méthode inverse de `split()` permettant de créer une chaîne de caractères en collant bout-à-bout les éléments d'une liste et en les séparant par un caractère spécifique. Cette méthode est utile pour écrire un fichier de données structurées en colonnes. Voici un exemple d'application :

```
>>> data = ['login', 'ae234fG4eZE', '/
home/login', 'bash']
>>> ":".join(data)
'login:ae234fG4eZE:/home/login:bash'
```

Et bien sûr, pour retrouver des données précises, vous pouvez utiliser les expressions régulières vues précédemment dans ce hors-série.

4 Enregistrement et restitution de variables

Il est possible que vous ayez besoin de sauvegarder des données présentes dans des structures complexes comme des listes ou des dictionnaires :

```
>>> data = {'entier':1, 'liste':[1, 2, 3],
'chaine':'Linux Pratique'}
```

La solution consisterait ici à décomposer notre dictionnaire en éléments et à les sauvegarder puis, lors de la lecture, à recréer la structure de dictionnaire. Bien sûr, ce système ne fonctionnerait que dans ce cas précis et il faudrait l'adapter pour pouvoir traiter une autre structure...

Python dispose d'un module permettant d'effectuer ce genre de tâches : le module `pickle`. Grâce à lui, vous pourrez enregistrer et charger des variables ayant des structures complexes grâce à la sérialisation : `pickle` va transformer les informations des variables sous forme de chaînes de caractères et sera capable de reconstituer les variables à partir de ces informations.

Notez que l'utilisation de `pickle` est très dangereuse : n'utilisez que des fichiers pour lesquels les données peuvent être vérifiées, car le chargement d'un fichier corrompu peut conduire à l'exécution d'un code malicieux !

Le module `pickle` existe sous deux déclinaisons : l'implémentation Python avec le module `pickle` et une implémentation en C, plus rapide, avec le module `cPickle`. Lors d'une utilisation de ce module, il peut donc être intéressant d'essayer de charger d'abord `cPickle` (et de le renommer en `pickle` pour n'avoir qu'un seul code d'appel des différentes fonctions) puis, si ce chargement échoue, de charger `pickle` :

```
>>> try:
...     import cPickle as pickle
... except:
...     import pickle
...
```

Pour sauvegarder les données, il faudra ouvrir le fichier en mode écriture binaire `'wb'` et bien sûr, lors de la lecture, ce sera le mode lecture binaire `'rb'`. On utilisera la fonction `dump()` pour écrire une variable en précisant en paramètre la variable et l'objet-fichier et, pour lire les variables d'un fichier, on utilisera la fonction `load()` en lui passant en paramètre l'objet-fichier référençant le fichier de données. Ainsi, pour sauvegarder puis lire la variable `data` présentée précédemment, après avoir importé le module `pickle` il faudra effectuer :

```
>>> fic = open('data', 'wb')
>>> pickle.dump(data, fic)
>>> fic.close()
>>> fic = open('data', 'rb')
>>> var = pickle.load(fic)
>>> var
{'liste': [1, 2, 3], 'chaine': 'Linux
Pratique', 'entier': 1}
```

Si vous enregistrez plusieurs variables dans un même fichier, il faudra effectuer des appels successifs à `load()` pour les lire une à une.

Si vous voulez avoir un aperçu de ce à quoi ressemble la sérialisation d'une variable, vous pourrez utiliser la fonction `dumps()`, qui effectue la même tâche que `dump()`, mais dans une chaîne de caractères et non un fichier :

```
>>> pickle.dumps(data)
"(dp1\nS'liste'\np2\n(lp3\nI1\naI2\naI3\
nasS'chaine'\np4\nS'Linux Pratique'\np5\
nsS'entier'\np6\nI1\ns."
```

5 Les fichiers de configuration

Un format de fichier très utile : le fichier de configuration (les fichiers d'extension `.ini`). La structure d'un fichier de configuration est la suivante :

```
[section 1]
option_1 = valeur_1
option_2 = 12

[section 2]
option_3 = valeur_3
...
```

On peut donc grouper des listes d'options contenant des valeurs de différents types et organisées en sections. La lecture de ces fichiers est simplifiée par le module `ConfigParser`, qui permet de récupérer les informations en spécifiant la section et le nom de l'option. Il est également possible d'utiliser ce module pour l'écriture des fichiers, mais l'intérêt sera ici plus discutable.

Voici un fichier de configuration `lp.ini` qui nous servira pour la mise en pratique de ce module :

```
[web]
name = Linux Pratique
url  = http://www.linux-pratique.com

[article]
author = Tristan Colombo
title  = Manipulation de fichiers avec Python
```

Nous avons deux sections qui comportent chacune deux options. L'accès à la valeur d'une option, connaissant son nom et sa section, est très simple :

```
>>> from ConfigParser import SafeConfigParser
>>> parser = SafeConfigParser()
>>> parser.read('lp.ini')
['lp.ini']
>>> print parser.get('web', 'name')
Linux Pratique
```

Vous pouvez également récupérer le nom et la valeur de l'ensemble des options d'une section :

```
>>> for name in parser.options('article'):
...     print 'Option : ' + name
...     print '   ' + parser.get('article', name)
...
Option : author
   Tristan Colombo
Option : title
   Manipulation de fichiers avec Python
```

Je ne développerai pas ici l'écriture de fichiers de configuration depuis Python... dans les cas les plus courants, un simple éditeur de texte suffit.

Conclusion

Cet article nous a permis de découvrir comment manipuler des fichiers de données texte en Python. Vous avez pu voir que Python reste fidèle à lui-même : très peu d'instructions qui permettent d'effectuer simplement toutes les opérations dont nous pourrions avoir besoin.

Le point le plus important, celui qui nécessitera toute votre attention et qui vous prendra le plus de temps, est celui de la réflexion sur le format des données qui seront stockées dans vos fichiers. Mais comme toujours, un peu de temps perdu pour beaucoup de temps gagné ! ■

LIRE ET ÉCRIRE DES FICHIERS XML

Tristan Colombo

De nos jours, tout le monde connaît, au moins de nom, le format XML. Ce format de fichiers permet de décrire des données de manière structurée et il peut être utilisé de manière très simple en Python. Je vous propose dans cet article de découvrir ou de redécouvrir de façon sommaire le format XML et la manière d'analyser un fichier XML pour en retirer des informations à l'aide de XPath. Nous pourrons ensuite utiliser ces techniques depuis Python grâce à la bibliothèque lxml.

1 Le langage XML

XML, pour *eXtensible Markup Language*, est un langage à balises ou encore à tags, les termes étant différents mais désignant le même objet. Les tags sont représentés en encadrant des termes par < et >. Par exemple, `<tag>` est un tag. Le XML suit les recommandations émises par le W3C (*World Wide Web Consortium*). Cet organisme à but non lucratif est chargé de promouvoir la compatibilité des technologies web (XML, HTML, XHTML, etc.). Pour chacune de ces technologies, il publie un document – une recommandation – dans laquelle sont énoncées les règles permettant de construire un document correct.

En XML, les tags ne sont pas figés, c'est-à-dire que vous pouvez créer un tag ayant n'importe quel nom vous passant par la tête. Comme le but de ce langage est de produire des documents structurés, les tags vont permettre de déterminer des blocs définissant des objets précis. Par exemple, un livre contient un titre et un auteur. Pour décrire cet objet en XML nous utiliserons trois tags et comme ceux-ci définissent des blocs (« livre » contient « titre » et « auteur »), à chaque tag démarrant un bloc (on l'appellera « tag ouvrant » `<tag>`), nous associerons un tag fermant le bloc (on l'appellera « tag fermant » `</tag>`). Voici le code correspondant à notre exemple :

```
<livre>
  <titre>The Python Standard Library by Example</titre>
  <auteur>Doug Hellmann</auteur>
</livre>
```

Grâce à cette structure, nous pouvons définir un nombre infini de livres pour les stocker par exemple dans une bibliothèque. Nous aurons alors un élément, appelé « racine », qui sera un tag contenant l'ensemble des tags de notre document XML :

```
<bibliotheque>
  <livre>
    <titre>The Python Standard Library by Example</titre>
    <auteur>Doug Hellmann</auteur>
  </livre>
  <livre>
    ...
  </livre>
  ...
</bibliotheque>
```

Pour être complet, un document XML doit contenir des informations supplémentaires : le prologue. Ces informations permettent d'indiquer, sous la forme de deux lignes d'entête, que le document est un document XML, quelle version des spécifications XML il respecte, quel est l'encodage des caractères employé, quel est l'élément racine permettant de commencer la lecture du document et enfin quelle est l'adresse de la grammaire ou DTD (*Document Type Definition*) définissant le document. Dans notre exemple, la grammaire précisera qu'un livre contient un tag `titre` et un tag `auteur` et qu'une `bibliotheque` peut contenir plusieurs `livre`.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE bibliotheque SYSTEM "biblio_grammaire.dtd">
```

Nous ne rentrerons pas dans les détails de la création d'une grammaire dans cet article dont le but est la manipulation des fichiers XML en Python, mais si vous devez travailler avec des fichiers XML de votre cru, cette étape, bien que non obligatoire, est fortement recommandée pour s'assurer de l'intégrité du document (et aussi respecter les bonnes pratiques de développement).

Pour finir sur cette rapide introduction à XML, sachez que pour chaque tag il est possible d'associer des informations supplémentaires appelées « attributs » sous la forme `nom = "valeur"`. Par exemple, pour ajouter la langue dans laquelle un livre a été écrit, nous pourrions coder :

```
<livre lang="en">
  ...
</livre>
```

Si vous souhaitez ajouter des commentaires à vos fichiers XML, il faudra les encadrer par `<!--` et `-->` :

```
<!-- code non interprété -->
<livre lang="en">
  ...
</livre>
```

2 XPath ou comment naviguer dans un document XML ?

Une fois qu'un document est créé, il est important de pouvoir y retrouver facilement des informations. Pour cela, on va « naviguer » dans le document. Un document XML a une structure arborescente déterminée par ses tags. Dans un contexte de navigation, chaque tag sera appelé « nœud » (le tag de départ restant le « nœud » racine et les attributs devenant des « nœuds » attributs).

Un XPath va être un chemin permettant de retrouver une information dans l'arborescence d'un document XML. L'écriture d'un chemin simple se fera de la même manière qu'en shell en séparant chaque nœud par le caractère `/`. Pour voir les possibilités offertes par les XPath, il est plus pratique d'étudier un exemple de document. Nous allons reprendre l'exemple précédent qui sera légèrement modifié pour offrir plus de combinaisons :

```
01: <bibliotheque>
02:   <livre lang="en">
03:     <titre>The Python Standard Library by Example</titre>
04:     <auteur>Doug Hellmann</auteur>
05:     <annee>2011</annee>
06:     <isbn>978-0-321-76734-9</isbn>
07:   </livre>
08:
09:   <livre lang="fr">
10:     <titre>Programmer avec Python 3</titre>
11:     ...
12:   </livre>
13:
14:   ...
15: </bibliotheque>
```

Je vous propose maintenant d'étudier un certain nombre de chemins XPath :

- `/bibliotheque/livre` : récupère l'ensemble des nœuds « livre » contenus dans le nœud racine « bibliotheque » (lignes 2 à 7, 9 à 12, etc.).
- `//livre` : désigne n'importe quel nœud ayant pour nom « livre ». Ici le résultat sera identique à celui obtenu précédemment avec `/bibliotheque/livre`, mais il faut faire attention car dans ce chemin il n'y a pas de spécification de lien de parenté : il ne s'agit pas forcément du tag « livre » qui est immédiatement inclus dans le tag « bibliotheque ».
- `/bibliotheque/livre[1]` : il s'agit ici de récupérer le premier nœud « livre » contenu dans « bibliotheque ». Il s'agit donc des lignes 2 à 7. L'écriture entre crochets après un nom de tag permet de spécifier le numéro du tag que l'on souhaite récupérer. Il existe des fonctions qui peuvent être utilisées pour spécifier le ou les numéro(s) de tags à récupérer :
  - → `/bibliotheque/livre[last()]` : récupère le dernier nœud « livre » de « bibliotheque ».
  - → `/bibliotheque/livre[position()<2]` : récupère les deux premiers nœuds « livre ». Dans notre exemple, il s'agit des nœuds des lignes 2 à 7 et 9 à 12.
- `/bibliotheque/livre[annee>=2011]/titre` : recherche les nœuds « livre » pour lesquels il existe un nœud « annee » ayant une valeur supérieure ou égale à 2011 et récupère le titre (ligne 3 de notre exemple).
- `//livre[@lang="fr"]` : récupère tous les livres ayant un attribut « lang » dont la valeur est « fr » (lignes 9 à 12 de notre exemple).
- `//livre/@lang` : désigne tous les attributs « lang » associés à un nœud « livre » (lignes 2 et 9, mais seulement l'attribut et sa valeur).
- `/bibliotheque/livre/*` : récupère l'ensemble des tags inclus dans les nœuds « livre » de « bibliotheque » (lignes 3 à 6, 10 et 11, etc.).
- `/bibliotheque/livre/titre | /bibliotheque/libre/isbn` : recherche les nœuds « titre » et « isbn » dans « livre » (lignes 3, 6, 10, etc.). Le caractère `|` permet de grouper des conditions.
- `/bibliotheque/livre[contains(., "Python")]` : récupère les nœuds de « livre » qui contiennent le mot « Python » (le point en premier paramètre de la fonction `contains()` représente le contenu du nœud). Dans notre exemple, cela correspond aux lignes 3 et 10.
- `/bibliotheque/livre[contains(@state, "new")]` : écriture similaire à la précédente, mais où la contrainte va être exercée sur l'attribut « state ». On recherche ici tous les nœuds de « livre » qui contiennent un attribut « state » ayant pour valeur « new » (aucun nœud de notre exemple).

En utilisant cette écriture de chemins XPath, on peut effectuer des requêtes dans des documents XML pour obtenir des informations. On parle alors de XQuery. Il s'agit d'un système équivalent aux requêtes SQL utilisées pour interroger une base de données.

3 Utiliser la bibliothèque lxml

Il existe de nombreux modules en Python permettant de traiter les fichiers XML. Parmi les deux les plus utilisés, on retrouve le module `xml.etree.ElementTree` intégré dans la bibliothèque standard et la bibliothèque `lxml`. On retrouve plus ou moins les mêmes fonctionnalités entre les deux modules, mais les traitements effectués par `lxml` sont réputés plus rapides. J'utiliserai donc cette bibliothèque pour la suite.

Comme `lxml` ne fait pas partie de la bibliothèque standard, il faut l'installer... Elle est basée sur les bibliothèques `libxml2` et `libxslt` du C et il faut donc, avant d'installer le module, installer ces bibliothèques. Sur des distributions basées sur Debian :

```
sudo aptitude install libxml2-dev libxslt1-dev
```

Le module s'installe ensuite à l'aide de `pip` (paquetage `python-pip` si vous ne l'avez pas encore installé) :

```
pip install lxml
```

Le plus compliqué est fait : vous savez écrire des XQuery et `lxml` est installée... L'utilisation est ensuite un jeu d'enfant.

3.1. Lecture de fichiers XML

Avant utilisation d'un module, comme toujours en Python, il va falloir l'importer. Nous aurons besoin ici de l'objet `etree` :

```
from lxml import etree
```

Il faut ensuite indiquer le nom du fichier à analyser. Cela sera fait en créant un objet de type `etree` :

```
tree = etree.parse("mon_fichier.xml")
```

Il n'y a plus qu'à lancer les requêtes XQuery à l'aide de la méthode `xpath()` et à utiliser les résultats :

```
for node in tree.xpath("//livre"):
  print node.tag + " en " + node.get("lang")
```

L'attribut `tag` permet d'obtenir le nom du tag (ici « livre ») et la méthode `get()` permet d'accéder aux attributs du tag.

D'autres méthodes peuvent être utiles pour exploiter un nœud :

- la méthode `items()` permet d'obtenir la liste des attributs d'un tag :

```
for node in tree.xpath("//livre"):
  print node.items()
```

- l'attribut `text` donne accès au contenu d'un nœud :

```
for node in tree.xpath("//livre"):
  print node.text
```

- la méthode `getParent()` permet de remonter d'un nœud vers le parent (si l'on se trouve sur « livre » on remontera sur le nœud « bibliotheque »).

Les XQuery peuvent être appelées sur des résultats de requêtes XQuery de manière à les affiner. Exemple :

```
# *-* coding: utf-8 *-*

from lxml import etree

tree = etree.parse(mon_fichier.xml")

for livre in tree.xpath("//livre"):
  print "Le livre \"", livre.xpath("titre")[0].text, "\", paru en", livre.
xpath("annee")[0].text, ", a été écrit par ", livre.xpath("auteur")[0].
text, "(en ", livre.get("lang"), ")"
```

Dans cet exemple, après avoir obtenu les nœuds « livre », on applique à nouveau des XQuery pour obtenir les nœuds « titre », « annee » et « auteur ». Sur notre document XML d'exemple, le résultat est :

```
Le livre " The Python Standard Library by Example " paru en
2011 a été écrit par Doug Hellmann (en en)
Le livre " Programmer en Python 3 " paru en ...
...
```

3.2. Écriture de fichiers XML

Pour générer un document XML, il faudra créer l'arborescence du document pas-à-pas en créant des noeuds et en indiquant leur parent. Le premier nœud à créer est bien sûr la racine :

```
root = etree.Element("bibliotheque")
```

Ensuite, les nœuds sont créés à l'aide de l'objet `SubElement` :

```
livre = etree.SubElement(root, " livre ")
```

Nous avons créé ici un nœud « livre » dont le parent est root, c'est-à-dire « bibliotheque ». Nous pouvons ajouter des attributs à ce tag avec la méthode `set()` :

```
livre.set(" lang ", " en ")
```

L'attribut `text` permet toujours de spécifier le contenu d'un nœud :

```
titre = etree.SubElement(livre, "titre")
titre.text = "The Python Standard Library by Example"
```

Nous avons ici créé un nouveau nœud « titre » dont le père est « livre » et qui a pour valeur le titre du premier livre du document.

Une fois que tous les nœuds ont été créés, il n'y a plus qu'à générer le code XML sous forme de chaîne de caractères :

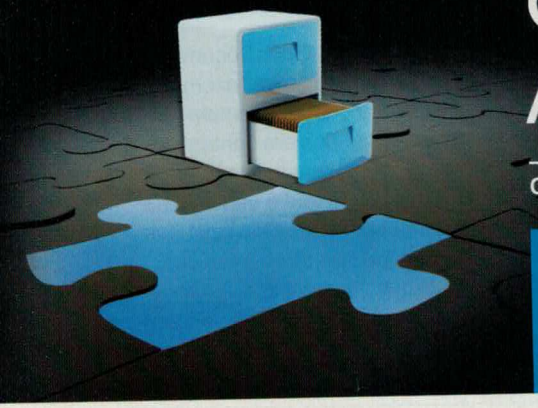
```
print etree.tostring(root, pretty_print=True)
```

Le paramètre `pretty_print` permet d'obtenir un code correctement indenté. Pour enregistrer votre document XML, vous n'aurez plus qu'à utiliser les fonctions standards de Python `open()`, `write()`, et `close()`.

Conclusion

La manipulation de fichiers XML est très simple en Python. Les points sur lesquels vous passerez le plus de temps seront la réflexion sur la structure de vos fichiers et l'écriture des XPath.

Même si ce format peut apparaître comme le meilleur format de stockage de données quelles qu'elles soient, gardez à l'esprit qu'il est particulièrement verbeux (utilise beaucoup de caractères inutiles en machine) et ne convient donc pas à tout type de données. La folie du « tout XML » s'est éteinte non sans raison : il faut toujours choisir le format le plus adapté aux données que l'on souhaite manipuler et à partir du moment où un format a été correctement pensé et standardisé, il n'y a pas de mauvais format... Il n'y a que de mauvais usages ! ■



CRÉER ET EXPLOITER DES ARCHIVES AVEC PYTHON

Olivier Delhomme

Dans cet article, nous allons découvrir comment manipuler les fichiers compressés de type gzip/bzip2 ainsi que les archives de type zip/tar.

1 Rappel sur les différences gzip/bzip2 et zip/tar

Les fichiers au format gzip ou bzip2 sont dits compressés. Seul l'algorithme de compression (et de décompression) les différencie. Bzip2 est le plus efficace en compression, mais aussi bien plus demandeur en ressources (le nouveau format xz est encore plus efficace tout en étant moins gourmand en ressources). Ces formats de compression ne concernent qu'un seul fichier. Ils ne permettent pas l'inclusion de plusieurs fichiers ou de répertoires entiers contrairement aux fichiers que je nomme « archives » dont le format peut être de type zip ou tar entre autres.

Le type de fichier tar n'est pas compressé par défaut. Toutefois, il est le plus souvent compressé à l'aide de gzip ou bzip2 (en réalité, il est très rare de trouver des archives tar non compressées). Enfin, le type de fichier zip est quant à lui généralement compressé par défaut avec un taux moyen. Le module `zipfile` de Python ne compresse pas par défaut et utilise l'archive zip comme un conteneur simple. Il est toutefois possible de lui demander d'effectuer la compression et il me semble que ce serait dommage de s'en priver avec les processeurs actuels.

Note

Le module permettant de gérer le format de fichier xz, qui est basé sur la compression LZMA, est en cours d'inclusion dans Python (cf. [xz]) ainsi que dans le module `tarfile` qui est survolé dans cet article.

2 Utiliser gzip ou bzip2 avec Python

2.1 Exemple de lecture d'un fichier

Ces deux modules donnent accès aux fonctions des bibliothèques écrites en C qui font le travail effectif. L'utilisation de leurs fonctionnalités de base est identique. En effet, ces deux modules fournissent des classes qui surchargent certaines fonctions d'utilisation de fichiers telles `open()` (seulement disponible dans le module `gzip` – on utilisera `BZ2File()` pour ouvrir un fichier bz2), `close()`, `read()`, `readline()`, `write()`, `writelines()`. Ainsi, on accède à un fichier de type gzip ou bzip2 exactement comme s'il s'agissait d'un fichier « normal ». Attention, en Python 3 les modules retournent une variable de type `bytes` qu'il faudra « décoder » si l'on souhaite la transformer en chaîne de caractères.

Voici un exemple, simplifié à l'extrême, qui permet d'afficher le contenu de fichiers textes qu'ils soient plats ou compressés avec gzip ou bzip2. Le lecteur soucieux d'écrire un programme plus propre pourra, en s'inspirant des articles de ce hors-série, ajouter des blocs `try...except` pour la gestion d'erreurs systèmes, utiliser le module `magic` [magic] pour détecter le format du fichier (plutôt que de se baser sur son extension) et enfin gérer les éventuelles erreurs de décodage des chaînes issues des fichiers compressés :

```
#!/usr/bin/python3
# -*- coding: utf8 -*-

import os
import sys
import getopt
import gzip
import bz2

opts, args = getopt.getopt(sys.argv[1:], [], [])

for fichier in args:

    if '.gz' in (fichier):
        the_file = gzip.open(fichier, 'rb')

    elif '.bz2' in (fichier):
        the_file = bz2.BZ2File(fichier, 'rb')

    else:
        the_file = open(fichier, 'r')


    for line in the_file:

        if type(line) is str:
            print('%s' % line, end='')

        elif type(line) is bytes:
            print('%s' % line.decode('utf_8',
'ignore'), end='')

    the_file.close()
```

On pourra appeler pompeusement ce programme `cat.py` et l'utiliser comme suit (sur des fichiers qui ne contiennent que du texte, comme des fichiers de log) :

```
./cat.py *.log *.log.gz *.log.bz2
```

2.2 Exemple d'ajout de données dans un fichier gzip

Les remarques concernant la gestion des erreurs et des exceptions de l'exemple précédent s'appliquent ici également. Cet exemple montre

comment ajouter des données (ici, il s'agit de données issues de fichiers) à une archive de type gzip. Attention, ici on ajoute bien le contenu de fichiers à la suite de notre fichier compressé. Décompresser le fichier obtenu ne permettra pas de récupérer les fichiers que nous lui avons ajoutés, mais bien un seul fichier contenant l'ensemble, les uns à la suite des autres.

J'ai mis une option (que j'ai rendue obligatoire) sur la ligne de commandes pour indiquer le nom du fichier qui sera le fichier compressé auquel on souhaite ajouter des données. En effet, se baser sur l'ordre des arguments pour cela n'est pas possible, car ils ne sont pas forcément retournés dans l'ordre dans lequel ils ont été entrés.

Note

Le module `getopt` est toujours valable pour l'utilisation dans Python 3. Seul le module `optparse` est déprécié à partir de la version 3.2 et remplacé par le module `argparse` [argparse]. Ce module permet beaucoup plus de choses sur la ligne de commandes comme les options comportant un nombre d'arguments variable.

Essayer ce programme avec un fichier de type bzip2 ne fonctionnera pas ! Il conviendra de remplacer `gzip.GzipFile` par `bz2.BZ2File` pour obtenir le même résultat avec un fichier de type bzip2. En effet, ce programme a été écrit pour montrer l'utilisation de la commande `with` qui a été introduite dans Python 3.1 [with] pour le module `gzip`. Le module `bzip2` semble supporter cette instruction depuis Python 3.0 (elle a été introduite dans Python 2.7 pour faciliter le passage à la version 3).

Le lecteur soucieux d'une plus grande généralité pourra récrire ce programme à la manière de l'exemple précédent :

```
#!/usr/bin/python3
# -*- coding: utf8 -*-

import os
import sys
import getopt
import gzip

short_options = "f:"
long_options = ["file="]
archive = ''
opts, args = getopt.getopt(sys.argv[1:],
short_options, long_options)

for opt, arg in opts:

    if opt in ('-f', '--file'):
        archive = arg

if archive != '' and len(args) > 1:

    with gzip.GzipFile(archive, 'ab') as arch:

        for fichier in args:

            the_file = open(fichier, 'rb')
            arch.writelines(the_file.readlines());
            print('Added %s' % fichier)
            the_file.close()

            arch.close()
```

3 Utiliser les fichiers d'archives de type zip

Le module `zipfile` qui permet d'utiliser les archives zip dans Python ne permet pas l'utilisation d'archives scindées en plusieurs morceaux, mais en revanche, peut utiliser des archives dont la taille est supérieure à 4Go à condition toutefois qu'elles contiennent les extensions ZIP64. De même, ce module ne gère pas encore le chiffrement des données mais seulement le déchiffrement (qui est, aux dires de la page de documentation [zipfile], extrêmement lent).

Tout d'abord, ce module contient une fonction qui permet de vérifier si un fichier est, ou non, une archive zip : `zipfile.is_zipfile('mon_fichier.zip')`. Elle renvoie `True` quand le fichier est bien une archive zip et `False` sinon.

3.1 Lister les fichiers contenus dans une archive zip

Le module `zipfile` définit une classe `ZipInfo` qui stocke les données descriptives des fichiers contenus dans l'archive zip. On trouvera notamment le nom du fichier, les éventuels commentaires associés, la taille originelle du fichier, sa taille une fois compressé et la date de sa dernière modification (dans l'archive). Il existe de nombreux autres paramètres (comme un code de correction d'erreur basique CRC32) que le lecteur curieux pourra découvrir sur [zipinfo].

Les fichiers au format OpenDocument (c'est le cas de cet article par exemple) sont au format zip. Ainsi, l'exemple suivant, qui liste les fichiers contenus dans l'archive avec leurs tailles, met en évidence que la taille totale des fichiers OpenDocument est principalement due à un fichier image miniature (plus de 70% dans le cas de cet article !). C'est cette image qui est affichée par les gestionnaires de fichiers (comme Thunar) lorsqu'ils affichent un fichier de ce type.

```
zfile = zipfile.ZipFile('mon_article.odt', 'r')
for zinfo in zfile.infolist():
    print('%s (%s --> %s)' % (zinfo.filename,
zinfo.file_size, zinfo.compress_size))
zfile.close()
```

3.2 Extraire un fichier d'une archive zip

Voyons comment extraire cette fameuse image qui se trouve dans un dossier virtuel nommé `Thumbnails` contenu dans l'archive. L'image en elle-même se nommant `thumbnail.png`. La classe `ZipFile` implémente la fonction `read` qui permet de lire un fichier dans une archive zip (les données sont décompressées de manière transparente si nécessaire). L'exemple suivant montre comment lire ce fameux fichier image et le sauvegarder dans un nouveau fichier qui sera nommé `image.png`. Il faut bien prendre soin de préciser à la fonction `open` qu'il s'agit d'un fichier binaire que nous voulons créer en ajoutant `b` après le `w`. En effet, les données que nous lisons dans l'archive et que nous allons écrire dans ce nouveau fichier sont de type binaire.

```
zfile = zipfile.ZipFile('mon_article.odt', 'r')
zdata = zfile.read('Thumbnails/thumbnail.png')
fichier = open('image.png', 'wb')
fichier.write(zdata)
fichier.close()
zfile.close()
```

Il existe également une fonction toute faite nommée `extract` qui permet de réaliser une opération similaire. La fonction `extractall`, quant à elle, extraira l'ensemble des fichiers de l'archive. Ces fonctions recréent les arborescences, c'est-à-dire que

l'utilisation de la fonction sur l'exemple précédent : `zfile.extract('Thumbnails/thumbnail.png')` créera le répertoire `Thumbnails` et le fichier `thumbnail.png` à l'intérieur de celui-ci.

Note

La documentation indique qu'il est préférable d'utiliser la fonction `extractall` qui est plus sécurisée que la fonction `extract` notamment sur la gestion des extractions possibles des fichiers dans un répertoire se trouvant dans un niveau supérieur à celui où l'archive est extraite.

3.3 Inclure un nouveau fichier dans une archive zip existante

L'inclusion d'un fichier dans une archive existante est réalisée grâce à la fonction `write`. Au préalable, on aura pris soin d'ouvrir l'archive en mode ajout `a` ou écriture `w`. Attention toutefois si vous ouvrez une archive existante avec le mode d'écriture, les fichiers qu'elle contient seront effacés ! Nous pouvons par exemple inclure le logo du projet Heraia [heraia] dans notre document précédent :

```
zfile = zipfile.ZipFile('mon_article.odt', 'a')
zfile.write('heraia.png', arcname='Logos/heraia.png')
zfile.close()
```

Ici, nous ajoutons dans l'archive `mon_article.odt` le fichier `heraia.png` qui se trouve dans le même dossier que le script. Le paramètre `arcname` permet de spécifier un chemin et un nom de fichier. Ici, le fichier est placé dans un dossier virtuel nommé `Logos`. Il aurait également été possible de renommer le fichier dans la même opération en spécifiant `arcname='Logos/heraia_256x256.png'` par exemple.

Le fichier est ajouté tel quel, sans compression, puisque c'est le comportement par défaut. Afin de pouvoir bénéficier de la compression, il faut utiliser le paramètre `compress_type` avec la fonction `write` (et ce ne sera valable que pour cette écriture bien précise, indépendamment du reste de l'archive) ou bien utiliser le paramètre `compression` à l'ouverture du fichier. Ces paramètres ne prennent que deux valeurs : `ZIP_STORED` (on ne compresse pas) ou `ZIP_DEFLATED` (on compresse) :

```
zfile = zipfile.ZipFile(archive, 'a', compression=zipfile.ZIP_
DEFLATED)
```

ou

```
zfile.write('kern.log', arcname='logs/kern.log', compress_
type=zipfile.ZIP_DEFLATED)
```

Je vous recommande, lorsque vous insérez un fichier dans une archive existante, de bien vérifier qu'il n'y soit pas déjà. En effet, s'il existe déjà, cela ajoutera bien le fichier, mais créera ainsi un doublon (cf. le bug [doublon] toujours ouvert à l'heure où j'écris ces lignes) qu'il vous sera bien difficile de différencier par la suite (cf. note ci-dessous).

Note

Le module `zipfile` ne fournit pas de fonction permettant de supprimer un fichier dans une archive. Comme il n'est pas possible de remplacer un fichier d'une archive zip en incluant un nouveau fichier ayant le même nom, car il est simplement ajouté à l'archive. L'extraction avec la fonction `extractall` de deux fichiers comportant deux noms identiques aura pour résultat l'écrasement du premier fichier par le deuxième (celui inclus le plus récemment dans l'archive). Ce comportement n'est pas celui de l'utilitaire unzip qui posera la question à l'utilisateur de savoir que faire avec le doublon : « replace Thumbnails/thumbnail.png? [y]es, [n]o, [A]ll, [N]one, [r]ename: ».

L'ajout d'un fichier dans une nouvelle archive ne diffère en rien de celui dans une archive pré-existante. Attention, créer une archive de type zip et n'y ajouter aucun fichier, puis la fermer produira un fichier vide qui ne sera pas reconnu comme une archive zip !

3.4 Ajouter ses propres données dans les fichiers d'une archive zip

Nous savons maintenant ajouter (fonction `write`) et extraire (fonctions `extract` et `extractall`) des fichiers d'une archive de type zip. Nous savons même comment lire les données d'un fichier sans l'extraire (fonction `read`). Il serait peut-être temps de voir comment remplir des fichiers d'une archive avec nos données !

Avant toute chose, il faut avoir ouvert l'archive dans laquelle on souhaite écrire, soit en mode ajout `a`, soit en mode écriture `w`. Écrire dans un fichier de l'archive ainsi ouverte se réalise à l'aide de la fonction `writestr`. Elle prend pour arguments le nom du fichier dans lequel on souhaite écrire (ou un objet `zipinfo`) et une chaîne de caractères de type bytes (qui contient les données à écrire).

L'exemple ci-dessous crée l'archive zip nommée `nombres.zip` et y ajoute deux fichiers nommés `premiers.txt` et `periodes.txt` contenant respectivement des nombres premiers et les différences entre eux. L'archive est ouverte en mode `ZIP_DEFLATED`, ce qui indique que les données seront compressées de manière transparente.

```
premiers = b'1\n2\n3\n5\n7\n11\n13\n'
periodes = b'1\n1\n2\n2\n4\n2\n'
zfile = zipfile.ZipFile('nombres.zip', 'w', compression=zipfile.
ZIP_DEFLATED)
zfile.writestr('premiers.txt', premiers)
zfile.writestr('periodes.txt', periodes)
zfile.close()
```

Vérifions cela :

```
$ unzip nombres.zip
Archive:  nombres.zip
  inflating: premiers.txt
  inflating: periodes.txt

$ cat premiers.txt
1
2
3
5
7
11
13

$ cat periodes.txt
1
1
2
2
2
4
2
```

4 Utiliser les fichiers d'archive de type tar

L'archive de type tar est une archive non compressée. Comme dans la vraie vie il est très rare d'en rencontrer une qui ne le soit pas, le module `tarfile` [tarfile] permet la manipulation de ces archives qu'elles soient compressées (via gzip ou bzip2 - xz est pour bientôt !) ou non.

Les interfaces entre le module `zipfile` et le module `tarfile` ne sont pas encore complètement uniformisées, mais les fonctionnalités sont quasiment identiques. En effet, il est possible de vérifier si un fichier est bien de type tar grâce à la fonction `is_tarfile('monfichier.tar.gz')`. Elle renvoie `True` si c'est le cas et `False` sinon. De plus elle générera une erreur `IOError` si le fichier n'existe pas ou ne peut être lu.

4.1 Lister les fichiers contenus dans une archive tar

Pour lister le contenu d'une archive, le module `tarfile` procure une classe nommée `tarinfo` qui remplit un rôle similaire à `zipinfo`. Chaque objet de type `tarinfo` [tarinfo] contient le nom d'un fichier ainsi que quelques attributs qui lui sont associés tels sa taille, son type, le nom et le groupe de l'utilisateur auquel il appartient (au sens Unix du terme), ses permissions, etc. La fonction `getmembers`, sans paramètres, renvoie la liste des objets `tarinfo` d'une archive de type tar. C'est-à-dire la liste des fichiers de l'archive avec tous leurs attributs :

```
#!/usr/bin/python3
# -*- coding: utf8 -*-

import tarfile

t = tarfile.open('exemple2.tar.gz', 'r')

for finfo in t.getmembers():

    print('%s a une taille de %d octets' %
(finfo.name, finfo.size), end='')

    if finfo.isdir():
        print(' et est un répertoire')
    elif finfo.isfile():
        print(' et est un fichier normal')
```

Ici, le type de l'objet `tarinfo` (variable nommée `finfo`) est testé grâce aux fonctions booléennes `isdir` et `isfile`. Il existe 6 autres fonctions de ce type qui permettent de savoir si un fichier donné d'une archive tar est un lien symbolique, un périphérique, un tube nommé, etc.

4.2 Extraire un fichier d'une archive tar

Extraire un fichier d'une archive tar se réalise pratiquement de la même manière que pour une archive zip. En effet, il existe les deux fonctions `extract` et `extractall` qui permettent respectivement d'extraire un fichier ou tous les fichiers de l'archive. La note sur la sécurité des deux fonctions identiques du module `zipfile` est également valable ici.

Il existe également la fonction `extractfile` qui renvoie un objet de type fichier. Avec cet objet, il est possible de réaliser la lecture de ses données et donc de les écrire dans un autre fichier (comme je l'ai fait dans l'exemple pour l'archive de type zip). Ainsi, l'exemple suivant lit le fichier nommé `premiers.txt` qui est présent dans l'archive `exemple2.tar.gz` et copie ses données dans un nouveau fichier (hors de l'archive) nommé `nombres_premiers.txt` la décompression ayant été effectuée de manière transparente (qu'elle soit de type gzip ou de type bzip2) :

```
t = tarfile.open('exemple2.tar.gz', 'r')
extr = t.extractfile('premiers.txt')
fichier = open('nombres_premiers.txt', 'wb')

for lines in extr:
    fichier.write(lines)

fichier.close()
extr.close()
t.close()
```

4.3 Créer une archive tar et y ajouter des fichiers

La création d'une archive tar est différente de celle d'une archive zip en ceci qu'il existe la possibilité de choisir non seulement si l'archive sera ou non compressée, mais également le type de la compression (gzip ou bzip2 vus précédemment). Si la lecture peut s'effectuer de manière transparente quel que soit le type de compression, l'écriture nécessite quant à elle de spécifier clairement le type de compression. Ainsi, si le mode d'ouverture est `w`, l'archive sera créée sans compression. S'il s'agit de `w:gz` le type de compression gzip sera choisi. Enfin, s'il s'agit de `w:bz2` ce sera le type bzip2 :

```
t = tarfile.open('premiers.tar', 'w')
tz = tarfile.open('premiers.tar.gz', 'w:gz')
tbz2 = tarfile.open('premiers.tar.bz2', 'w:bz2')

t.add('premiers.txt')
tz.add('premiers.txt')
tbz2.add('premiers.txt')

t.close()
tz.close()
tbz2.close()
```

Cet exemple crée trois archives respectivement non compressée, compressée de type gzip et enfin compressée de type bzip2. Le fichier `premiers.txt` est ajouté dans chacune de ces archives à l'aide de la fonction `add`. Ce qui donne les fichiers suivants :

```
$ ls -ls premiers.t*
12 -rw-r--r-- 1 dup dup 10240 2012-01-03 22:03 premiers.tar
 4 -rw-r--r-- 1 dup dup   142 2012-01-03 22:03 premiers.tar.bz2
 4 -rw-r--r-- 1 dup dup   147 2012-01-03 22:03 premiers.tar.gz
 4 -rw------- 1 dup dup    16 2011-12-30 17:59 premiers.txt
```

La taille de 10240 octets pour l'archive `premiers.tar` s'explique par le format du fichier tar [tar format] qui utilise des blocs de taille fixe constitués en réalité de 20 petits blocs d'une taille de 512 octets chacun, soit 10240 octets. Les blocs non utilisés sont complétés avec des zéros, ce qui explique les excellents taux de compression pour les types gzip (69 fois) et bzip2 (72 fois) obtenus dans cet exemple.

La fonction `add`, comme la fonction `write`, pour les archives zip, permet d'ajouter des répertoires entiers de manière récursive, c'est-à-dire que tous les fichiers et sous-répertoires seront ajoutés à l'archive (y compris les fichiers cachés). En effet, si j'avais écrit `t.add('/home/dup')` dans l'exemple précédent, l'archive `premiers.tar` aurait contenu l'ensemble de mon répertoire personnel. Pour éviter le comportement récursif (par défaut), il est possible d'ajouter le paramètre `recursive=False` de la manière suivante : `add('/home/dup', recursive=False)`.

La documentation indique qu'il n'est pas possible d'ajouter des fichiers dans une archive tar pré-existante et compressée (que ce soit par gzip ou bzip2). En revanche, il est tout à fait possible d'ajouter des fichiers dans une archive pré-existante, mais non compressée toujours grâce à la fonction `add`.

4.4 Ajouter ses propres données dans les fichiers d'une archive tar

Dans le module `tarfile` il n'existe pas de fonction équivalente à `writestr` du module `zipfile`. Pour écrire dans un fichier d'une archive tar, il convient d'utiliser la structure `tarinfo` et la fonction `addfile`. La dernière remarque du paragraphe précédent implique soit de créer une archive à partir de zéro, soit d'utiliser une archive non compressée (que l'on pourra éventuellement compresser par la suite – cf. chapitre 2). Dans l'exemple suivant, j'ai choisi de créer une archive compressée de type bzip2. La méthode présentée peut paraître compliquée, mais elle montre la capacité de la fonction `addfile` à travailler avec des flux. Il peut s'agir de n'importe quel flux pourvu que l'interface d'utilisation de ce flux soit semblable à celle des fichiers. Ici, j'utilise le module `io` qui implémente ce type d'interface : `io.BytesIO`.

```
#!/usr/bin/python3
# -*- coding: utf8 -*-

import tarfile
import io

premiers = b'1\n2\n3\n5\n7\n11\n13\n'
periodes = b'1\n1\n2\n2\n4\n2\n'

tarbz2 = tarfile.open('nombres.tar.bz2', 'w:bz2')

info = tarfile.TarInfo('premiers.txt')
info.size = len(premiers)

tarbz2.addfile(info, io.BytesIO(premiers))

tarbz2.close()
```

Note

À partir de la version 3 de Python, le module `io` remplace les modules `StingIO` et `cStringIO`.

Conclusion

Cet article n'est qu'un survol rapide des possibilités offertes par les modules gzip, bz2, zipfile et tarfile. Toutefois, nous n'avons maintenant plus d'excuse pour ne pas compresser nos fichiers lorsque nous les créons. Cela économise de la place et, avec les processeurs actuels, ne grève pas vraiment les performances.

De même, plutôt que de générer des dizaines de fichiers les uns à côté des autres, nous devrions les générer dans une seule et même archive ! Les systèmes de fichiers seraient alors soulagés d'autant de fichiers !

La seule vraie difficulté dans tout cela est bien celle de choisir le format de compression et/ou d'archive en fonction de leurs fonctionnalités et de leurs limites !! ■

Références

[xz] : http://bugs.python.org/issue6715

[magic] : https://github.com/ahupp/python-magic

[with] : http://docs.python.org/dev/library/gzip.html

[argparse] : http://www.python.org/dev/peps/pep-0389

[zipfile] : http://docs.python.org/library/zipfile.html

[zipinfo] : http://docs.python.org/py3k/library/zipfile.html?highlight=zipfile#zipinfo-objects

[heraia] : http://heraia.tuxfamily.org/

[doublon] : http://bugs.python.org/issue2824

[tarfile] : http://docs.python.org/release/3.1.3/library/tarfile.html

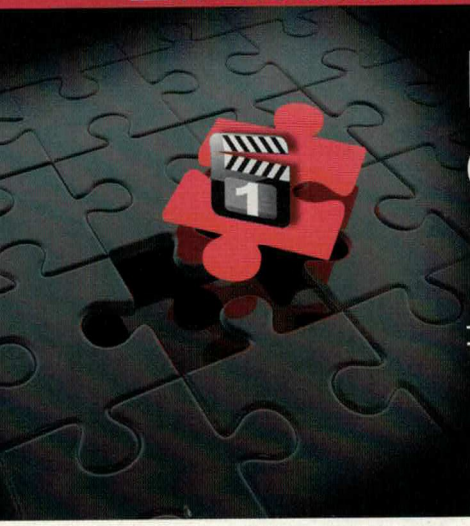
[tarinfo] : http://docs.python.org/release/3.1.3/library/tarfile.html#tarfile.TarInfo

[tar format] : http://fr.wikipedia.org/wiki/Tar_(informatique)

Et les exemples de Doug Hellmann :

http://www.doughellmann.com/PyMOTW/zipfile/

http://www.doughellmann.com/PyMOTW/tarfile/



LA PROGRAMMATION ORIENTÉE OBJET EN PYTHON – ÉPISODE 1 : LA THÉORIE

Tristan Colombo

En Python, tout est objet : lorsque vous manipulez une simple variable, il s'agit en fait d'un objet. Ce langage intègre donc naturellement la possibilité de développer une architecture orientée objet. Dans cette première partie, je vous présente la théorie de la programmation orientée objet vue depuis Python.

Le langage Python se distingue des autres langages par sa philosophie un peu particulière privilégiant un développement rapide et peu complexe grâce à un haut niveau d'abstraction (voir la recommandation PEP 20 [1]). Cette philosophie se retrouve dans la façon dont la programmation orientée objet – POO en abrégé – est traitée. Dans cet article, nous allons donc voir la théorie de la POO mais vue depuis Python. En effet, certaines notions n'ont pas été implémentées en Python par souci de simplification du langage et, certaines notions, communes à tous les langages orientés objet, auront une syntaxe et un fonctionnement très particuliers. Cet article étant dédié à la théorie et un autre étant dédié à la pratique de la POO en Python, pour une meilleur lisibilité, j'ai utilisé le même plan dans les deux articles. Vous pourrez ainsi retrouver simplement la théorie depuis un exemple de code Python et inversement.

1 Introduction

La POO est une méthode de programmation qui permet de suivre le logiciel tout au long du cycle de sa vie (expression des besoins, analyse, conception, tests, etc.). Elle a été créée pour permettre de gérer de gros projets impliquant des dizaines de développeurs : pour optimiser le développement, il faut que des équipes puissent travailler sur des petits morceaux de code indépendants, qu'en cas de modification du code d'une équipe il n'y ait pas d'impact sur le code d'une autre équipe, que chaque équipe soit capable de lire le code produit par une autre équipe, etc. Cette méthode introduit donc de la rigueur dans le développement... et dans un langage interprété, ceci est fondamental pour ne pas aboutir à un code incompréhensible.

Si cette méthode cible plus particulièrement le développement en équipes, rien n'empêche de l'utiliser dans le cadre d'un projet personnel... Par contre, pour un petit développement (script effectuant quelques calculs par exemple), il serait totalement aberrant d'utiliser la POO : la POO ne remplace pas la programmation impérative ! Dans l'article d'introduction de ce hors-série, je vous ai montré l'exemple du Java qui imposait la création d'une classe pour afficher un simple « Hello World »... Il serait dommage de retomber dans le travers de Java.

En POO, on utilise la notion d'« objet ». Un objet, c'est ce que nous voyons dans notre environnement : un clavier d'ordinateur, un moniteur, une table, une chaise, etc. Chaque objet peut être défini plus ou moins précisément : pour un clavier, par exemple, je peux indiquer sa couleur, son nombre de touches, sa configuration azerty ou qwerty, etc. Certains objets sont composés d'une agrégation d'autres objets : pour créer une voiture, il me faudra utiliser quatre objets « roue », un objet « moteur », etc. En faisant communiquer les objets entre eux on obtient ensuite une entité autonome vue de l'extérieur (je n'ai pas à savoir comment fonctionne précisément le moteur pour utiliser l'objet « voiture »). Ces mécanismes sont la base de la POO. Voyons maintenant en détails comment ils sont définis et quel est le vocabulaire employé.

2 Classes

L'Homme a toujours eu besoin de classer et les objets n'échappent pas à cette règle. Pour un même « type » d'objet (on parlera alors de « classe »), je peux avoir plusieurs objets de même fonction mais d'apparence différente. En reprenant l'exemple du clavier : un clavier noir azerty à 102 touches est différent d'un clavier blanc qwerty à 105 touches... Pourtant, ce sont deux claviers et ils ont la même fonction. Ils appartiennent à un ensemble qui définit de manière générique ce que doit être un clavier : la classe Clavier.

Pour définir cette classe, il va falloir préciser les éléments qui vont permettre de représenter l'objet : son « apparence » et ses « fonctions ».

2.1. Les attributs

L' « apparence » d'un objet, c'est tout ce qui va permettre de le distinguer d'un autre objet de la même classe. Un clavier blanc est différent d'un clavier noir : la couleur est un élément permettant de différencier

deux claviers, on dira qu'il s'agit d'un attribut de la classe Clavier. Pour pouvoir représenter un clavier noir azerty à 102 touches nous aurons besoin de trois attributs : la couleur, la disposition des touches et le nombre de touches.

D'un point de vue informatique, en simplifiant, on peut voir les classes et les attributs comme une structure de données. On pourrait représenter la classe Voiture à l'aide d'un dictionnaire :

```
class_voiture = {
  'couleur' : None,
  'disposition' : None,
  'nb_touches' : None
}
```

2.2. Les méthodes

Le « fonctionnement » d'un objet est décrit par un ensemble de fonctions qui peuvent lui être appliquées. Par exemple, pour un clavier, on peut appuyer sur une touche donnée : il doit donc y avoir une fonction `appuyer_sur_touche()` dans la classe Clavier. Une telle fonction est appelée « méthode ».

Il existe une méthode spéciale, présente dans toutes les classes, le constructeur. Cette méthode, lorsqu'elle est appelée, permet de créer un objet tel que défini dans la classe. La classe peut être vue comme un plan... en tant que tel on ne peut rien en faire (Fig. 1). En appelant le constructeur, on regroupe l'ensemble des matériaux dont on a besoin et on construit l'objet qui a été décrit par le plan. On parle alors d'une instance de l'objet : le clavier noir est une instance de la classe Clavier, de même que le clavier blanc.

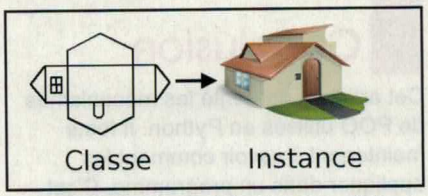


Fig. 1 : Le « plan » – la classe – et la « réalisation » – l'instance

2.3. Visibilité

Nous avons commencé à parler de la visibilité tout à l'heure avec l'exemple de la voiture qui était composée d'une agrégation d'objets : j'ai indiqué que pour utiliser un tel objet je n'étais pas obligé de connaître le fonctionnement des objets qui le composent (moteur, bougies, cardan, etc.). La POO permet d'attribuer une visibilité aux attributs et aux méthodes d'une classe : est-ce que le développeur peut utiliser cet attribut ou cette méthode en travaillant sur une instance de l'objet ? Si la réponse est oui, alors nous utiliserons la visibilité publique. Dans le cas contraire, nous devrons utiliser la visibilité privée : l'attribut ou la méthode cible ne pourront être utilisés qu'à l'intérieur de la classe.

Il s'agit d'un mécanisme de protection des données : le développeur à qui nous avons fourni la classe Voiture n'est pas un spécialiste des voitures : nous lui donnons le droit de voir les messages du tableau de bord, mais tout ce qui se trouve sous le capot est une grosse boîte noire dont il n'a pas à connaître le fonctionnement. De la visibilité découle une notion importante (voire fondamentale) de la programmation orientée objet et du génie logiciel : l'encapsulation.

L'encapsulation consiste à masquer le plus possible les détails d'implémentation et le fonctionnement interne des objets. Cette dissimulation permet de masquer la complexité de la classe et permet également de préserver l'intégrité des attributs. En effet, si la valeur d'un attribut peut être modifiée depuis n'importe quel endroit, il n'y a plus aucun contrôle sur sa valeur ! Si on interdit l'accès direct et que la modification passe obligatoirement par une méthode, on assure un contrôle sur la valeur qui sera stockée. En général, pour masquer la complexité d'une classe et la rendre modifiable ultérieurement, tous les attributs doivent être déclarés privés dans toutes les classes. Mais si tous les attributs sont privés, comment y accéder tout de même ? On peut avoir besoin d'afficher ou de modifier une valeur depuis l'extérieur de la classe... On utilise alors des méthodes publiques appelées « méthodes d'accès » et « méthodes de modification ».

Tout ce mécanisme d'encapsulation peut paraître lourd et inutile au premier abord, mais il s'agit d'un élément fondamental de la programmation orientée objet, car il participe à la modularité du code (chaque objet ne dépend pas du code interne d'un autre objet).

Pour aller encore plus loin dans le principe d'encapsulation, on peut décider de n'utiliser que des méthodes d'accès pour accéder aux attributs d'une classe, même à l'intérieur de celle-ci. Ainsi, si une modification des attributs (de ce qu'ils représentent) intervient, il suffira de récrire les méthodes d'accès et non pas tout le code de la classe.

2.4. Attributs et méthodes de classe

Jusqu'à présent, nous avons supposé que tous les attributs et les méthodes d'un objet se rapportaient à une instance particulière. En fait, il est possible de définir des attributs qui seront « partagés » par tous les objets d'une même classe : il existe alors un seul exemplaire de cet attribut qui peut être modifié par n'importe quelle instance de l'objet. Un attribut ayant cette propriété est un attribut statique ou attribut de classe. Prenons par exemple la classe Clavier à laquelle nous allons ajouter un attribut de classe `cpt` qui servira à compter le nombre d'instances créées (à chaque appel du constructeur de la classe Clavier on incrémente l'attribut `cpt`). La figure 2 illustre le comportement d'une telle classe.

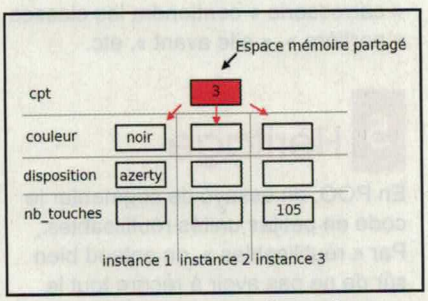


Fig. 2 : Comportement d'une classe possédant un attribut de classe

Les méthodes peuvent être également déclarées comme des méthodes statiques ou méthodes de classe. Une méthode de classe est une méthode qui s'exécute en

référence à sa classe et non plus en référence à une instance particulière (il ne faut pas créer d'objet pour l'exécuter). Cette méthode ne peut désigner et modifier que des attributs de classe.

3 Composition de classes

Un objet peut être formé à partir d'autres objets (toujours l'exemple de la voiture). À ce moment-là, on utilise des objets en tant qu'attributs et on parle de « composition de classes » : une ou des classes sont utilisées pour créer une autre classe.

4 Organiser les objets en paquetages

Les fichiers de classes peuvent être organisés dans une arborescence grâce au mécanisme des espaces de nom : à l'aide d'une structure de fichiers et d'une syntaxe particulière, on peut définir des ensembles de classes à l'intérieur de répertoires spécifiques (et de sous-répertoires, sous-sous-répertoires, etc.). Les paquetages permettent d'introduire une classification au-dessus des classes. Pour l'objet Voiture, le paquetage « éclairage » contiendra les classes « phare », « feu de position », « clignotant », ..., le paquetage « carrosserie » contiendra les classes « portière », « aile avant », etc.

5 Héritage

En POO, on essaye de segmenter le code en petites unités réutilisables. Par « réutilisables », on entend bien sûr de ne pas avoir à récrire tout le code. En programmation impérative, il est très simple d'écrire par exemple une bibliothèque de fonctions et de les appeler ensuite dans diverses autres fonctions sans avoir à les récrire. Par contre, si vous désirez utiliser une fonction `A()` en modifiant quelque peu son comportement, vous devrez écrire une fonction `A'()`, copie de `A()`, dont le contenu ne variera que de quelques lignes. C'est ici qu'intervient la notion d'héritage : on souhaite pouvoir réutiliser un objet en y apportant quelques modifications de comportement (au niveau des attributs et/ou des méthodes). On crée ainsi une nouvelle classe à partir d'une classe qui existe déjà en la complétant pour qu'elle permette de créer des objets plus spécifiques (on part ainsi du plus générique et on « spécialise » l'objet de départ au fur et à mesure).

Si l'on définit une classe Mammifère, comme un Chien est un mammifère, nous pourrons dire que la classe Chien hérite de Mammifère, que Chien est la classe fille de Mammifère, et que Mammifère est la classe mère de Chien.

5.1. Accessibilité

Les attributs et méthodes ayant une visibilité privée ne sont pas hérités : ils font partie de la « boîte noire » de la classe mère.

5.2. Définition du constructeur

Le constructeur d'une classe fille doit faire appel au constructeur de sa classe mère pour initialiser les attributs dont la classe fille hérite. Des attributs spécifiques peuvent être ajoutés par la suite.

5.3. Héritage multiple

L'héritage multiple fonctionne sur le même modèle que l'héritage simple, mais ici, la classe fille n'a plus une seule classe mère mais plusieurs. Par exemple, avec les classes Mammifère et Carnivore, la classe Lion hérite à la fois de la classe Mammifère et de la classe Carnivore.

L'héritage multiple est à manipuler avec précautions :

- si plusieurs classes dont on hérite implémentent une méthode de même nom, il faudra être capable d'indiquer quelle méthode doit être exécutée.
- un héritage en diamant est possible : si une classe B et une classe C héritent d'une même classe mère A et qu'une classe D hérite de B et de C (Fig. 3), alors D hérite deux fois de la même classe A.

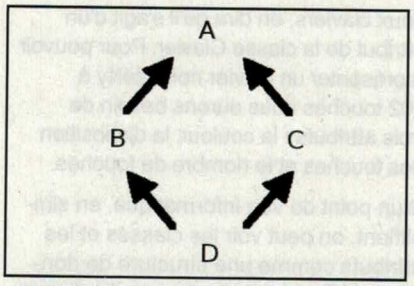


Fig. 3 : Héritage multiple en diamant

Python sait gérer ce genre de configurations, mais il faudra être très vigilant au niveau de la modélisation pour être certain de manipuler les bonnes données et que le cas se prêtait bien à un héritage multiple.

6 Les exceptions

Une exception est un cas particulier d'information non standard que doit traiter une méthode et qui ne peut pas être traité par la méthode ou nécessite un traitement très spécifique. Les raisons de ce traitement peuvent être diverses : erreur de programmation, erreur d'entrée/sortie, valeur non autorisée passée en paramètre, etc. Il existe des exceptions standards (division par zéro, erreur d'entrée/sortie, etc.) et des exceptions qui peuvent être définies et lancées (on dit « levées ») par le développeur.

Pour traiter une exception, on doit commencer par scruter un bloc de code à la recherche d'une exception. Si une exception est détectée dans ce bloc, alors un bloc de code spécifique sera exécuté.

Conclusion

Cet article a présenté les mécanismes de POO utilisés en Python. Il reste maintenant à savoir comment les appliquer dans un programme. C'est le sujet de la suite de cet article : « La programmation orientée objet en Python – épisode 2 : la pratique ». ■

Référence

[1] Recommandation PEP 20 : http://www.python.org/dev/peps/pep-0020/

LA PROGRAMMATION ORIENTÉE OBJET EN PYTHON – ÉPISODE 2 : LA PRATIQUE

Tristan Colombo

Après avoir vu (rapidement !) la théorie de la POO, passons à la mise en pratique. Dans la première partie de cet article nous ne nous sommes attachés qu'à la compréhension d'un mécanisme général. Python est un langage très simple, mais également très complexe lorsque l'on veut vraiment savoir ce que l'on fait. J'ai l'habitude de dire que c'est un langage de « détails » : si vous voulez vraiment programmer en Python, il faudra avoir vu au moins une fois ces « détails » de syntaxe pour ne pas passer des heures par la suite à rechercher l'origine d'un bug... Qui n'en est finalement pas un, mais plutôt une incompréhension du langage. La POO en Python n'échappe pas à cette règle : prenez du temps pour bien comprendre ce qui se passe en machine. En plus, Python fournit un outil formidable pour ce genre de tests : l'interpréteur interactif !

1 Introduction

Cet article n'étant basé que sur la syntaxe, certaines notions vues dans la partie théorique seront traitées rapidement : soit parce que l'implémentation se fait très simplement, soit parce que d'autres notions (vues par la suite) sont nécessaires pour une mise en pratique. Dans ce dernier cas, vous trouverez un exemple d'application dans les sections suivantes.

2 Classes

La définition d'une classe se fait à l'aide du mot-clé `class`. Contrairement à Java où une classe doit porter le même nom que le fichier dans lequel elle se trouve, Python n'impose aucune règle de nomenclature et vous pouvez, si vous le souhaitez, écrire toutes vos classes au sein d'un même fichier. Pour plus de lisibilité, je préfère utiliser les normes Java : le nom des classes est donné en « camel case » (nom commençant par une majuscule et séparations entre les mots indiquées par des majuscules), le nom du fichier est le même que celui de la classe qu'il porte, et le plus souvent possible, une seule classe par fichier.

Lors de la définition de la classe, on doit indiquer en paramètre l'héritage. Nous verrons cela plus en détail dans la section dédiée à l'héritage, mais comme tous les objets héritent d'une classe mère « originelle » nommée `object`, il va falloir l'indiquer. La syntaxe est donc la suivante :

```
class MaClasse(object):
    """
        Définition de la classe MaClasse
    """
    ...
```

2.1. Les attributs

Python étant un langage à typage dynamique, les variables n'ont pas à être déclarées... Pourtant, il faut bien indiquer quels sont les attributs de la classe, même si aucune valeur ne leur est affectée... Les attributs seront déclarés dans le constructeur de l'objet et la valeur `None` devra leur être affectée.

Les attributs étant attachés à l'objet courant, on les distinguera des variables locales en les préfixant par `self`. Ainsi, `self.couleur` désigne l'attribut « couleur » d'une classe.

2.2. Les méthodes

Les méthodes sont décrites comme des fonctions où le premier argument est obligatoirement `self` : cet argument fait référence à l'instance de l'objet sur lequel la méthode a été appelée. Lors de l'appel d'une méthode sur un objet, le premier paramètre sera transmis automatiquement. Donc, pour une méthode définie avec `n` arguments, l'appel se fera avec `n-1` arguments explicites.

Une méthode appelée depuis sa classe de définition devra être précédée de `self` pour indiquer qu'elle est utilisée sur l'objet courant.

Le constructeur est défini en utilisant un nom de méthode spécial et réservé : `__init__`. C'est dans le constructeur que nous définissons les attributs. Voici un exemple de classe :

```
class Clavier(object):
  def __init__(self, c, dispo, n):
    self.couleur = c
    self.disposition = dispo
    self.nb_touches = n
    self.marque = None

  def appuyer_sur_touche(self, touche):
    print "L'utilisateur a appuyé sur la touche '%s'" % touche

  def relacher_touche(self, touche):
    self.appuyer_sur_touche(touche)
    print "L'utilisateur a relâché la touche '%s'" % touche
```

La création d'une instance d'un objet se fait en appelant simplement le constructeur. Attention : il ne faut pas appeler directement la méthode `__init__`, l'appel se fait en précisant le nom de la classe qui constituera une sorte d'alias vers le constructeur.

```
monClavier = Clavier('noir', 'azerty', '105')
```

Pour appeler une méthode sur une instance d'un objet nous utiliserons l'opérateur « `.` ». C'est en fait cet opérateur qui transmet l'instance à la méthode sous la forme de `self`. Ainsi, pour appeler la méthode `appuyer_sur_touche()` sur notre instance `monClavier`, il faudra exécuter :

```
monClavier.appuyer_sur_touche('A')
```

2.3. Visibilité

En Python, la visibilité est gérée par une simple convention d'écriture. Par défaut, les attributs et méthodes sont publics. Pour les rendre privés, il faut préfixer leur nom par `__`. Voici une petite expérience menée sur deux attributs, l'un public et l'autre privé :

```
class Visibilite(object):
  def __init__(self, v1, v2):
    self.public = v1
    self.__prive = v2

  def affiche(self):
    print "Valeur de public : %d" % self.public
    print "Valeur de __prive : %d" % self.__prive

  def __methode_prive(self):
    print "Je suis une méthode privée !!"

obj = Visibilite(1, 1)
```

`obj` est une instance de `Visibilite` et les attributs `public` et `__prive` ont la même valeur `1`. La méthode `affiche()` permet d'accéder aux valeurs des deux attributs de manière interne. Effectuons quelques tests depuis l'interpréteur interactif :

```
>>> obj = Visibilite(1, 1)
>>> obj.affiche()
Valeur de public : 1
Valeur de __prive : 1
```

Après création de l'objet, les valeurs sont bien identiques.

```
>>> print obj.public
1
>>> print obj.__prive
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Visibilite' object has no attribute '__prive'
```

Si nous voulons afficher les valeurs des attributs, l'accès à l'attribut privé provoque une exception au message étrange : « L'objet 'Visibilite' n'a pas d'attribut '__prive' »... Voilà qui est bizarre, puisque lors de l'affichage précédent nous avons bien obtenu la valeur `1`.

```
>>> obj.public = 5
>>> obj.affiche()
Valeur de public : 5
Valeur de __prive : 1
```

Nous pouvons modifier la valeur de l'attribut déclaré avec une visibilité publique.

```
>>> obj.__prive = 5
```

Étrangement, la modification de l'attribut privé ne déclenche pas d'erreur... Aurions-nous pu modifier cette valeur ? Vérifions...

```
>>> obj.affiche()
Valeur de public : 5
Valeur de __prive : 1
```

Non, tout va bien, l'attribut privé n'a pas été modifié ! Mais que s'est-il passé alors ? Essayons à nouveau d'afficher la valeur de `obj.__prive` :

```
>>> print obj.__prive
5
```

Là ça devient vraiment confus : il y a quelques secondes, nous ne pouvions pas afficher la valeur de `obj.__prive` et maintenant c'est possible... Donc, si j'appelle la méthode `affiche()` je devrais avoir la même valeur `5` pour les deux attributs :

```
>>> obj.affiche()
Valeur de public : 5
Valeur de __prive : 1
```

Encore raté... Il doit y avoir une explication...

En fait, la convention d'écriture en `__` cache l'attribut pour l'extérieur de la classe. Il est donc normal de ne pas y avoir accès depuis une instance. Par contre, il est possible de créer des attributs à la volée depuis une instance et à l'extérieur d'une classe les caractères `__` n'ont plus la même signification, ils ne cachent plus l'attribut. Dans notre exemple, nous avions en fait trois attributs :

- l'attribut public `public` ;
- l'attribut privé `__private` ;
- l'attribut public `__private` (même nom que le précédent mais différent).

Il faut donc être très rigoureux dans la manipulation des visibilités en Python ! Si vous implémentez l'encapsulation, vous ne devriez pas avoir de problème.

Avec l'encapsulation, nous allons devoir écrire des méthodes d'accès et de modification pour les attributs qui seront tous déclarés privés. Prenons l'exemple d'une classe `Point` contenant naturellement deux attributs (pour un point en 2D) : `x` et `y`. Nous allons déclarer deux méthodes d'accès `get_x` et `get_y` et deux méthodes de modification `set_x` et `set_y`. Ces méthodes devront être publiques pour pouvoir être appelées depuis l'extérieur de la fonction :

```
class Point(object):
    def get_x(self):
        return self.__x
    def set_x(self, x):
        self.__x = x

    def get_y(self):
        return self.__y
    def set_y(self, y):
        self.__y = y

    def __init__(self, x, y):
        self.set_x(x)
        self.set_y(y)
```

Si nous testons cette classe, nous obtenons bien l'effet désiré :

```
>>> p = Point(2, 3)
>>> print p.get_x()
2
>>> p.set_x(10)
>>> print p.get_x()
10
```

Toutefois, cette écriture est un peu lourde. Python propose deux écritures (même code interne) permettant d'enfouir les accesseurs et modifieurs et de les rendre invisibles pour le développeur extérieur :

- la fonction `property` : cette fonction va créer des alias vers les méthodes d'accès et de modification. Cette fonction sera appelée après chaque définition de paire accesseur/modifieur en lui indiquant quelle est la méthode employée pour accéder à l'attribut et quelle est la méthode pour modifier l'attribut :

```
    def __get_x(self):
        return self.__x
    def __set_x(self, x):
        self.__x = x
    x = property(__get_x, __set_x)
```

En passant les méthodes `get_x` et `set_x` en visibilité privée et en créant une propriété nommée `x` utilisant ces méthodes, nous avons « défini » un attribut `x` accessible pour le développeur qui ne travaille pas directement sur le code de la classe.

```
>>> p = Point(2, 3)
>>> print p.x # Fait en fait appel à
p.__get_x()
2
>>> p.x = 10 # Fait en  fait appel à
p.__set_x(10)
>>> print p.x
10
```

- les décorateurs `property` et `setter` : il s'agit de fonctions spéciales (leur appel commence par un `@`) qui s'exécutent « par-dessus » la fonction qu'ils précèdent (ou « décorent »). L'utilisation des décorateurs ne modifie ici que l'écriture du code, le résultat étant le même.

```
    @property
    def x(self):
        return self.__x
    @x.setter
    def x(self, x):
        self.__x = x
```

2.4. Attributs et méthodes de classe

Les attributs de classe ou attributs statiques sont déclarés en dehors de toute méthode (après la ligne `class`...). Comme ils ne sont pas liés à une instance particulière, leur nom n'est pas précédé de `self`. Pour les utiliser dans des méthodes de la classe il faudra les préfixer du nom de la classe ou de `self.__class__` (qui renvoie le nom de la classe), de manière à les différencier des variables locales.

Les méthodes de classe ou méthodes statiques n'ont accès qu'aux attributs statiques. En effet, ces méthodes sont appelées sans instance de la classe et du coup, elles ne comportent pas l'argument `self` dans la liste de leurs paramètres. Pour les déclarer, il faudra utiliser le décorateur `@staticmethod`. Pour les utiliser, il faudra les préfixer du nom de la classe.

À titre d'exemple, définissons une classe `Compteur` qui comptabilisera seulement le nombre d'instances de cette classe.

```
class Compteur(object):
    count = 0

    def __init__(self):
        Compteur.count +=1

    @staticmethod
    def affiche():
        print Compteur.count
```

Nous pouvons appeler la méthode `affiche()` sans même avoir créé d'instance de `Compteur` :

```
>>> Compteur.affiche()
0
```

Nous avons accès à l'attribut statique `count` (pour lui donner la visibilité privée, il faudra le renommer en `__count`) :

```
>>> print Compteur.count
0
```

Et lorsque nous créons une instance de `Compteur`, nous incrémentons correctement notre compteur :

```
>>> c = Compteur()
>>> Compteur.affiche()
1
```

3 Composition de classes

La composition de classes se réalise très simplement en deux étapes :

1. Charger la ou les classe(s) à utiliser.
2. Utiliser les classes.

En réutilisant notre classe `Point`, nous allons définir une classe `Segment`, composée... de deux points :

```
from Point import *

class Segment(object):
```

```
  def __init__(self, p1_x, p1_y, p2_x, p2_y):
    self.__p1 = Point(p1_x, p1_y)
    self.__p2 = Point(p2_x, p2_y)

  def __str__(self):
    return "(" + str(self.__p1.x) + "," +
str(self.__p1.y) + ") - (" + str(self.__p2.x) +
"," + str(self.__p2.y) + ")"
```

Un `Segment` est donc un objet comportant deux attributs privés `p1` et `p2` qui sont des `Points`. La méthode `__str__` est une méthode spéciale : elle renvoie obligatoirement une chaîne de caractères qui est utilisée lorsque l'objet apparaît dans un contexte de chaîne de caractères (par exemple avec un `print`). Voici un exemple d'utilisation :

```
>>> s = Segment(2, 5, 3, 4)
>>> print s
(2, 5) - (3, 4)
```

4 Organiser les objets en paquetages

Pour regrouper logiquement des objets, vous pouvez créer des paquetages en plaçant vos fichiers sources dans des répertoires contenant un fichier spécial `__init__.py`. La présence de ce fichier « transforme » le répertoire en paquetage. Il faut noter que la plupart du temps le fichier `__init__.py` est vide, mais si vous avez des valeurs à initialiser lors du chargement du paquetage, vous pouvez ajouter des instructions dans ce fichier.

Voici un exemple d'organisation de fichiers en vue de créer des paquetages :

```
.
├── carrosserie
│   ├── __init__.py
│   ├── Aile.py
│   ├── Portiere.py
└── eclairage
    ├── __init__.py

    ├── exterieur
    │   ├── __init__.py
    │   ├── Phare.py
    │   ├── Stop.py
    └── interieur
        ├── __init__.py
        ├── Plafonnier.py
```

Pour réaliser des imports depuis ces paquetages (vous pouvez utiliser les paquetages pour structurer vos modules, même si vous n'utilisez pas la POO), il faudra indiquer le chemin complet vers les fichiers Python en séparant les répertoires par des points. Exemple :

```
>>> from carrosserie.Aile import *
>>> from eclairage.exterieur.Phare import *
```

5 Héritage

L'héritage est spécifié dès la création de la classe entre parenthèses : `class Nom(Classe_mère)`. Les méthodes héritées de la classe mère peuvent être utilisées en préfixant leur nom par le nom de la classe mère : `Classe_mère.nomMéthode()`. Par défaut, quand une classe n'hérite d'aucune autre classe, elle hérite de la classe `object`, « super classe mère » définissant le prototype d'un objet.

L'ensemble des méthodes disponibles pour un objet peut être obtenu par la fonction `dir()`. Par exemple pour une chaîne de caractères :

```
>>> dir(str)
['__add__', '__class__', '__contains__',
'__delattr__', '__doc__', '__eq__', '__
format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__',
'__gt__', '__hash__', '__init__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '_formatter_
field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find',
'format', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

En dehors de méthodes déjà vues telles que `split`, `strip`, etc., nous pouvons voir des noms de méthodes encadrés par `__`. Ces méthodes sont héritées de la classe `object` et sont donc communes à tous les objets par héritage. À quoi servent-elles ? Eh bien, par exemple, à définir le comportement de l'objet lorsqu'on lui applique un opérateur : `__add__` pour l'addition, `__mul__` pour la multiplication, etc. Par exemple, en utilisant l'héritage de `object` sur la classe `Point`, nous allons définir l'addition de deux points comme étant un point dont les coordonnées sont la somme des coordonnées des points participant à l'opération :

```
class Point(object):

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __add__(self, p2):
    return Point(self.x+p2.x, self.
y+p2.y)

  def __str__(self):
    print '(%d,%d)' % (self.x, self.y)
```

Grâce à ce code, l'« addition » de deux points est grandement simplifiée :

```
>>> p1 = Point(1, 2)
>>> p2 = Point(3, 4)
>>> p3 = p1 + p2
>>> print p3
(4,6)
```

5.1. Accessibilité

Les attributs et méthodes ayant une visibilité privée ne sont pas hérités. Par exemple, si `Classe_mère` possède un attribut privé `__x` sans méthode d'accès publique, alors la classe fille n'y aura pas accès.

5.2. Définition du constructeur

L'appel à `Classe_mère.__init__(self, ...)` permet d'appeler le constructeur de la classe mère à l'intérieur du constructeur de la classe fille et ainsi d'initialiser correctement les attributs hérités.

Voici un exemple complet d'héritage. Nous allons définir une classe `Polygone` qui possède un seul attribut : une liste de `Points` (classe définie précédemment). La classe

Triangle va hériter de Polygone, mais son constructeur ne permettra de saisir que trois Points.

```
01: from Point import *
02:
03: class Polygone(object):
04:
05:   def __get_pts(self):
06:     return self.__pts
07:   def __set_pts(self, liste):
08:     self.__pts = liste
09:   pts = property(__get_pts, __set_pts)
10:
11:   def __init__(self, *points):
12:     self.__set_pts(points)
13:
14:   def display(self):
15:     for i in self.__get_pts():
16:       print i
```

Les lignes 5 à 9 définissent l'accesseur et le modifieur pour l'attribut pts. Le constructeur est donné en lignes 11 et 12. On peut noter l'utilisation d'un karg (*points) permettant de ne pas limiter le nombre de paramètres et de récupérer ces derniers sous forme de liste. La méthode des lignes 14 à 16 parcourt la liste des points et les affiche en utilisant la méthode __str__ associée à l'objet Point (ici i est un Point). Nous pouvons maintenant écrire la classe Triangle :

```
01: from Polygone import *
02:
03: class Triangle(Polygone):
04:
05:   def __init__(self, p1, p2, p3):
06:     Polygone.__init__(self, p1, p2, p3)
07:
08:   def display(self) :
09:     print 'Redéfinition de display() dans
Triangle'
10:     Polygone.display()
11:
12: t = Triangle(Point(1, 5), Point(3, 4),
13:              Point(5, 6))
14: t.display()
```

On voit bien que cette classe hérite de Polygone (ligne 3) et que le constructeur de Polygone est appelé en lui passant seulement trois points en ligne 6. Dans les lignes 8 à 10, nous redéfinissons la méthode display(), puis nous appelons cette même méthode telle qu'elle était définie dans Polygone (un peu comme si nous faisions un copier/coller de son code).

5.3. Héritage multiple

La syntaxe de l'héritage multiple sera la même que celle de l'héritage simple, à la différence près que plusieurs classes devront être indiquées en héritage lors de la définition de la classe : class Nom(Classe_mère_1, Classe_mère_2, ...). Pour accéder aux méthodes de chacune des classes mère, il faudra les préfixer par le bon nom...

6 Les exceptions

Pour pouvoir récupérer une exception et la traiter, il faut que la commande ayant levé l'exception se trouve dans un bloc d'« analyse » défini par try. La partie traitement se trouve dans un bloc except et il est possible d'ajouter un bloc finally indiquant du code à exécuter que l'exception soit levée ou non.

```
try:
  # commandes
except:
  # traitement des erreurs
finally:
  # commandes à exécuter dans tous les
cas
```

Dans le modèle précédent, toutes les exceptions sont traitées dans le même bloc. Pour effectuer un traitement plus fin, il faut nommer les exceptions pour chaque bloc except :

```
try:
  # commandes
except (IOError, ZeroDivision):
  # traitement des erreurs IOError,
ZeroDivision
except TypeError:
  # traitement des erreurs TypeError
except:
  # traitement des erreurs restantes
```

Ces commandes permettent de récupérer les exceptions standards. Si vous souhaitez lever une exception, il vous faudra créer un objet Exception :

```
raise Exception("Mon message d'erreur")
```

Pour aller plus loin et transmettre éventuellement des paramètres lors de la levée de l'exception, vous pouvez personnaliser les exceptions en utilisant l'héritage sur la classe Exception :

```
class MonException(Exception):

  def __init__(self, valeur_1,
valeur_2):
    self.__valeur_1 = valeur_1
    self.__valeur_2 = valeur_2

  def affiche_erreur(self):
    print "Déclenchement MonException
avec les valeurs", val.__valeur_1, " et
code : ", val.__valeur_2
```

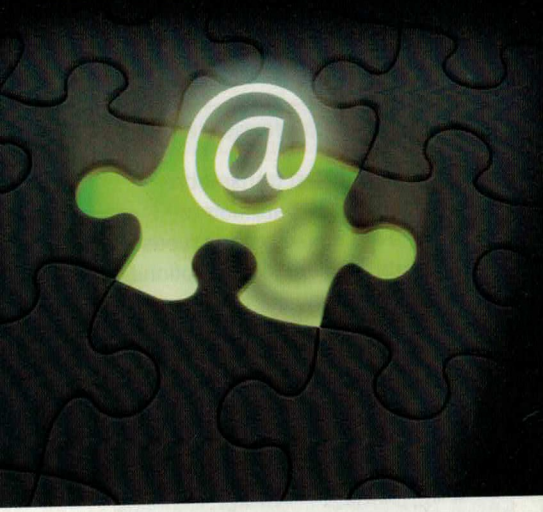
Pour lever une exception MonException nous ferons appel à raise :

```
try:
  raise MonException("Mon message
d'erreur", 10)
except MonException, val:
  val.affiche_erreur()
```

La variable val est une instance de la classe MonException. On peut donc lui appliquer les méthodes définies dans la classe MonException.

Conclusion

L'implémentation de la POO en Python est assez particulière et j'espère par cet article vous avoir suffisamment détaillé les mécanismes de base. Pour ceux d'entre vous qui maîtrisent déjà un langage à typage statique tel que Java, la syntaxe de Python vous paraîtra sans doute farfelue et peu fiable. Mais la philosophie de Python n'est pas de considérer le développeur comme un garnement irresponsable. Et c'est même complètement le contraire, car comme le dit l'un des slogans de Python : « We're all consenting adults here » (« Nous sommes tous ici des adultes consentants »). C'est donc au développeur de savoir ce qu'il fait et de correctement documenter son travail... ■



RÉDIGER ET ENVOYER UN E-MAIL AVEC PYTHON 3.2

Carl Chenet

Justement qualifié par la devise « batteries included », le langage Python offre une large boîte à outils permettant de créer et d'envoyer facilement des e-mails. Dans cet article, nous présenterons le module email disponible dans la bibliothèque standard. Utiliser ce dernier vous permettra d'écrire des programmes complets d'envoi d'e-mail sans avoir à requérir à une bibliothèque externe, cette dernière entraînant inévitablement pour le programmeur les contraintes liées aux dépendances.

À travers une série d'exemples présentés en Python 3.2, la dernière version stable de Python, je vous proposerai différentes techniques pour préparer vos e-mails - qu'ils soient constitués de texte, de fichiers multimédias ou en HTML - avant de les envoyer via un serveur SMTP local ou distant.

1 Installer Python 3.2

Il est, en général et à juste titre, conseillé d'utiliser les paquets disponibles pour votre distribution GNU/Linux. Par exemple, sur votre système Debian (Wheezy ou Sid), il suffit de taper la commande suivante en tant qu'utilisateur root :

```
# apt-get install python3.2
```

Toutefois, si votre distribution ne propose pas un paquet pour Python 3.2, vous pouvez l'installer de la façon suivante à partir des sources :

```
$ wget http://www.python.org/ftp/python/3.2.2/
Python-3.2.2.tgz
$ tar zxvf Python-3.2.2.tgz
$ cd Python-3.2.2
$ ./configure
$ make
```

Vous avez maintenant dans votre répertoire courant l'exécutable `python`. Vous pouvez vous en assurer en lançant la commande suivante :

```
$ ./python
Python 3.2 (r32:88445, Feb 22 2011, 00:31:03)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

2 Composer un e-mail à partir d'un fichier texte et l'envoyer via un serveur SMTP distant

Afin d'envoyer à un correspondant un message grâce à un e-mail généré automatiquement en Python, nous commençons par écrire un message dans un fichier texte à partir d'un terminal :

```
$ echo 'Bien le bonjour aux lecteurs de Linux Pratique!' > emailbody
```

Nous écrivons maintenant dans le fichier `emailfromfile.py` les lignes suivantes qui vont constituer un programme à part entière :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# import pour manipuler le texte en format MIME
from email.mime.text import MIMEText

# nous ouvrons notre fichier texte en lecture
myfile = open('emailbody', 'r')
# conversion en format MIME
myemail = MIMEText(myfile.read())
# fermeture du fichier
myfile.close()

# le sujet de l'e-mail
myemail['Subject'] = 'Un e-mail contenant du texte'
# l'adresse de l'expéditeur
myemail['From'] = 'sender@sender.com'
# l'adresse du destinataire
myemail['To'] = 'recipient@recipient.com'
```

```
# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x emailfromfile.py
$ ./emailfromfile.py
```

Nous voyons qu'à travers la création de l'objet `myemail`, le fichier donné en entrée a bien été utilisé comme le corps du message de notre courrier. Il nous a suffit d'y adjoindre un sujet, puis les adresses de l'expéditeur et du destinataire.

Apportons immédiatement quelques informations supplémentaires quant aux destinataires de l'e-mail. Il est possible de définir plusieurs destinataires en les séparant par une virgule, de la façon suivante :

```
myemail['To'] = 'recipient@recipient.com,foo@foo.com,bar@bar.com'
```

Vous pouvez souhaiter envoyer l'e-mail à une personne et mettre la seconde en copie. Pour cela, nous utilisons le code suivant :

```
myemail['To'] = 'recipient@recipient.com'
myemail['Cc'] = 'bar@bar.com'
```

Enfin, vous pouvez souhaiter envoyer votre e-mail à un destinataire et faire parvenir à d'autres personnes une copie cachée de cet e-mail. Pour ce faire, nous utilisons les deux lignes suivantes :

```
myemail['To'] = 'recipient@recipient.com'
myemail['Bcc'] = 'bar@bar.com'
```

Enfin, si vous souhaitez utiliser un serveur SMTP local pour l'envoi de vos e-mails, il suffit de remplacer la ligne contenant l'adresse de votre serveur SMTP distant par :

```
smtpserver = smtplib.SMTP('127.0.0.1')
```

Ou par l'adresse IP ou le nom de domaine de la machine qui héberge le serveur SMTP.

Ce premier exemple introduit les bases fondamentales pour envoyer un e-mail en Python. Les programmes suivants vont nous permettre d'enrichir notre domaine de possibilités.

3 Utilisation des caractères accentués dans le sujet et le corps du message

Pour rédiger ses e-mails en bon français, il est nécessaire de gérer les caractères accentués lors de l'envoi de vos e-mails. Tous les clients de messagerie dignes de ce nom supportent aujourd'hui l'encodage UTF-8. Nous l'emploierons donc dans le sujet et le corps de nos e-mails. Cela peut se faire très simplement en ajoutant quelques modifications au programme de l'exemple précédent. Voici le code dans un nouveau fichier `email-from-utf8-text-file.py` :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# sert à utiliser l'UTF-8 dans le sujet de l'e-mail
from email.header import Header

# import pour manipuler le texte en format MIME
from email.mime.text import MIMEText

# nous ouvrons notre fichier texte en lecture
myfile = open('emailutf8body', 'r')
# conversion en format MIME
myemail = MIMEText(myfile.read(), _charset='utf8')
# fermeture du fichier
myfile.close()

# le sujet de l'e-mail
myemail['Subject'] = Header('Déjà le deuxième exemple', 'utf8')
# l'adresse de l'expéditeur
myemail['From'] = 'sender@sender.com'
# l'adresse du destinataire
myemail['To'] = 'recipient@recipient.com'

# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x email-from-utf8-text-file.py
$ ./email-from-utf8-text-file.py
```

Soulignons les quelques différences. Lors de notre appel à la classe `MIMEText`, nous passons l'argument `_charset='utf8'` afin de spécifier l'encodage d'origine de nos données. Un peu plus bas, nous importons la classe `Header` du module `email.header` qui va nous permettre de définir un en-tête codé en UTF-8 comme sujet de notre e-mail. Nous pouvons ainsi librement utiliser les caractères accentués sans avoir à attendre de surprises à l'envoi du message ou à son arrivée.

4 Joindre à votre e-mail une image

Nous allons maintenant voir comment joindre une image, en l'occurrence un fichier PNG, à notre e-mail. Il peut en effet être intéressant d'automatiser le processus d'envoi d'une image par e-mail pour, par exemple, joindre un graphique à l'envoi automatique d'une alerte à destination de vos collègues de travail dans le cadre de la supervision d'un parc informatique.

Voici le code complet d'un programme vous permettant d'envoyer une image par e-mail. Nous saisissons les lignes suivantes dans le fichier nommé `email-with-image.py` :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# sert à utiliser l'UTF-8 dans le sujet de l'e-mail
from email.header import Header

# import pour manipuler le texte en format MIME
from email.mime.text import MIMEText

# import pour manipuler une image en format MIME
from email.mime.image import MIMEImage

# import pour joindre image et texte dans un e-mail
from email.mime.multipart import MIMEMultipart

# nous préparons le corps de notre message
mybody = MIMEText('Voici un e-mail contenant une image,
opération réussie!', _charset='utf8')

# Nous ouvrons l'image en question
myfile = 'image.png'
image = open(myfile, 'rb')
# Nous préparons l'utilisation de l'image
myimage = MIMEImage(image.read())
myimage.add_header('Content-Disposition', 'attachment',
filename=myfile)

# Nous préparons le support au texte et à l'image
myemail = MIMEMultipart()
# réunissons maintenant le texte et l'image
myemail.attach(mybody)
myemail.attach(myimage)
# le sujet de l'e-mail
myemail['Subject'] = Header("Réunissons le texte et l'image",
'utf8')
# l'adresse de l'expéditeur
myemail['From'] = 'sender@sender.com'
# l'adresse du destinataire
myemail['To'] = 'recipient@recipient.com'

# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x email-with-image.py
$ ./email-with-image.py
```

Détaillons un peu le code ci-dessus. Nous commençons par importer les classes qui, comme nous l'avons vu dans l'exemple précédent, vont nous permettre d'écrire un e-mail orienté texte. Puis, nous importons deux classes `MIMEImage` et `MIMEMultipart` qui vont nous permettre respectivement d'inclure une image dans notre e-mail et d'avoir à la fois du texte et une image dans l'e-mail.

Un peu plus bas dans le code, nous ouvrons l'image et passons le résultat de la méthode `read()` à la classe `MIMEImage`. Nous définissons également à l'aide de `add_header()` que cette image est bien une pièce jointe qui porte un nom que nous définissons à l'aide de l'argument `filename`.

Enfin, nous instançons un objet à partir de la classe `MIMEMultipart`, lui attachons le texte puis l'image avant d'envoyer notre résultat de la même façon que dans les exemples précédents.

Envoyer un e-mail à l'aide de la classe `MIMEMultipart` vous permet d'obtenir un e-mail au format texte dont les pièces jointes sont bien distinctes du corps de votre message. La plupart des clients de messagerie actuels présenteront clairement les deux parties du message, ce qui vous permettra de bien faire la distinction et de facilement enregistrer les images en question sur votre ordinateur.

5 Embarquer une image dans votre e-mail au format HTML

Nous venons de voir comment envoyer un e-mail dissociant proprement texte et image. Toutefois, le format HTML offre des possibilités intéressantes de mélange entre le texte et les images. C'est pourquoi nous vous proposons le programme suivant, qui indique comment embarquer dans votre e-mail des images qui seront affichées dans le corps de votre message au format HTML.

Nous saisissons les lignes suivantes dans le fichier nommé `email-html-image.py` :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# sert à utiliser l'UTF-8 dans le sujet de l'e-mail
from email.header import Header

# import pour manipuler le HTML au format MIME
from email.mime.text import MIMEText

# import pour joindre image et HTML dans un e-mail
from email.mime.multipart import MIMEMultipart

# import pour manipuler une image en format MIME
from email.mime.image import MIMEImage

# nous préparons le corps de notre message
htmlcode = """\
<html>
  <head></head>
  <body>
    <p>Un e-mail au format HTML !<br>
       Nous embarquons une image : <br>
       Voici cette image : <img src="cid:image1">
    </p>
  </body>
</html>
"""
# utf8 afin d'utiliser des caractères accentuées dans le code HTML
```

```
htmlpart = MIMEText(htmlcode, 'html', _charset='utf8')

# Nous ouvrons l'image en question
image = open('image.png', 'rb')
# Nous préparons l'utilisation de l'image
myimage = MIMEImage(image.read())
image.close()
# nous identifions l'image en lui attribuant un cid
myimage.add_header('Content-ID', '<image1>')

# Nous préparons le support au texte et à l'image - related
est important
myemail = MIMEMultipart('related')
# réunissons maintenant le texte et l'image
myemail.attach(htmlpart)
myemail.attach(myimage)
# le sujet de l'e-mail
myemail['Subject'] = Header("Une image dans une page HTML !",
'utf8')
# l'adresse de l'expéditeur
myemail['From'] = 'sender@sender.com'
# l'adresse du destinataire
myemail['To'] = 'recipient@recipient.com'
# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x email-html-image.py
$ ./email-html-image.py
```

Cet exemple est sensiblement le même que le précédent. Pourtant quelques importantes différences sont à expliquer.

Lorsque nous définissons le code HTML à embarquer dans notre e-mail, nous utilisons comme source d'image la chaîne `cid:image1`. Cette dernière signifie que la source de l'image est une image dont l'identifiant de contenu est nommé `image1`. Il ne s'agit donc pas du nom de l'image sur votre disque dur. Cette valeur est définie plus bas lorsque nous définissons l'en-tête de l'image à l'aide de la méthode `add_header` qui prend en premier argument `Content-ID` suivi par une balise dont le nom est `image1`.

Autre paramètre important, vous devez spécifier la chaîne `related` comme argument lors de l'instanciation de la classe `MIMEMultipart`.

Avec cet exemple, vous êtes maintenant paré pour écrire des e-mails en HTML vous permettant d'inclure directement des images au sein de votre texte, sans la séparation nette entre les différents contenus qui caractérise le format texte simple.

6 Joindre un fichier son à votre e-mail

De la même façon que dans l'exemple précédent, il est également possible de joindre à votre e-mail un fichier son. L'exemple précédent nécessite toutefois quelques modifications.

Voici le code complet d'un programme vous permettant d'envoyer un fichier son par e-mail. Nous saisissons les lignes suivantes dans le fichier nommé `email-with-music.py` :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# sert à utiliser l'UTF-8 dans le sujet de l'e-mail
from email.header import Header

# import pour manipuler le texte en format MIME
from email.mime.text import MIMEText

# import pour manipuler une fichier son en format MIME
from email.mime.audio import MIMEAudio

# import pour joindre image et texte dans un e-mail
from email.mime.multipart import MIMEMultipart

# nous préparons le corps de notre message
mybody = MIMEText('Voici un e-mail contenant un fichier son,
opération réussie!', _charset='utf8')

# Nous ouvrons le fichier son en question
myfile = 'music.mp3'
music = open(myfile, 'rb')
# Nous préparons l'utilisation du fichier son
mymusic = MIMEAudio(music.read(), 'mpeg')
mymusic.add_header('Content-Disposition', 'attachment',
filename=myfile)

# Nous préparons le support au texte et au son
myemail = MIMEMultipart()
# réunissons maintenant le texte et le son
myemail.attach(mybody)
myemail.attach(mymusic)
# le sujet de l'e-mail
myemail['Subject'] = Header("Réunissons le texte et le son !",
'utf8')
# l'adresse de l'expéditeur
myemail['From'] = 'sender@sender.com'
# l'adresse du destinataire
myemail['To'] = 'recipient@recipient.com'

# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x email-with-music.py
$ ./email-with-music.py
```

Les modifications par rapport à l'exemple précédent commencent avec la déclaration d'import de la classe `MIMEAudio` qui va nous servir à joindre notre fichier son à l'e-mail. Plus bas, nous ouvrons notre fichier avant de transmettre son contenu à notre objet `mymusic` instancié à partir de la classe `MIMEAudio`. Le second paramètre `mpeg` lors de cette instanciation sert à identifier le type de fichier son utilisé.

7 Envoyer un e-mail avec n'importe quel type de fichier en pièce jointe

Nous avons présenté dans les deux exemples précédents comment envoyer une image ou un fichier son attaché à votre e-mail. Toutefois, il y a fort à parier que vous voudrez rapidement envoyer d'autres types de fichiers qu'il n'est pas possible de définir aussi précisément. Afin d'assurer à vos communications une base plus large de pièces jointes, nous présentons ici la méthode appropriée pour joindre à vos e-mails n'importe quel type de données. Nous saisissons les lignes suivantes dans le fichier nommé `email-with-file.py` :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# sert à encoder la pièce jointe
from email import encoders

# sert à utiliser l'UTF-8 dans le sujet de l'e-mail
from email.header import Header

# import pour manipuler le texte en format MIME
from email.mime.text import MIMEText

# import pour manipuler une fichier arbitraire en format MIME
from email.mime.base import MIMEBase

# import pour joindre image et texte dans un e-mail
from email.mime.multipart import MIMEMultipart

# nous préparons le corps de notre message
mybody = MIMEText('Voici un e-mail contenant un fichier de type
indéfini, opération réussie!', _charset='utf8')

# Nous ouvrons le fichier en question
myfile = 'attachment.tar.gz'
# Nous définissons un type par défaut
ctype = 'application/octet-stream'
maintype, subtype = ctype.split('/', 1)
# nous ouvrons le fichier
archive = open(myfile, 'rb')
# Nous préparons l'inclusion du fichier dans notre e-mail
myarchive = MIMEBase(maintype, subtype)
myarchive.set_payload(archive.read())
archive.close()
myarchive.add_header('Content-Disposition', 'attachment',
filename=myfile)
encoders.encode_base64(myarchive)

# Nous préparons le support au texte et au fichier joint
myemail = MIMEMultipart()
# réunissons maintenant le texte et le fichier joint
myemail.attach(mybody)
myemail.attach(myarchive)
# le sujet de l'e-mail
myemail['Subject'] = Header("Réunissons le texte et un fichier
arbitraire !", 'utf8')
# l'adresse de l'expéditeur
myemail['From'] = 'sender@sender.com'
# l'adresse du destinataire
myemail['To'] = 'recipient@recipient.com'

# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x email-with-file.py
$ ./email-with-file.py
```

Plusieurs différences importantes se sont glissées dans ce nouveau programme par rapport au précédent. Nous importons le module `encoders` afin d'encoder explicitement la pièce jointe. Nous importons également la classe `MIMEBase` afin de générer un objet MIME de base, à la différence de ceux orientés image ou son que nous avons utilisés précédemment.

Plus bas, après avoir précisé le nom de notre pièce jointe, nous définissons un type générique que nous utilisons lors de l'instanciation de la classe `MIMEBase` pour générer l'objet `myarchive`. Nous faisons ensuite appel à sa méthode `set_payload` pour définir les données contenues dans notre objet MIME de base. Enfin, nous encodons ces données grâce à la fonction `encode_base64` du module `encoders`.

Cet exemple revêt un fort intérêt pour tous vos envois automatisés d'e-mail embarquant des pièces jointes. En effet, si vous ne souhaitez pas transporter une image ou du son, les caractéristiques de base de l'objet MIME présenté ci-dessus s'adapteront parfaitement à vos envois. Nous avons ici pris l'exemple d'une archive, mais vous pouvez tout aussi bien distribuer des fichiers binaires ou tout autre fichier n'étant pas au format texte.

8 Faire parvenir un e-mail au format texte et au format HTML

Dans le cadre de l'envoi automatisé d'un e-mail à un certain nombre de personnes, il peut s'avérer intéressant de proposer deux formats pour votre e-mail. Un format enrichi écrit en HTML qui permet l'inclusion de liens ou d'images mélangés au texte, et un format texte simple assurant une lecture épurée.

Voici le code complet d'un programme vous permettant d'envoyer un e-mail lisible dans les deux formats texte et HTML. Nous saisissons les lignes suivantes dans le fichier nommé `email-two-formats.py` :

```
#!/usr/bin/python3.2

# import de smtplib pour envoyer l'email
import smtplib

# sert à utiliser l'UTF-8 dans le sujet de l'e-mail
from email.header import Header

# import pour manipuler le texte en format MIME
from email.mime.text import MIMEText

# import pour joindre image et texte dans un e-mail
from email.mime.multipart import MIMEMultipart

# nous préparons le corps de notre message
textpart = MIMEText('Voici un e-mail se présentant à la fois sous format texte et sous
format HTML !', 'plain', _charset='utf8')
htmlcode = """\
<html>
  <head></head>
  <body>
    <p>Un e-mail au format HTML!<br>
       En voici la preuve : <br>
       Il contient un lien hypertexte vers le site des Éditions Diamond : <a
href="http://ed-diamond.com/">le lien</a> en question.
    </p>
  </body>
</html>
"""
htmlpart = MIMEText(htmlcode, 'html', _charset='utf8')

# Nous préparons le support au texte et au format HTML
myemail = MIMEMultipart('alternative')
# réunissons maintenant le texte et l'HTML
myemail.attach(textpart)
myemail.attach(htmlpart)
# le sujet de l'e-mail
myemail['Subject'] = Header("Un e-mail dans deux formats", 'utf8')
# l'adresse de l'expéditeur
myemail['From'] = 'chaica@ohmytux.com'
# l'adresse du destinataire
myemail['To'] = 'chaica@ohmytux.com'

# Envoi du message via notre serveur SMTP distant
smtpserver = smtplib.SMTP('smtp.free.fr')
smtpserver.send_message(myemail)
smtpserver.quit()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x email-two-formats.py
$ ./email-two-formats.p
```

L'e-mail qui parvient à votre destinataire peut être consulté de deux façons. Selon vos préférences par défaut, il s'affichera au format HTML, vous présentant un formatage caractéristique et un lien hypertexte aisément identifiable, ou au format texte simple. Cette offre d'alternative permet à tous les lecteurs de votre e-mail de profiter de son contenu dans leur format préféré.

Conclusion

Les différents programmes que nous avons présentés couvrent efficacement les multiples possibilités que vous offre l'envoi d'e-mail à l'aide du module `email` en Python 3.2. Les paragraphes 2 et 3 ont posé les bases de ce que doit être un e-mail bien rédigé, en utilisant correctement les structures offertes par le module `email` et en gérant les caractères accentués.

Les paragraphes 4, 5, 6 et 7 nous ont permis de voir comment joindre une pièce jointe à un e-mail. Ce type d'opération est simple à automatiser et revêt un fort intérêt pour, par exemple, faire parvenir de façon automatique des images, des fichiers audio ou tout fichier quel que soit son type par e-mail à plusieurs personnes, leur évitant la démarche de devoir eux-mêmes agir pour accéder aux fichiers en question.

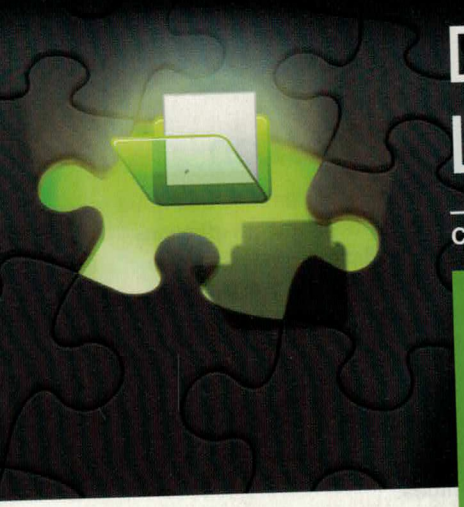
Le paragraphe 8, quant à lui, a mis l'accent sur la possibilité d'offrir au sein d'un même e-mail deux formats de visionnage à vos utilisateurs, afin de s'adapter aux habitudes, mais aussi aux contraintes de vos destinataires. Dans un contexte où le contenu des e-mails ne cesse de s'enrichir, au point d'arriver parfois à des résultats illisibles, une lecture simple et directe via un format simplifié en mode texte pur plaira sans aucun doute à une bonne part de vos lecteurs.

Vous êtes désormais armé pour automatiser vos envois d'e-mails, n'hésitez pas à faire évoluer les programmes ci-dessus pour répondre aux besoins de votre entreprise ou de vos projets personnels ! ■

Liens

Documentation officielle du module `email` de Python 3.2 : http://docs.python.org/py3k/library/email.html

Documentation officielle du module `smtplib` de Python 3.2 : http://docs.python.org/py3k/library/smtplib.html#module-smtplib



DU SQL DANS VOS FICHIERS : LE MODULE SQLITE3

Carl Chenet

Vous cherchez à stocker des informations dans un fichier plat ou en mémoire vive ? Vous souhaitez également pouvoir en extraire des informations à l'aide du langage de requêtes le plus puissant aujourd'hui, à savoir SQL ? Présent dans la puissante bibliothèque standard du langage Python, le module sqlite3 vous offre la possibilité de traiter un fichier ou un espace mémoire exactement comme une base de données...

Présentant le double avantage d'être présent par défaut à l'installation de Python 3.2 et d'être extrêmement riche dans son utilisation, `sqlite3` gagne à être connu et conviendra parfaitement à un grand nombre d'utilisations, en lieu et place des classiques fichiers plats pour lesquels une mauvaise insertion ou l'utilisation d'un caractère non attendu suffit à rendre inopérante toute la mécanique que vous avez patiemment développée.

Déléguant cette tâche au SQL et à une interface de base de données avec laquelle les développeurs et les administrateurs système ont l'habitude de travailler, `sqlite3` comblera vos attentes en renforçant la robustesse de vos développements. Présentation en règle.

1 Installer Python 3.2 et SQLite3

Il est de manière générale conseillé d'utiliser les paquets disponibles pour votre distribution GNU/Linux. Par exemple, sur votre système Debian (Wheezy ou Sid), il suffit de taper la commande suivante en tant qu'utilisateur root :

```
# apt-get install python3.2 sqlite3
```

Si votre distribution ne propose pas de paquet pour Python 3.2, vous pouvez l'installer de la façon suivante à partir des sources :

```
$ wget http://www.python.org/ftp/python/3.2.2/Python-3.2.2.tgz
$ tar zxvf Python-3.2.2.tgz
$ cd Python-3.2.2
$ ./configure
$ make
```

Vous avez maintenant dans votre répertoire courant l'exécutable `python` que vous pouvez utiliser directement. Pour l'installer, vous devez passer la commande suivante en tant qu'utilisateur root :

```
# make install
```

Nous allons faire de même pour SQLite3. Ce programme n'est pas indispensable à la mise en place de votre base de données, mais il nous permettra de consulter simplement le contenu de votre base à partir d'un terminal interactif. Si votre distribution ne propose pas un paquet pour `sqlite3`, vous pouvez l'installer de la façon suivante à partir des sources :

```
$ wget http://www.sqlite.org/sqlite-autoconf-3070900.tar.gz
$ tar zxvf sqlite-autoconf-3070900.tar.gz
$ cd sqlite-autoconf-3070900/
$ ./configure
$ make
```

Vous avez maintenant dans votre répertoire courant l'exécutable `sqlite3` que vous pouvez utiliser directement. Pour l'installer, vous devez passer la commande suivante en tant qu'utilisateur root :

```
# make install
```

Votre ordinateur est dorénavant équipé de Python 3.2 et de l'application SQLite3. Entrons dans le vif du sujet avec un exemple concret.

2 Votre collection de DVD... dans un fichier plat

Les cinéphiles ont parfois des collections de DVD très fournies. Il peut être utile de recourir à l'informatique pour garder une idée claire de l'état de sa collection. Toutefois, installer une base de données MySQL ou PostgreSQL pour quelques centaines, voire quelques milliers de titres, peut sembler disproportionné. Il est en effet difficile de s'assurer que chaque nouvelle version d'un programme que l'on écrit soi-même demeure compatible dans le temps avec ce programme externe, sans même parler de l'installation d'une base de données comme celles précédemment citées qui peut parfois tourner au cauchemar, en particulier lors de la définition des droits d'accès à appliquer aux différents utilisateurs.

D'un autre côté, gérer un grand nombre d'attributs par DVD pour un grand nombre de titres entraîne le besoin d'un fichier bien organisé et simple à consulter. Quelques lignes en Python 3.2 - la plus récente version stable de Python - et le module `sqlite3` apparaissent comme une excellente solution à ce problème.

3 Créer votre base de données SQLite3

Commençons par créer le fichier plat dont se servira le module `sqlite3` pour stocker les données concernant nos DVD. Pour cela, nous créons un script Python nommé `create-dvdbase.py` dans lequel nous écrivons les lignes suivantes :

```
#!/usr/bin/python3.2

# crée la base SQLite3 pour nos DVD

# importe le module sqlite3
import sqlite3

# crée la connexion à la base de données
dvdconn = sqlite3.connect('dvdbase')

# crée le curseur pour exécuter les requêtes SQL
dvdc = dvdconn.cursor()

# crée la table dvd
dvdc.execute('''create table dvd
(id integer primary key, title text, purchasedate text,
 comment text)''')

# enregistre les modifications effectuées
dvdconn.commit()

# ferme le curseur précédemment utilisé
dvdc.close()
```

Nous rendons le programme exécutable et l'exécutons à l'aide des commandes suivantes :

```
$ chmod u+x create-dvdbase.py
$ ./create-dvdbase.py
```

Notre base de données est maintenant créée. Afin de nous en assurer, nous pouvons utiliser le programme externe `sqlite3` qui va nous offrir un terminal à partir duquel nous obtiendrons différentes informations sur notre nouvelle base SQLite3 :

```
$ sqlite3 dvdbase
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------------------
0    main             /home/chaica/progra/python/sqlite3/dvdbase
sqlite> .tables
dvd
sqlite> select * from dvd;
sqlite>
```

Le résultat est cohérent. L'ordre `.databases` nous indique bien le chemin du fichier que nous avons créé lors de la connexion `sqlite3` et l'ordre `.tables` retourne bien le nom de la table `dvd` qui vient juste d'être créée. Notre premier `select` - censé retourner tous les enregistrements présents dans la table `dvd` - ne retourne rien.

Revenons un instant sur le code Python ci-dessus. Nous commençons par explicitement invoquer le programme Python 3.2 qui interprétera notre script. Nous importons ensuite le module `sqlite3`. Immédiatement après, nous définissons dans notre code un objet qui, émanant de l'appel à la fonction `sqlite3.connect` - à laquelle on passe un chemin vers le fichier qui nous servira désormais de base de données - représentera notre connexion à la base.

Nous instançons par la suite et à partir de la connexion précédemment créée, un curseur, c'est-à-dire un objet qui nous permettra d'exécuter et de consulter le résultat de requêtes SQL. Aussitôt après, nous exécutons grâce à ce curseur à travers l'appel à la méthode `execute` l'ordre SQL de création d'une table `dvd`, destinée à accueillir les différentes informations concernant nos DVD.

Un ordre très important suit l'exécution des requêtes SQL. En effet, pour le moment, si les modifications ont bien été exécutées, elles ne sont toutefois pas encore enregistrées dans votre fichier. Pour cela, nous faisons un appel explicite à la méthode `commit` de l'objet incarnant notre connexion à la base de données. Enfin, nous fermons explicitement le curseur que nous avons utilisé un peu plus haut.

4 Insérer des données dans votre base

Afin d'insuffler un peu de vie à notre base de DVD, nous allons maintenant y injecter quelques données. Pour cela, nous créons un script Python nommé `insert-data-dvd-base.py` dans lequel nous écrivons les lignes suivantes :

```
#!/usr/bin/python3.2

# insère des données dans la base SQLite3

# importe le module sqlite3
import sqlite3

# crée la connexion à la base de données
dvdconn = sqlite3.connect('dvdbase')

# crée le curseur pour exécuter les requêtes SQL
dvdc = dvdconn.cursor()

# ajoute trois enregistrements à la table dvd
for data in [('Star Wars 4', '2009-12-24', 'Un très bon film'),
```

```
            ('Star Wars 5', '2010-01-22', 'Sûrement le meilleur de la
            série'),
            ('Star Wars 6', '2011-12-24', 'Je suis ton père')]:
    dvdc.execute('insert into dvd (title, purchasedate, comment) values
    (?,?,?)', data)

# enregistre les modifications effectuées
dvdconn.commit()

# ferme le curseur précédemment utilisé
dvdc.close()
```

Nous rendons exécutable le programme et le lançons :

```
$ chmod u+x  insert-data-dvdbase.py
$ ./ insert-data-dvdbase.py
```

Un rapide coup d'œil à notre session de terminal `sqlite3` nous permet de vérifier que les nouveaux enregistrements ont bien été ajoutés :

```
sqlite> select * from dvd;
1|Star Wars 4|2009-12-24|Un très bon film
2|Star Wars 5|2010-01-22|Sûrement le meilleur de la série
3|Star Wars 6|2011-12-24|Je suis ton père
```

Dans l'exemple ci-dessus, nous avons défini une liste nommée `data`. Chaque élément de cette liste est un tuple Python. Chaque tuple représente un enregistrement à insérer dans notre base de données. Cette insertion est exécutée par l'appel à la méthode `execute` de l'objet `dvdc`. L'itération à l'aide de la boucle `for...in` nous permet - dans un style très Pythoniste - d'exécuter nos différentes requêtes les unes à la suite des autres avec un minimum de code.

Moins élégant mais plus flexible si par exemple les insertions concernent différentes tables, nous avons la possibilité d'utiliser la méthode `executescript` avec le code suivant :

```
dvdc.executescript("""insert into dvd (title, purchasedate, comment)
values ('Star Wars 4', '2009-12-24', 'Un très bon film');
insert into dvd (title, purchasedate, comment) values ('Star Wars 5',
'2010-01-22', 'Sûrement le meilleur de la série');
insert into dvd (title, purchasedate, comment) values ('Star Wars 6',
'2011-12-24', 'Je suis ton père');
""")
```

Cette méthode prend simplement une chaîne en entrée, cette dernière contenant les différents ordres SQL à exécuter. Il est bien sûr possible d'y placer plusieurs insertions vers différentes tables de votre base, mais aussi des créations de tables ou toute autre requête SQL.

5 Améliorer l'intégrité de votre base de données

Afin de garantir l'intégrité de notre base de données, nous allons utiliser les gestionnaires de contexte avec le module `sqlite3`. Les gestionnaires de contexte assurent l'exécution de certaines opérations de façon transparente. On peut citer en exemple le gestionnaire de contexte utilisé lorsque vous manipulez un objet fichier créé avec l'appel à la fonction `open`, appel précédé du mot-clé `with`. En reprenant l'exemple du chapitre précédent, nous obtenons le code suivant que nous écrivons dans le fichier `insert-data-dvdbase-with-cm.py` :

```
#!/usr/bin/python3.2

# insère des données dans la base SQLite3

# importe le module sqlite3
import sqlite3

# crée la connexion à la base de données
dvdconn = sqlite3.connect('dvdbase')

# crée le curseur pour exécuter les requêtes SQL
dvdc = dvdconn.cursor()

# insertion de trois nouveaux DVD
try:
    with dvdconn:
        for data in [('Star Wars 4', '2009-12-24', 'Un très bon film'),
                     ('Star Wars 5', '2010-01-22', 'Sûrement le meilleur
                     de la série'),
                     ('Star Wars 6', '2011-12-24', 'Je suis ton père')]:
            dvdc.execute('insert into dvd (title, purchasedate, comment)
            values (?,?,?)', data)
except sqlite3.IntegrityError:
    print('Issue while inserting data in the database.')

# ferme le curseur précédemment utilisé
dvdc.close()
```

Puis, nous passons les commandes suivantes afin de pouvoir exécuter le programme :

```
$ chmod u+x  insert-data-dvdbase-with-cm.py
$ ./insert-data-dvdbase-with-cm.py
```

Au sein du bloc `try... except`, nous plaçons le mot-clé `with` suivi de notre objet de connexion à la base. Cela va nous assurer qu'un *commit* est bien exécuté à la fin des insertions.

Si une erreur venait à se produire pendant l'insertion des données, le gestionnaire de contexte nous garantit l'exécution d'un *rollback*, opération nous assurant la disparition de toutes les modifications effectuées sur notre base depuis le dernier commit. Et tout ceci de manière implicite, simplement lié au fait que nous mettons en relation le mot-clé `with` avec un objet représentant une connexion à une base de données SQLite3. Les gestionnaires de contexte peuvent tout d'abord sembler assez abstraits à aborder, mais utilisés dans des cas concrets comme la gestion du bon déroulement des transactions SQL avec le module `sqlite3`, ils vous seront bientôt indispensables.

6 Consulter des données dans notre base SQLite3

Nous avons, dans l'exemple précédent, eu recours à l'utilitaire `sqlite3` pour consulter les différents enregistrements présents dans notre table. Il serait pour

nous bien plus intéressant de le faire en Python. Pour cela, nous allons réaliser un programme nous retournant le contenu de la table `dvd` de notre base. Pour ce faire, nous copions les lignes suivantes dans le fichier `select-data-dvdbase.py` :

```
#!/usr/bin/python3.2

# consulter des données dans la base SQLite3

# importe le module sqlite3
import sqlite3

# crée la connexion à la base de données
dvdconn = sqlite3.connect('dvdbase')

# crée le curseur pour exécuter les requêtes SQL
dvdc = dvdconn.cursor()

# exécution de la requête SQL
dvdc.execute('select * from dvd')
# nous parcourons les enregistrements de notre curseur
for row in dvdc:
    # nous adoptons le même formatage que la commande sqlite3
    print('{}|{}|{}|{}'.format(*row))

# enregistre les modifications effectuées
dvdconn.commit()

# ferme le curseur précédemment utilisé
dvdc.close()
```

Nous rendons ce programme exécutable avant de le lancer :

```
$ chmod u+x select-data-dvdbase.py
$ ./select-data-dvdbase.py
1|Star Wars 4|2009-12-24|Un très bon film
2|Star Wars 5|2010-01-22|Sûrement le meilleur de la série
3|Star Wars 6|2011-12-24|Je suis ton père
```

L'exécution de ce programme nous retourne la liste des enregistrements présents dans la table `dvd` de notre base de données SQLite3, sous le format (vu un peu plus haut) utilisé par l'utilitaire `sqlite3`.

Dans notre code Python, nous itérons à l'aide de la structure `for...in` au sein des enregistrements présents dans notre curseur. Ce dernier retourne des enregistrements un à un, chacun sous la forme d'un tuple Python. L'utilisation de la fonction `print` et de 4 jokers (motif `{}`) associée à la méthode `format` de l'objet chaîne de caractères nous assure une bonne flexibilité et la réduction de la taille de notre code.

7 Une base de données en mémoire vive

Afin de décrire une fonctionnalité majeure de SQLite3 accessible via le module `sqlite3` de Python, nous allons quitter le cadre de l'exemple suivi jusqu'ici. `sqlite3` vous permet d'effectuer toutes les opérations que nous avons vues jusqu'ici, mais avec la possibilité supplémentaire de ne pas créer de fichier sur votre disque. Il va donc s'agir de travailler en mémoire vive, ce qui sous-entend une volatilité importante de vos données, par exemple en cas de panne matérielle.

On peut imaginer de nombreux cas d'utilisation de ce type de base, par exemple un environnement où la rapidité d'accès, l'intégrité des données, ainsi que leur simplicité de manipulation vont être primordiales pendant le temps d'exécution de votre application, leur conservation n'étant pas nécessaire.

Nous reprenons ici l'exemple de création de notre base de données et copions les lignes suivantes dans le fichier `create-dvdbase-in-memory.py` :

```
#!/usr/bin/python3.2

# crée la base SQLite3 pour nos DVD en mémoire vive

# importe le module sqlite3
import sqlite3

# crée la connexion à la base de données
dvdconn = sqlite3.connect(':memory:')

# crée le curseur pour exécuter les requêtes SQL
dvdc = dvdconn.cursor()

# crée la base de données
try:
    with dvdconn:
        dvdc.execute('''create table dvd
        (id integer primary key, title text, purchasedate text,
         comment text)''')
except sqlite3.IntegrityError:
    print('Error while creating the database')

# ferme le curseur précédemment utilisé
dvdc.close()
```

Nous rendons le programme exécutable et l'exécutons à l'aide des commandes suivantes :

```
$ chmod u+x create-dvdbase-in-memory.py
$ ./create-dvdbase-in-memory.py
```

Soulignons la différence majeure qui permet au code ci-dessus de s'exécuter en mémoire, à savoir principalement la déclaration de la chaîne `:memory:` en lieu et place de l'habituel chemin vers le fichier que vous souhaitez traiter comme une base de données. Les autres opérations restent inchangées. Cette flexibilité offerte par SQLite3 permet d'adapter votre code à différents environnements faiblement pourvus ou dénués de mémoire de masse.

Il peut toutefois être utile de pouvoir sauver les données que vous avez stockées en mémoire pour, par exemple, les réexploiter dans un environnement pourvu, lui, de mémoire de masse. Pour ce faire, nous allons réaliser une extraction de ces données sous la forme d'un fichier contenant les opérations SQL nécessaires à reproduire notre base de données dans un autre environnement. On parle en anglais de *dump* d'une base de données.

Le module sqlite3 de Python offre une méthode simple pour générer ce fichier. Nous présentons son utilisation dans le programme suivant, qui extrait les données de notre base de données. Nous écrivons le code suivant dans le fichier dump-dvdbase.py :

```
#!/usr/bin/python3.2

# importe le module sqlite3
import sqlite3

# connecte à la base en mémoire
dvdconn = sqlite3.connect('dvdbase')
# gère un objet fichier via le gestionnaire de contexte
with open('dvdbase.sql', 'w') as dvdfile:
    # extrait les différents enregistrements
    for line in dvdconn.iterdump():
        # écrit les enregistrements dans le fichier
        dvdfile.write('%s\n' % line)
```

Nous rendons le programme exécutable et l'exécutons à l'aide des commandes suivantes :

```
$ chmod u+x dump-dvdbase.py
$ ./dump-dvdbase.py
```

Très concrètement, nous établissons une connexion avec la base de données. Puis, nous créons un fichier destiné à accueillir les données en provenance de notre base de données. Une simple boucle for...in permet ensuite de récupérer l'ordre SQL nécessaire pour recréer chaque enregistrement de notre base de données. Cette méthode est bien sûr utilisable avec les bases en mémoire en transformant le chemin du fichier en :memory: comme vu plus haut.

Dernière étape de la réalisation de notre sauvegarde, nous souhaitons maintenant à partir du fichier défini plus haut recréer notre base, notre table et nos enregistrements. Pour cela, rien de plus simple, nous allons utiliser notre fichier de sauvegarde allié à la méthode executescript déjà vue au chapitre 4.

Nous copions les lignes suivantes dans le fichier restore-dvdbase.py :

```
#!/usr/bin/python3.2

# importe le module sqlite3
import sqlite3

# connecte à la base en mémoire
dvdconn = sqlite3.connect('dvdbase')
dvdc = dvdconn.cursor()
# gère un objet fichier via le gestionnaire de contexte
with open('dvdbase.sql', 'r') as dvdfile:
    dvdc.executescript(dvdfile.read())
```

Puis, nous rendons ce programme exécutable. Avant de lancer ce script, il est nécessaire de supprimer le fichier dvdbase pour simuler la perte de nos données :

```
$ chmod u+x  restore-dvdbase.py
$ rm -f dvdbase
$ ./ restore-dvdbase.py
```

En exécutant à nouveau le programme select-data-dvdbase.py que nous avons écrit plus haut dans cet article, nous constatons que nos données sont bien de retour. Nous avons en effet ouvert le fichier dvdbase.sql et transmis son contenu sous la forme d'une chaîne de caractères, grâce à la méthode read et à la méthode executescript vue précédemment, qui s'est chargée d'exécuter les requêtes SQL en question.

La série d'exemples de ce chapitre vous a permis de vous familiariser avec une technique fondamentale dans le monde des bases de données, à savoir l'extraction des données à des fins de sauvegarde suivie par une restauration du schéma de la base et des enregistrements dans les tables.

Savoir que le schéma de la base et toutes ses données sont embarqués dans un unique fichier encourage à uniquement sauvegarder ce fichier. Et il est vrai que l'utilisation de iterdump n'est pas primordiale avec le module sqlite3 de Python dans un nombre important de cas d'utilisation. Toutefois, la technique présentée ici vous sera indispensable pour espérer conserver vos données issues d'une base créée dans un environnement pauvre ou dépourvu de mémoire de masse.

Conclusion

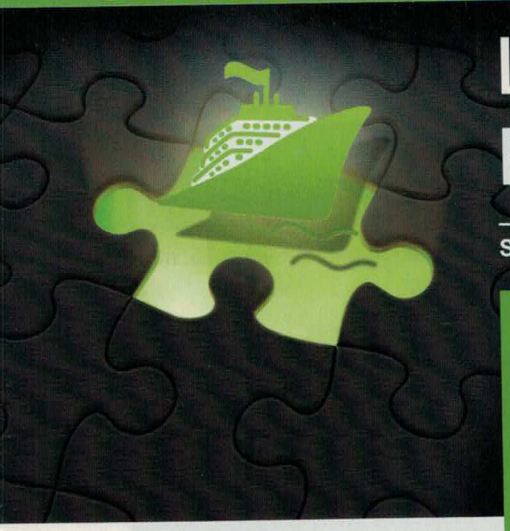
Comme nous l'avons vu à travers les différents exemples de cet article, le module sqlite3 offre un moyen sûr, souple et puissant d'interagir via le langage SQL avec un fichier plat ou un espace en mémoire vive. Trop souvent, les développeurs ont recours à une base de données traditionnelle comme MySQL ou PostgreSQL pour une simple application bureautique s'exécutant sur votre poste de travail, ce qui s'avère à la fois complexe et coûteux à maintenir dans le temps, avec principalement une dépendance dont il faut vérifier régulièrement l'évolution et la compatibilité avec votre code.

Le module sqlite3 présent par défaut dans la bibliothèque standard de Python simplifie grandement cet aspect en vous offrant de base la puissance du SQL associée à l'utilisation du langage Python. Création du schéma de la base, insertion des données, consultation des enregistrements, création et restauration de sauvegardes sont des opérations directement exécutables depuis votre code écrit en Python 3.2. Alors n'hésitez plus et plongez-vous dans l'utilisation du module sqlite3 ! ■

Liens

Documentation officielle du module sqlite3 de Python 3.2 : http://docs.python.org/py3k/library/sqlite3.html#module-sqlite3

Les gestionnaires de contexte en Python 3.2 : http://docs.python.org/py3k/reference/datamodel.html#context-managers



LE PYTHON QUI JOUAIT À LA BATAILLE NAVALE

Sébastien Chazallet

Python est un langage disposant de types de données hors du commun et de possibilités algorithmiques qui, couplés à l'excellente bibliothèque Pygame, en font un excellent outil à mettre en avant pour le développement de jeux. Python permet de réaliser du développement rapide, mais est également une excellente solution de prototypage, les temps de développement pouvant être divisés par dix par rapport à la référence que reste C++. Cet article a pour ambition de présenter une vision de quelques problématiques usuelles en développement de jeux.

Note

Nous allons utiliser ici Python 2.x, car le portage de Pygame n'est pas encore terminé.

1 Représenter un plateau de bataille navale

La bataille navale est un jeu présentant deux plateaux carrés de 10 cases de côté. Sur le premier, on place ses bateaux, puis on note les coups de l'adversaire tandis que sur le second, on note ses propres coups. Un tir dans l'eau se note par l'apposition d'un pion blanc et un tir réussi par un pion rouge. Avant même de se poser une quelconque question à propos des problématiques liées à Pygame et son utilisation, il est nécessaire de savoir comment on va bien pouvoir représenter ces plateaux de jeu.

La manière la plus basique consiste à dire qu'un plateau de jeu est un **ensemble** de cases qui sont chacune un 2-uplet constitué des lignes qui sont représentées par une lettre (de A à J) et des colonnes qui le sont par un chiffre (de 0 à 9).

On aurait pu penser à un tableau de lignes, lesquelles seraient des tableaux de cases, mais cette structure est beaucoup trop lourde pour nos besoins réels. Un ensemble suffit largement et en plus nous permet de gérer toutes les problématiques d'unicité, mais surtout de profiter des notions ensemblistes (au sens mathématique du terme) que sont l'union, l'intersection et la différence qui correspondent exactement à nos besoins. Ce genre d'opération n'est pas commune, mais est l'un des points forts de Python, il faut savoir l'utiliser. Il y en a d'autres qui seront mis en exergue dans les petits éléments à suivre.

Pour commencer, il est nécessaire d'utiliser le module `itertools` qui est, soit dit en passant, un véritable bijou :

```
>>> from itertools import product
>>> table=set(product('ABCDEFGHIJ', '0123456789'))
>>> table
{('I', '9'), ('H', '4'), ('C', '4'), ('E', '0'), ('D', '5'), ('F', '4'), ('C', '3'), ('A', '1'),
('G', '9'), ('I', '0'), ('B', '0'), ('F', '5'), ('C', '2'), ('E', '2'), ('D', '3'), ('D', '8'),
('G', '8'), ('B', '1'), ('F', '6'), ('C', '1'), ('A', '8'), ('A', '3'), ('I', '2'), ('B', '2'),
('H', '3'), ('F', '7'), ('E', '5'), ('H', '8'), ('C', '0'), ('D', '1'), ('B', '3'), ('F', '0'),
('A', '5'), ('I', '4'), ('B', '4'), ('H', '1'), ('F', '1'), ('E', '7'), ('D', '6'), ('J', '0'),
('B', '5'), ('F', '2'), ('J', '1'), ('A', '7'), ('G', '3'), ('I', '6'), ('B', '6'), ('H', '7'),
('F', '3'), ('E', '1'), ('E', '8'), ('D', '4'), ('J', '2'), ('G', '2'), ('B', '7'), ('J', '3'),
('I', '1'), ('G', '1'), ('I', '8'), ('B', '8'), ('H', '5'), ('E', '3'), ('D', '2'), ('J', '4'),
('G', '0'), ('B', '9'), ('A', '9'), ('A', '0'), ('J', '5'), ('I', '3'), ('G', '7'), ('H', '2'),
('D', '0'), ('D', '9'), ('J', '6'), ('G', '6'), ('C', '9'), ('A', '2'), ('J', '7'), ('I', '5'),
('G', '5'), ('H', '0'), ('C', '8'), ('E', '4'), ('H', '9'), ('G', '4'), ('F', '8'), ('C', '7'),
('A', '4'), ('I', '7'), ('F', '9'), ('H', '6'), ('C', '6'), ('E', '6'), ('E', '9'), ('D', '7'),
('J', '8'), ('C', '5'), ('J', '9'), ('A', '6')}
```

L'idée maintenant est de positionner des bateaux. Pour l'exemple, on en positionne deux verticaux :

```
>>> bateau1=frozenset(zip('B'*5, [str(i) for i in range(3, 8)]))
>>> bateau2=frozenset(zip('C'*4, [str(i) for i in range(3, 7)]))
```

Et trois horizontaux :

```
>>> bateau3=frozenset(zip('ABCD', '1'*4))
>>> bateau4=frozenset(zip('ABC', '0'*3))
>>> bateau5=frozenset(zip('BC', '9'*2))
```

Normalement, nos bateaux ne doivent pas se chevaucher (une case ne peut être occupée que par un seul bateau) :

```
>>> bateau1 & bateau2 & bateau3 &
bateau4 & bateau5
frozenset()
```

L'ensemble des cases qui seront gagnantes lorsque l'adversaire les jouera sont

```
>>> occupes=bateau1 | bateau2 | bateau3 |
bateau4 | bateau5
```

On peut en profiter pour faire une seconde vérification, qui consiste à s'assurer que tous les bateaux ne dépassent pas du plateau de jeu (toutes les cases sont contenues dans le tableau) :

```
>>> occupes < table
True
```

On peut également réunir nos bateaux dans un ensemble :

```
>>> bateaux={bateau1, bateau2, bateau3,
bateau4, bateau5}
```

Pour terminer d'initialiser un plateau, on peut créer les ensembles vides que seront ceux des coups déjà joués et ceux des touches réalisées :

```
>>> deja_joues, touches = set(), set()
```

Voici de quelle manière on pourrait gérer la logique des données lorsque l'on jouerait une case :

```
>>> def jouer(case):
...     if case in deja_joues:
...         return False
...     table.remove(case)
...     deja_joues.add(case)
...     if case in occupes:
...         touches.add(case)
...     return True
...
```

Voici comment simuler le fait de jouer jusqu'à gagner en simulant des choix de jeu effectués au hasard :

```
>>> import random
>>> while(len(touches)<18):
...     jouer(random.choice(list(table)))
...
```

Mesurons l'efficacité de notre « intelligence artificielle » qui est plus artificielle qu'intelligente :

```
>>> coups_au_total=100-len(table)
>>> coups_au_total
97
```

Autant dire que le résultat n'est franchement pas terrible (on peut arriver à 75, mais également à 100…). Ce n'est pas vraiment surprenant.

Le moyen de savoir si la partie est gagnée est :

```
>>> len(touches)>=18:
True
```

Voici deux moyens de détecter le fait qu'un bateau ait été coulé (toutes ses cases sont touchées) :

```
>>> len(bateau1-touches)==0
True
>>> bateau1&touches==bateau1
True
```

Ce qui va être fait ici est uniquement la mécanique permettant de proposer à un joueur seul de trouver les bateaux posés par l'IA, ce qui est déjà un bon début.

Maintenant que les principes sont vus, on va créer un fichier `modele.py` qui va contenir une classe représentant le plateau et son contenu et possédant les méthodes nécessaires :

```
class Plateau:
        """
        >>> from modele import Plateau

        Pour tester le plateau de jeu:
        >>> Plateau.test()

        Pour initialiser le plateau de jeu:
        >>> plateau = Plateau()
        """

        def __init__(self):
                self.table = set(product('ABCDEFGHIJ', '0123456789'))

                self.bateaux = {
                        1: frozenset(zip('B'*5, [str(i) for i in range(2, 7)])),
                        2: frozenset(zip('C'*4, [str(i) for i in range(3, 7)])),
                        3: frozenset(zip('EFGH', '7'*4)),
                        4: frozenset(zip('GHI', '0'*3)),
                        5: frozenset(zip('IJ', '9'*2)),
                }

                self.occupés = self.bateaux[1] | self.bateaux[2] | self.bateaux[3] |
                self.bateaux[4] | self.bateaux[5]

                self.déjà_joués, self.touchés = set(), set()

        def jouer(self, case):
                if case in self.déjà_joués:
                        return False
                self.table.remove(case)
                self.déjà_joués.add(case)
                if case in self.occupés:
                        self.touchés.add(case)
                return True
```

```
    def partie_est_gagnée(self):
            return len(self.touchés)>=18

    def bateau_est_coulé(self, numéro):
            if numéro not in [1, 2, 3, 4, 5]:
                    return None
            return len(self.bateaux[numéro]-self.touchés)==0

    def nb_joues(self):
            return len(self.deja_joues)

    def nb_touches(self):
            return len(self.touches)

    def nb_coules(self):
            return sum([self.bateau_est_coule(n) and 1 or 0 for n in range(1,
            6)])

    @staticmethod
    def test():
            plateau = Plateau()
            if len(plateau.occupés) != 18:
                    print("Initialisation incorrecte 1")
                    return
            if not plateau.occupés < plateau.table:
                    print("Initialisation incorrecte 2")
                    return
            if len(plateau.bateaux[1] & plateau.bateaux[2] & plateau.bateaux[3]
            & plateau.bateaux[4] & plateau.bateaux[5]) != 0 :
                    print("Initialisation incorrecte 3")
                    return
            print("Initialisation correcte")

            while not plateau.partie_est_gagnée():
                    plateau.jouer(random.choice(list(plateau.table)))
            print("Coups joués: %s" % (100-len(plateau.table)))
```

Le modèle présenté ici est un excellent exemple de l'utilisation des ensembles pour répondre à des problématiques qui peuvent se révéler complexes si l'on n'utilise que des listes ou des listes de listes (matrices). En effet, la méthode `nb_coules`, par exemple, reste extrêmement simple alors que la problématique est vraiment ardue, puisqu'il s'agit de compter le nombre de bateaux dont toutes les cases ont été jouées.

Et voici comment tester ce fichier dans une console :

```
>>> from modele import Plateau
>>> Plateau.test()
Initialisation correcte
Coups joués: 86
>>> Plateau.test()
Initialisation correcte
Coups joués: 75
>>> Plateau.test()
Initialisation correcte
Coups joués: 100
```

À cet instant, nous avons de quoi gérer les données du jeu indépendamment de toute la partie affichage.

2 Problématiques purement liées à Pygame

Pygame est une bibliothèque Python construite au-dessus de la SDL (*Simple DirectMedia Layer*), une bibliothèque écrite en C, idéale pour ceux qui veulent débuter dans le développement multimédia. SDL gère l'affichage vidéo et l'audio ainsi que les périphériques (clavier et souris en particulier) et un élément essentiel qui est le temps. La méthode de programmation consiste à utiliser un gestionnaire d'événements pour interagir avec tous les événements clavier et souris.

Pour commencer, il faut réaliser quelques imports :

```
# -*- coding: utf-8 -*-

import random
import time
import pygame
import sys
from pygame.locals import *
from constantes import *
```

Dans notre module Python `constantes.py`, on met quelques valeurs utiles pour construire notre écran. Ces constantes sont en majuscules et le détail du fichier est donné au chapitre suivant. On importe également notre modèle :

```
from modele import Plateau
```

On va maintenant détailler les problématiques purement liées à l'utilisation de Pygame. En premier lieu, nous allons voir l'initialisation de l'affichage et de l'utilitaire de gestion du temps :

```
class Jeu(object):
    def __init__(self):
        pygame.init()
        self.clock = pygame.time.Clock()
        self.surface = pygame.display.set_
        mode(TAILLE_FENETRE)
        self.fonts = {
            'defaut': pygame.font.
            Font('freesansbold.ttf', 18),
            'titre': pygame.font.
            Font('freesansbold.ttf', 72),
        }
        pygame.display.set_
caption('Application Bataille Navale')
```

Cette dernière ligne permet d'afficher un texte dans la barre de titre de la fenêtre de l'application. Ces étapes sont indispensables. L'attribut `clock` désigne l'objet qui va permettre de gérer l'écoulement du temps alors que l'objet `surface` désigne le lieu où tous les objets à afficher seront positionnés et rendus pour être affichés à l'utilisateur.

On veut maintenant gérer le début et la fin du jeu et afficher en chaque occasion un petit texte de manière à présenter un écran de bienvenue et un écran de fin. Pour cela, on fait deux méthodes de haut niveau utilisant des méthodes privées de bas niveau pour l'affichage du texte, la mise en attente et la demande de quitter le jeu :

```
def start(self):
    self._afficherTexte(u'Bataille Navale', CENTRE_FENETRE,
    font='titre')
    self._afficherTexte(u'Appuyer sur une touche...', POS)
    self._attente()
def stop(self):
    self._afficherTexte(u'Terminé.', CENTRE_FENETRE,
    font='titre')
    self._afficherTexte(u'Appuyer sur une touche...', POS)
    self._attente()
    self._quitter()
```

Maintenant, il est temps de voir directement chacune des trois méthodes de bas niveau utilisées ici. Commençons par la gestion de l'écriture d'un texte. Pour ma part, je fais passer toutes les écritures de texte par une seule et unique méthode qui prend une signature correspondant à tous mes besoins (qui restent assez basiques) :

```
def _afficherTexte(self, text, position, couleur=9,
font='defaut'):
    font = self.fonts.get(font, self.fonts['defaut'])
    couleur = COULEURS.get(couleur, COULEURS[9])
    rendu = font.render(text, True, couleur)
    rect = rendu.get_rect()
    rect.center = position
    self.surface.blit(rendu, rect)
```

En premier lieu, il faut utiliser l'une des deux polices initialisées plus haut, puis définir la couleur. Ce qui est important à voir est que l'on réalise le rendu à partir de l'objet police d'écriture. Dans un second temps, on positionne le texte rendu, puis on l'ajoute à notre surface à afficher.

Voici la fonctionnalité d'attente :

```
def _attente(self):
    print("Attente")
    while self._getEvent() == None:
        self._rendre()
```

Concrètement, tant que l'on ne capture pas un événement, on réalise une boucle infinie en effectuant un rendu. Voici comment on fait ce rendu (ce qui est fait lors d'une boucle d'attente, mais également lors de la boucle traditionnelle d'affichage lors du déroulé du jeu) :

```
def _rendre(self):
    pygame.display.update()
    self.clock.tick()
```

Dans un premier temps, on met l'affichage à jour, puis on utilise l'horloge afin de ne pas réaliser de boucle infinie, mais de faire boucler l'algorithme dans des temps humains. Concrètement, on demande au programme de passer la main à l'OS tant qu'un délai n'est pas respecté. Ceci permet de rendre le jeu moins gourmand en ressources.

D'autre part, les temps sont pris entre deux appels à tick et sont stables, ce qui permet de faire en sorte de donner une régularité au déroulement du jeu, quelle que soit la complexité des algorithmes utilisés, à partir du moment où une boucle peut être parcourue dans des temps inférieurs à la durée entre deux ticks.

Voici maintenant l'autre méthode centrale qui est la gestion des événements :

```
def _getEvent(self):
    for event in pygame.event.get():
        if event.type == QUIT:
            self._quitter()
        if event.type == KEYUP:
            if event.key == K_ESCAPE:
                self._quitter()
        if event.type == KEYDOWN:
            if event.key == K_ESCAPE:
                continue
            return event.key
        if event.type == MOUSEBUTTONDOWN:
            return "Clic!"
```

Concrètement, si l'on clique sur la croix pour fermer le programme, si l'on relâche la touche d'échappement, le programme appelle la méthode permettant de quitter proprement. Dans tous les autres cas, il renvoie les événements de pression sur les autres touches ou une chaîne de caractères particulière pour signaler un clic de souris.

Voici la méthode permettant de terminer le programme :

```
def _quitter(self):
    print("Quitter")
    pygame.quit()
    sys.exit()
```

On quitte les processus Pygame, puis le programme. Voici enfin la méthode qui va permettre d'ajouter la logique interne au jeu.

Pour l'instant, on ne fait rien :

```
def play(self):
    print("Not Yet Implemented")
    self.surface.fill(COULEURS.get(0))
```

Dans l'avenir, il faudra ajouter la gestion de l'affichage du plateau de jeu, des clics de souris, l'affichage du score et toute la logique permettant le gameplay.

Voici pour terminer le programme, la manière de lancer le jeu :

```
if __name__ == '__main__':
        j = Jeu()
        print("Jeu prêt")
        j.start()
        print("Partie démarée")
        j.play()
        print("Partie terminée")
        j.stop()
        print("Arrêt du programme")
```

3 Écriture de la logique du jeu

Cette dernière partie est le cœur du jeu et n'est réalisable qu'une fois ce qui précède écrit et testé. Contrairement à ce que l'on pourrait penser au premier abord, ce n'est clairement pas le plus difficile. Pour commencer, on va gérer l'affichage du plateau de jeu. Voici les constantes (définies dans `constantes.py`) qui vont permettre de gérer cela :

```
TAILLE_FENETRE = 640, 480
DIM_PLATEAU = 10, 10
BORDURE_PLATEAU = 4
TAILLE_BLOC = 20, 20

TAILLE_PLATEAU = tuple([DIM_PLATEAU[i]*TAILLE_BLOC[i] for i in range(2)])
TAILLE_PLABORD = tuple([DIM_PLATEAU[i]*TAILLE_BLOC[i]+BORDURE_PLATEAU*2
for i in range(2)])

MARGE = tuple([TAILLE_FENETRE[i]-TAILLE_PLATEAU[i]- BORDURE_PLATEAU*2 for
i in range(2)])
START_PLATEAU = int(MARGE[0]/2), MARGE[1]/2+2*BORDURE_PLATEAU
START_PLABORD = int(MARGE[0]/2)-BORDURE_PLATEAU, MARGE[1]/2+BORDURE_
PLATEAU

CENTRE_FENETRE = tuple([TAILLE_FENETRE[i]/2 for i in range(2)])
POS = CENTRE_FENETRE[0], CENTRE_FENETRE[1]+100
```

```
COULEURS = {
          0: (0, 0, 0),
          1: (167, 103, 38),
          2: (145, 40, 59),
          3: (169, 234, 254),
          4: (30, 127, 203),
          5: (255, 255, 255),
          6: (255, 255, 255),
          7: (255, 255, 255),
          8: (255, 255, 255),
          9: (255, 255, 255),
}

ALIGNEMENT_COLONNE_DROITE = TAILLE_FENETRE[0] - START_PLABORD[0] / 2
POSITION_JOUES =  ALIGNEMENT_COLONNE_DROITE, 120
POSITION_TOUCHES = ALIGNEMENT_COLONNE_DROITE, 150
POSITION_COULES = ALIGNEMENT_COLONNE_DROITE, 180

L, C = '0123456789', 'ABCDEFGHIJ'
```

Pour simuler un affichage quelconque, je vais simuler le fait d'avoir joué quelques coups et écrire la méthode d'affichage :

```
    def play(self):
        print("Beeing Implemented")
        self.plateau = Plateau()
        for i in range(25):
            self.plateau.jouer(random.choice(list(self.plateau.table)))
        self.surface.fill(COULEURS.get(0))
        self._dessinerPlateau()

    def _dessinerPlateau(self):
        self.surface.fill(COULEURS.get(0))
        pygame.draw.rect(self.surface, COULEURS[1], START_
        PLABORD+TAILLE_PLABORD, BORDURE_PLATEAU)
        for i, ligne in enumerate(L):
            for j, colonne in enumerate(C):
                if (colonne, ligne) in self.plateau.deja_joues:
                    if (colonne, ligne) in self.plateau.occupes:
                        couleur = COULEURS[2]
                    else:
                        couleur = COULEURS[3]
                else:
                    couleur = COULEURS[4]
                position = j, i
                coordonnees = tuple([START_PLATEAU[k] + position[k] *
                TAILLE_BLOC[k] for k in range(2)])
                pygame.draw.rect(self.surface, couleur, coordonnees +
                TAILLE_BLOC)
        self._afficherTexte(u'Joués: >%s' % self.plateau.nb_joues(),
        POSITION_JOUES)
        self._afficherTexte(u'Touchés: %s' % self.plateau.nb_touches(),
        POSITION_TOUCHES)
        self._afficherTexte(u'Coulés: %s' % self.plateau.nb_coules(),
        POSITION_COULES)

        self._rendre()
```

D'un côté purement technique, il faut noter l'utilisation de `enumerate` qui permet de donner à chaque chiffre (représentant les lignes) ou lettre (représentant les colonnes) un

indice sur lequel on s'appuiera pour savoir où le situer sur le plateau de jeu. On n'itère pas sur l'ensemble lui-même, puisque ce dernier n'a pas de relation d'ordre, mais directement sur le nom des lignes et des colonnes. Après, il ne reste plus qu'à calculer la coordonnée de chaque case à l'aide des indices puis de faire un carré à colorier.

On affiche également trois textes permettant de savoir combien de coups ont été joués, combien de touches ont été réalisées et combien de bateaux ont été coulés.

Pour le reste, il faut maintenant s'occuper du gameplay. On va utiliser la souris pour permettre de cliquer sur les cases. L'idée est de récupérer la case à partir de la position de la souris. Voici une astuce qui permettra de disposer de cette position, en ajoutant cette ligne juste avant l'appel à la méthode `_rendre` dans le code qui précède :

```
self._afficherTexte(u'Coulés: %s' % str(pygame.mouse.get_pos()),
POSITION_TEST)
```

Tout réside maintenant dans le calcul de la case à partir de la position de la souris :

```
def _get_selected_case(self):
    x, y = pygame.mouse.get_pos()
    x = (x - START_PLATEAU[0]) // TAILLE_BLOC[0]
    y = (y - START_PLATEAU[1]) // TAILLE_BLOC[1]
    return y, x
```

Et voici la logique de jeu (remplacement de la méthode précédemment écrite par celle-ci) :

```
def play(self):
    self.plateau = Plateau()
    while not self.plateau.partie_est_gagnee():
        self.surface.fill(COULEURS.get(0))
        self._dessinerPlateau()
        while True:
            if self._getEvent() == "Clic!":
                l, c = self._get_selected_case()
    v       if 0 <= l < 10 and 0 <= c < 10: break
        self.plateau.jouer((C[c], L[l]))
    self.surface.fill(COULEURS.get(0))
    self._dessinerPlateau()
```

L'idée est simplement de boucler tant que la partie n'est pas gagnée. Dans cette boucle, on dessine le plateau, puis on fait une boucle infinie sur les événements pour capturer explicitement les clics de souris. Lorsqu'un des clics se produit, on calcule la coordonnée du carré du plateau choisi et si elle est valide, on joue le coup correspondant et on refait l'affichage.

Une fois que l'on a gagné, on refait l'affichage pour afficher le dernier coup joué.

Il est à noter que la méthode `_getEvent` gère le fait de quitter l'application de manière autonome et que tout l'affichage est géré par la méthode `_dessinerPlateau` déjà vue. On a donc fait l'essentiel du travail.

Conclusion

Cet article présente une courte introduction à Pygame. Nous n'avons vu que la partie émergée de Pygame et il reste encore beaucoup à explorer (entre autres les *sprites*, le son). Nous n'avons clairement pas développé le jeu ultime, mais en moins de 200 lignes, nous avons un programme qui est ultra-lisible, très simple et qui gère un petit gameplay, un affichage minimaliste et qui fait l'essentiel. Ce qu'il reste à faire est une bonne intelligence artificielle pour placer automatiquement les bateaux.

Après, il ne reste plus qu'à donner sa propre touche au jeu en sachant bien utiliser les graphismes (ceux présentés ici sont inexistants, mais un bon commercial dirait « vintage »), les sons, ajouter des animations, compléter le gameplay en se montrant original sur les scénarios de jeu, voire jouer sur le nombre de bateaux, leur longueur et la taille du plateau.

Toujours est-il que l'on a pu mettre un pied dans Pygame sans avoir à apprendre des concepts trop complexes, sans écrire du code trop inaccessible. Ce code sera mis à disposition sur le site inspyration.org. Il vous suffira de le télécharger pour faire vos propres expériences et vous approprier ce qui a été vu.

Au-delà de cela, on a vu des éléments importants sur la manipulation de données avec Python. Pour s'en convaincre, il suffit d'essayer de refaire toute la première partie en n'utilisant que des listes. On verra alors que les algorithmes seront autrement plus complexes.

Ce sont entre autres ces types de données qui rendent ce langage extrêmement performant en termes de temps de développement. ■

À propos de l'auteur

Sébastien Chazallet, Ingénieur logiciels libres et auteur de « Python, les fondamentaux du langage » aux éditions ENI.