Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Plongez au coeur de Python

11 février 2006

Chapitre 1. Installation de Python

Bienvenue dans le monde de Python. Préparez−vous à plonger. Dans ce chapitre, nous allons installer une version de Python appropriée à votre situation.

1.1. Quel Python vous faut−il ?

La première chose à faire avec Python est de l'installer. Mais est−ce nécéssaire ?

Si vous utilisez un compte sur un serveur hébergé, votre FAI a peut−être déjà installé Python. La plupart des distributions Linux les plus courantes installent Python par défaut. Mac OS X 10.2 et les versions suivantes comprennent une version en ligne de commande de Python, mais vous souhaiterez sans doute installer une version ayant une interface graphique plus typique du Mac.

Windows n'inclut aucune version de Python, mais ne désespérez pas ! Il y a de nombreuses manières d'installer Python sous Windows.

Comme vous pouvez le constater, Python supporte un grand nombre de systèmes d'exploitation. La liste complète comprend Windows, Mac OS, Mac OS X et tous les systèmes libres compatibles UNIX comme Linux. Il y a aussi des versions pour Sun Solaris, AS/400, Amiga, OS/2, BeOS et une pléthore d'autres plate−formes dont vous n'avez sans doute jamais entendu parler.

De plus, les programmes Python écrits sur une plate−forme peuvent, en prenant en compte certain détails, être exécutés sur *toutes* les plate−formes supportées. Par exemple, je développe régulièrement des programmes Python sous Windows pour les déployer ensuite sous Linux.

Mais revenons à la question de départ, "Quel Python vous faut−il ?" Celle qui fonctionne sur la plate−forme que vous utilisez.

1.2. Python sous Windows

Sous Windows, il y a plusieurs possibilités pour installer Python.

ActiveState propose un programme d'installation pour Windows appelé ActivePython, qui comprend une version complète de Python, un IDE doté d'un éditeur de code prenant en compte Python et quelques extensions spécifiques à Windows pour Python qui permettent d'accèder aux services et aux APIs propres à Windows, ainsi qu'à la Base de registre.

ActivePython est librement téléchargeable, bien qu'il ne soit pas open source. C'est l'IDE que j'ai utilisé pour apprendre Python et je vous recommande de l'essayer, à moins que vous n'ayez une raison spécifique de ne pas le faire. Une de ces raisons pourrait être qu'ActiveState a généralement plusieurs mois de retard quand une nouvelle version de Python est publiée. Si vous avez absolument besoin de la dernière version de Python et qu'ActivePython a une version de retard, vous pourrez choisir la deuxième option pour installer Python sous Windows.

La deuxième option est le programme d'installation "officiel" de Python, distribué par les gens qui développent Python. Il est librement téléchargeable et open source et il installe toujours la version la plus récente de Python.

Procédure 1.1. Option 1 : installer ActivePython

Voici la procédure pour installer ActivePython :

1. Téléchargez ActivePython depuis http://www.activestate.com/ASPN/Downloads/ActivePython/.
2. Si vous utilisez Windows 95, Windows 98 ou Windows ME, vous aurez également besoin de télécharger et d'installer Windows Installer 2.0 (http://download.microsoft.com/download/WindowsInstaller/Install/2.0/W9XMe/EN-US/InstMsiA.exe) avant d'installer ActivePython.
3. Double-cliquez sur le programme d'installation, `ActivePython-2.2.2-224-win32-ix86.msi`.
4. Suivez les étapes du programme d'installation.
5. Si vous manquez d'espace disque, vous pouvez sélectionner l'installation personnalisée et déselectionner la documentation, mais je ne vous le recommande pas, à moins que vous ayez vraiment besoin des 14 Mo.
6. Lorsque l'installation est terminée, fermez le programme d'installation et lancez Démarrer->Programmes->ActiveState ActivePython 2.2->PythonWin IDE. Vous verrez quelque chose comme l'écran suivant :

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994-2001 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>>
```

Procédure 1.2. Option 2 : installer Python depuis Python.org (http://www.python.org/)

1. Téléchargez la dernière version du programme d'installation Python pour Windows depuis http://www.python.org/ftp/python/2.3.3/ en sélectionnant le numéro de version le plus haut et en téléchargeant le programme `.exe`.
2. Double-cliquez sur le programme d'installation, `Python-2.xxx.yyy.exe`. Le nom dépend de la version de Python disponible au moment du téléchargement.
3. Suivez les étapes du programme d'installation.
4. Si vous manquez d'espace disque, vous pouvez déselectionner le fichier d'aide HTMLHelp, les scripts utilitaires (`Tools/`) et/ou la suite de tests (`Lib/test/`).
5. Si vous n'avez pas les droits d'administrateur de la machine, vous pouvez sélectionner Advanced Options, puis choisir Non-Admin Install. Cette option modifie uniquement l'emplacement des entrées de la Base de registre et des raccourcis du menu Démarrer.
6. Après l'achèvement de l'installation, fermez le programme d'installation et lancez Démarrer->Programmes->Python 2.3->IDLE (Python GUI). Vous verrez quelque chose comme l'écran suivant :

```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0
>>>
```

1.3. Python sous Mac OS X

Sous Mac OS X, vous avez deux possibilités, garder la version préinstallée ou installer une nouvelle version. Vous préfèrerez sans doute cette dernière solution.

Mac OS X 10.2 et les version ultérieures contiennent une version en ligne de commande de Python. Si vous êtes à l'aise avec la ligne de commande, vous pouvez utiliser cette version pour le premier tiers du livre. Par contre, la

version préinstallée ne contient pas d'analyseur pour le XML, vous devrez donc installer la version complète lorsque vous arriverez au chapitre traitant du XML.

Installer la dernière version présente également l'avantage d'une interface graphique interactive.

Procédure 1.3. Exécuter la version préinstallée de Python sous Mac OS X

Pour utiliser la version préinstallée de Python, suivez ces étapes :

1. Ouvrez le dossier `/Applications`.
2. Ouvrez le dossier `Utilitaires`.
3. Double−cliquez sur `Terminal` pour ouvrir une fenêtre de terminal et accéder à la ligne de commande.
4. Tapez **`python`** sur la ligne de commande.

Essayez :

```
Welcome to Darwin!
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp−precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

Procédure 1.4. Installation de la dernière version de Python sous Mac OS X

Suivez ces étapes pour télécharger et installer la dernière version de Python:

1. Téléchargez l'image disque `MacPython−OSX` depuis http://www.cwi.nl/~jack/macpython.html.

   Si votre navigateur n'a pas monté l'image, double−cliquez sur `MacPython−OSX−2.3−1.dmg` pour la monter sur le bureau.
2. Double−cliquer sur le programme d'installation, `MacPython−OSX.pkg`.
3. Le programme d'installation vous demandera votre nom et mot de passe d'administrateur.
4. Suivez les étapes du programme d'installation.
5. Lorsque l'installation est terminée, fermez le programme d'installation et ouvrez le dossier `/Applications`.
6. Ouvrez le dossier `MacPython−2.3`.
7. Double−cliquez sur `PythonIDE` pour lancer Python.

L'IDE MacPython devrait afficher un écran d'accueil, puis vous amener à l'interface interactive. Si l'interface n'apparaît pas, sélectionnez Window−>Python Interactive (**Cmd−0**). La fenêtre qui s'ouvre ressemblera à l'écran suivant :

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Notez qu'une fois que vous installez la dernière version, la version préinstallée est toujours présente. Si vous exécutez des scripts depuis la ligne de commande, vous devez savoir quelle version de Python vous utilisez.

Exemple 1.1. Deux versions de Python

```
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you% /usr/local/bin/python
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

1.4. Python sous Mac OS 9

Mac OS 9 n'est fournit avec aucune version de Python, mais l'installation en est très simple.

Suivez ces étapes pour installer Python sous Mac OS 9 :

1. Téléchargez le fichier `MacPython23full.bin` depuis http://www.cwi.nl/~jack/macpython.html.
2. Si votre navigateur n'a pas décompressé le fichier automatiquement, double-cliquez `MacPython23full.bin` pour le décompresser avec Stuffit Expander.
3. Double-cliquez sur le programme d'installation, `MacPython23full`.
4. Suivez les étapes du programme d'installation.
5. Lorsque l'installation est terminée, fermez le programme d'installation et ouvrez le dossier `/Applications`.
6. Ouvrez le dossier `MacPython-OS9 2.3`.
7. Double-cliquez `Python IDE` pour lancer Python.

L'IDE MacPython devrait afficher un écran d'accueil, puis vous amener à l'interface interactive. Si l'interface n'apparaît pas, sélectionnez Window->Python Interactive (**Cmd-0**). La fenêtre qui s'ouvre ressemblera à l'écran suivant :

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

1.5. Python sous RedHat Linux

L'installation sous un système d'exploitation compatible UNIX tel que Linux est simple si vous choisissez l'installation d'un paquetage binaire. Des paquetage binaires précompilés sont disponibles pour les distributions Linux les plus répandues. Vous pouvez également compiler à partir des sources.

Téléchargez le dernier RPM Python en allant sur http://www.python.org/ftp/python/2.3.3/ et en sélectionnant le numéro de version le plus haut, puis en sélectionnant le sous-répertoire `rpms/` de cette version. Téléchargez ensuite le RPM ayant le plus haut numéro de version. Vous pouvez l'installer avec la commande **rpm**, comme ci-dessous :

Exemple 1.2. Installation sous RedHat Linux 9

```
localhost:~$ su -
Password: [enter your root password]
```

```
[root@localhost root]# wget http://python.org/ftp/python/2.3/rpms/redhat-9/python2.3-2.3-5pydotorg.i38
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet-stream]
...
[root@localhost root]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
Preparing...                ########################################### [100%]
   1:python2.3              ########################################### [100%]
[root@localhost root]# python          ❶
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# python2.3       ❷
Python 2.3 (#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# which python2.3 ❸
/usr/bin/python2.3
```

❶ Attention ! Taper simplement **`python`** lance l'ancienne version de Python -- celle qui était installée par défaut. Ce n'est pas celle que vous voulez.

❷ Au moment où j'écris, la version la plus récente s'appelle **`python2.3`**. Vous aurez sans doute à changer le chemin à la première ligne du script pour pointer vers une version plus récente.

❸ C'est le chemin complet de la version la plus récente de Python que vous venez d'installer. Utilisez-le sur la ligne `#!` (la première ligne de chaque script) pour vous assurez que les scripts utilisent la dernière version de Python et faites bien attention de taper **`python2.3`** pour accéder à l'interface interactive.

1.6. Python sous Debian GNU/Linux

Si vous avez la chance d'utiliser Debian GNU/Linux, vous pouvez installer Python à l'aide de la commande **apt**.

Exemple 1.3. Installation sous Debian GNU/Linux

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
```

```
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to exit]
```

1.7. Installation de Python à partir du source

Si vous préférez installer à partir du source, vous pouvez télécharger le code source de Python à partir de http://www.python.org/ftp/python/2.3.3/. Sélectionnez le numéro de version le plus haut de la liste, téléchargez le fichier `.tgz` et faites la séquence habituelle **`configure`**, **`make`**, **`make install`**.

Exemple 1.4. Installation à partir du source

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Modules/python.o Modules/python.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/grammar1.o Parser/grammar1.c
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
localhost:~$
```

1.8. L'interface interactive

Maintenant que Python est installé, nous allons voir en quoi consiste cette interface interactive que vous avez lancé.

Il faut comprendre une chose : Python mène une double vie. C'est un interpréteur de scripts que vous pouvez lancer depuis la ligne de commande, ou en double−cliquant sur un script. Mais c'est aussi une interface interactive qui peut évaluer n'importe quelle instruction ou expression. C'est très utile pour le débogage, pour écrire rapidement du code et pour les tests. Je connais même des gens qui utilisent l'interface interactive de Python comme calculatrice !

Lancez l'interface interactive de Python de la manière qui convient à plate−forme et commençons la plongée par les étapes suivantes :

Exemple 1.5. Premiers pas dans l'interface interactive

```
>>> 1 + 1                ❶
2
>>> print 'hello world'  ❷
hello world
>>> x = 1                ❸
>>> y = 2
>>> x + y
3
```

❶ L'interface interactive de Python peut évaluer une expression Python quelconque, y compris une expression arithmétique de base.

❷ L'interface interactive peut exécuter des instructions Python, y compris l'instruction **print**.

❸ Vous pouvez aussi assigner des valeurs à des variables et les valeurs seront conservées aussi longtemps que l'interface est ouverte (mais pas plus longtemps que cela).

1.9. Résumé

Vous devez maintenant avoir une version de Python installée et fonctionnelle.

En fonction de votre plate−forme, vous pouvez avoir plus d'une version de Python installée. Si c'est le cas, vous devez connaître vos chemins d'accès. Si entrer simplement **python** sur la ligne de commande ne lance pas la version de Python que vous souhaitez utiliser, vous aurez peut être à entrer le chemin complet de votre version préférée.

Félicitations et bienvenue dans le monde de Python.

Chapitre 2. Votre premier programme Python

La plupart des autres livres expliquent pas à pas les concepts de programmation pour vous amener à la fin à l'écriture d'un programme complet et fonctionnel. Et bien nous allons sauter toutes ces étapes.

2.1. Plonger

Voici un programme Python complet et fonctionnel.

Il ne signifie probablement rien pour vous. Ne vous en faites pas, nous allons le disséquer ligne par ligne. Mais lisez−le et voyez ce que vous pouvez déjà en retirer.

Exemple 2.1. `odbchelper.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Lancez maintenant ce programme et observez ce qui se passe.

Dans l'IDE ActivePython sous Windows, vous pouvez exécuter le programme Python que vous êtes en train d'éditer par File−>Run... (**Ctrl−R**). La sortie est affichée dans la fenêtre interactive.

Dans l'IDE Python sous Mac OS, vous pouvez exécuter un module avec Python−>Run window... (**Cmd−R**) mais il y a une option importante que vous devez activer préalablement. Ouvrez le module dans l'IDE, ouvrez le menu des options des modules en cliquant le triangle noir dans le coin supérieur droit de la fenêtre et assurez−vous que "Run as __main__" est coché. Ce réglage est sauvegardé avec le module, vous n'avez donc à faire cette manipulation qu'une fois par module.

Sur les systèmes compatibles UNIX (y compris Mac OS X), vous pouvez exécuter un programme Python depuis la ligne de commande : **`python odbchelper.py`**

La sortie de `odbchelper.py` ressemblera à l'écran suivant :

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

2.2. Déclaration de fonctions

Python dispose de fonctions comme la plupart des autre langages, mais il n'a pas de fichiers d'en−tête séparés comme C++ ou de sections `interface`/`implementation` comme Pascal. Lorsque vous avez besoin d'une fonction, vous n'avez qu'à la déclarer et l'écrire.

```
def buildConnectionString(params):
```

Il y a plusieurs remarques à faire. Premièrement, le mot clé `def` débute une déclaration de fonction, suivi du nom de la fonction, puis des arguments entre parenthèses. Les arguments multiples (non montré ici) sont séparés par des virgules.

Deuxièmement, la fonction ne défini pas le type de données qu'elle retourne. Les fonctions Python ne définissent pas le type de leur valeur de retour, elle ne spécifient même pas si elle retournent une valeur ou pas. En fait chaque fonction Python retourne une valeur, si la fonction exécute une instruction `return`, elle va en retourner la valeur, sinon elle retournera `None`, la valeur nulle en Python.

En Visual Basic, les fonctions (qui retournent une valeur) débutent avec `function` et les sous−routines (qui ne retourne aucune valeur) débutent avec `sub`. Il n'y a pas de sous−routines en Python. Tout est fonction, toute fonction retourne un valeur (même si c'est `None`) et toute fonction débute avec `def`.

Troisièmement, les arguments, `params`, ne spécifient pas de types de données. En Python, les variables ne sont jamais explicitement typées. Python détermine le type d'une variable et en garde la trace en interne.

En Java, C++ et autres langage à typage statique, vous devez spécifier les types de données de la valeur de retour d'une fonction ainsi que de chaque paramètre. En Python, vous ne spécifiez jamais de manière explicite le type de quoi que ce soit. En se basant sur la valeur que vous lui assignez, Python gère les types de données en interne.

2.2.1. Comparaison des types de données en Python et dans d'autres langages de programmation

Un lecteur érudit propose l'explication suivante pour comparer Python et les autres langages de programmation :

langage à typage statique
: Un langage dans lequel les types sont fixés à la compilation. La plupart des langages à typage statique obtiennent cela en exigeant la déclaration de toutes les variables et de leur type avant leur utilisation. Java et C sont des langages à typage statique.

langage à typage dynamique
: Un langage dans lequel les types sont découverts à l'exécution, l'inverse du typage statique. VBScript et Python sont des langages à typage dynamique, ils déterminent le type d'une variable la première fois que vous lui assignez une valeur.

langage fortement typé
: Un langage dans lequel les types sont toujours appliqués. Java et Python sont fortement typés. Un entier ne peut être traité comme une chaîne sans conversion explicite

langage faiblement typé
: Un langage dans lequel les types peuvent être ignorés, l'inverse de fortement typé. VBScript est faiblement typé. En VBScript, vous pouvez concaténer la chaîne `'12'` et l'entier `3` pour obtenir la chaîne `'123'` et traiter le résultat comme l'entier `123`, le tout sans faire de conversion explicite.

Python est donc à la fois *à typage dynamique* (il n'utilise pas de déclaration de type explicite) et *fortement typé* (une fois qu'une variable a un type, cela a une importance).

2.3. Documentation des fonctions

Vous pouvez documenter une fonction Python en lui donnant une chaîne de documentation (`doc string`).

Exemple 2.2. Définition d'une `doc string` pour la fonction `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
```

Les tripes guillemets indiquent une chaîne multi−lignes. Tout ce qu'il y a entre l'ouverture et la fermeture des guillemets fait partie de la chaîne, y compris les retours chariot et les autres guillemets. On peut les utiliser partout, mais vous les verrez le plus souvent utilisées pour définir une `doc string`.

Les triples guillemets sont aussi un moyen simple de définir une chaîne contenant à la fois des guillemets simples et doubles, comme `qq/.../` en Perl.

Tout ce qui se trouve entre les triples guillemets fait partie de la `doc string` de la fonction, qui décrit ce que fait la fonction. Une `doc string`, si elle existe, doit être la première chose déclarée dans une fonction (la première chose après les deux points). Techniquement parlant, vous n'êtes pas obligés de donner une `doc string` à votre fonction, mais vous devriez toujours le faire. Je sais que vous avez entendu cela à tous les cours de programmation auxquels vous avez assisté mais Python vous donne une motivation supplémentaire : la `doc string` est disponible à l'exécution en tant qu'attribut de fonction.

Beaucoup d'IDE Python utilisent les `doc string` pour fournir une documentation contextuelle, ainsi lorsque vous tapez le nom d'une fonction, sa `doc string` apparaît dans une bulle d'aide. Cela peut être incroyablement utile, mais cette utilité est liée à la qualité de votre `doc string`.

Pour en savoir plus sur la documentation des fonctions

- La PEP 257 (http://www.python.org/peps/pep−0257.html) définit les conventions pour les `doc string`.
- Le *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) explique la manière d'écrire de bonnes `doc string`.
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite des conventions d'espacement dans les `doc string` (http://www.python.org/doc/current/tut/node6.html#SECTION006750000000000000000).

2.4. Tout est objet

Au cas ou vous ne l'auriez pas noté, je viens de dire que les fonctions Python ont des attributs et que ces attributs étaient disponibles au moment de l'exécution.

Une fonction, comme tout le reste en Python, est un objet.

Ouvrez votre IDE Python favorite et suivez ces étapes :

Exemple 2.3. Accéder à la `doc string` de la fonction `buildConnectionString`

```
>>> import odbchelper                                    ❶
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
```

```
>>> print odbchelper.buildConnectionString(params) ❷
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__ ❸
Build a connection string from a dictionary

Returns string.
```

❶ La première ligne importe le programme `odbchelper` comme module -- un morceau de code qui peut être utilisé interactivement ou depuis un programme Python (vous verrez des exemples de programmes Python multimodules au Chapitre 4). Une fois que vous importez un module, vous pouvez référencer chacune de ses fonctions, classes ou attributs publics. Les modules peuvent faire cela pour accéder aux fonctionnalités offertes par d'autres modules et vous pouvez le faire dans l'IDE également. C'est un concept important et nous allons en discuter plus amplement plus tard.

❷ Quand vous souhaitez utiliser des fonctions définies dans un module importé, vous devez inclure le nom du module. Vous ne pouvez donc pas dire `buildConnectionString`, ce doit être `odbchelper.buildConnectionString`. Si vous avez utilisé des classes en Java, cela devrait vous sembler vaguement familier.

❸ Plutôt que d'appeler la fonction comme vous l'auriez attendu, nous demandons un des attributs de la fonction, `__doc__`.

`import` en Python est similaire à `require` en Perl. Une fois que vous importez un module Python, vous accédez à ses fonctions avec *`module.function`*. Une fois que vous incluez un module Perl, vous accédez à ses fonctions avec *`module::function`*.

2.4.1. Le chemin de recherche d'import

Avant d'aller plus loin, je veux mentionner rapidement le chemin de recherche de bibliothèques. Python cherche dans plusieurs endroits lorsque vous essayez d'importer un module. Plus précisément, il regarde dans tous les répertoires définis dans `sys.path`. C'est une simple liste et vous pouvez facilement la voir ou la modifier à l'aide des méthodes standard de listes (nous en apprendrons plus sur les listes plus loin dans ce chapitre).

Exemple 2.4. Chemin de recherche d'import

```
>>> import sys                      ❶
>>> sys.path                        ❷
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/site-packages/piddle']
>>> sys                             ❸
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path') ❹
```

❶ Importer le module `sys` rend toutes ses fonctions et attributs disponibles.

❷ `sys.path` est une liste de répertoires qui constitue le chemin de recherche actuel (le votre sera différent en fonction de votre système d'exploitation, la version de Python que vous utilisez et l'endroit où vous l'avez installé). Python recherchera dans ces repertoires (dans l'ordre donné) un fichier `.py` portant le nom de module que vous tentez d'importer.

❸ En fait j'ai menti, la réalité est plus compliquée que ça car tous les modules ne sont pas dans des fichiers `.py`. Certains, comme le module `sys`, sont des modules intégrés, il sont inclus dans Python lui-même. Les modules intégrés se comportent comme des modules ordinaires, mais leur code source Python n'est pas disponible car il ne sont pas écrits en Python (le module `sys` est écrit en C).

❹ Vous pouvez ajouter un nouveau répertoire au chemin de recherche de Python en le joignant à `sys.path` et Python cherchera dans ce répertoire également lorsque vous essayez d'importer un module. Cela dure tant que

Python tourne (nous reparlerons de `append` (joindre) et des autres méthodes de listes au Chapitre 3).

2.4.2. Qu'est−ce qu'un objet ?

En Python, tout est objet et presque tout dispose d'attributs et de méthodes. Toutes les fonctions ont un attribut prédéfini `__doc__` qui retourne la `doc string` définie dans le code source de la fonction. Le module `sys` est un objet qui a (entre autres choses) un attribut appelé `path`. Et ainsi de suite.

Reste la question : qu'est−ce qu'un objet ? Chaque langage de programmation définit le terme "objet" à sa manière. Pour certain, cela signifie que *tout* objet *doit* avoir des attributs et des méthodes, pour d'autres, cela signifie que tout les objets doivent être dérivables. En Python, la définition est plus flexible. Certains objets n'ont ni attributs ni méthodes (nous verrons cela au Chapitre 3) et tous les objets ne sont pas dérivables (voir le Chapitre 5). Mais tout est objet dans le sens où tout peut être assigné à une variable ou passé comme argument à une fonction (voir au Chapitre 4).

Ceci est important et il ne fait aucun mal de le souligner une derniere fois: *en Python tout est objet*. Les chaînes sont des objets. Les listes sont des objets. Les fonctions sont des objets. Même les modules sont des objets.

Pour en savoir plus sur les objets

- La *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique précisémment ce qu'implique de dire que tout est objet en Python (http://www.python.org/doc/current/ref/objects.html), puisque certains pédants aiment discuter longuement de ce genre de choses.
- eff−bot (http://www.effbot.org/guides/) propose un résumé des objets Python (http://www.effbot.org/guides/python−objects.htm).

2.5. Indentation du code

Les fonctions Python n'ont pas de `begin` ou `end` explicites, ni d'accolades qui pourraient marquer là ou commence et ou se termine le code de la fonction. Le seul délimiteur est les deux points (":") et l'indentation du code lui−même.

Exemple 2.5. Indentation de la fonction `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Les blocs de code (fonctions, instructions `if`, boucles `for` ou `while` etc.) sont définis par leur indentation. L'indentation démarre le bloc et la désindendation le termine. Il n'y a pas d'accolades, de crochets ou de mots clés spécifiques. Cela signifie que les espaces blancs sont significatifs et qu'ils doivent être cohérents. Dans cet exemple, le code de la fonction (y compris sa `doc string`) sont indentés de 4 espaces. Cela ne doit pas être forcément 4 espaces, mais il faut que ce soit cohérent. La première ligne non indentée est en dehors de la fonction.

L'Exemple 2.6, «Instructions if» montre un exemple d'indentation du code avec des instructions `if`.

Exemple 2.6. Instructions `if`

```
def fib(n):                     ❶
    print 'n =', n              ❷
    if n > 1:                   ❸
```

```
        return n * fib(n - 1)
    else:                                   ❹
        print 'end of the line'
        return 1
```

❶ Voici une fonction nommée `fib` qui prend un argument, `n`. Tout le code de cette fonction est indenté.

❷ Afficher une sortie à l'écran est très facile en Python, il suffit d'utiliser `print`. Les instructions `print` peuvent prendre n'importe quel type de données, y compris les chaînes, les entiers et d'autres types prédéfinis comme les dictionnaires et les listes, que vous découvrirez dans le prochain chapitre. Vous pouvez même mélanger les types pour imprimer plusieurs éléments sur la même ligne en utilisant une liste de valeurs séparées par des virgules. Chaque valeur est affichée sur la même ligne, séparée par des espaces (les virgules ne sont pas imprimées). Donc, lorsque `fib` est appelé avec `5`, cette ligne affichera "n = 5".

❸ Les instructions `if` sont un type de bloc de code. Si l'expression `if` est évaluée à vrai, le bloc de code indenté est exécuté, sinon on saute au bloc `else`.

❹ Bien sûr, les blocs `if` et `else` peuvent contenir des lignes multiples, tant qu'elles sont toutes indentées au même niveau. Ce bloc `else` contient deux lignes de code. Il n'y a pas d'autre syntaxe pour les blocs de codes multilignes. Indentez et c'est tout.

Après quelques protestations initiales et des analogies méprisantes à Fortran, vous vous en accomoderez et commencerez à en voir les bénéfices. Un des bénéfices majeurs est que tous les programmes Python ont la même apparence puisque l'indentation est une caractéristique du langage et non une question de style. Cela rend plus simple à comprendre le code Python des autres.

Python utilise le retour chariot pour séparer les instructions, deux points et l'indentation pour séparer les blocs de code. C++ et Java utilisent des points−virgules pour séparer les instructions et des accolades pour séparer les blocs de code.

Pour en savoir plus sur l'indentation du code

- La *Python Reference Manual* (http://www.python.org/doc/current/ref/) discute des aspects multiplate−formes de l'indentation et présente diverses erreurs d'indentation (http://www.python.org/doc/current/ref/indentation.html).
- Le *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discute du bon usage de l'indentation.

2.6. Test des modules

Les modules Python sont des objets et ils ont de nombreux attributs utiles. C'est un aspect que vous pouvez utiliser pour tester facilement vos modules au cours de leur écriture. Voici un exemple qui utilise l'astuce `if __name__`.

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

Quelques remarques avant de passer aux choses sérieuses. Premièrement, les parenthèses ne sont pas obligatoires autour de l'expression `if`. Ensuite, l'instruction `if` se termine par deux points et est suivie de code indenté.

A l'instar de C, Python utilise `==` pour la comparaison et `=` pour l'assignement. Mais au contraire de C, Python ne permet pas les assignations dans le corps d'une instruction afin d'éviter qu'une valeur soit accidentellement assignée alors que vous pensiez effectuer une simple comparaison.

En quoi cette instruction `if` est−elle une astuce ? Les modules sont des objets et tous les modules disposent de l'attribut prédéfini `__name__`. Le `__name__` d'un module dépend de la façon dont vous l'utilisez. Si vous importez

le module, son `__name__` est le nom de fichier du module sans le chemin d'accès ni le suffixe. Mais vous pouvez aussi lancer le module directement en tant que programme, dans ce cas `__name__` va prendre par défaut une valeur spéciale, `__main__`.

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Sachant cela, vous pouvez concevoir une suite de tests pour votre module au sein même de ce dernier en la plaçant dans ce `if`. Quand vous lancez le module directement, `__name__` est `__main__` et la séquence de tests s'exécute. Quand vous importez le module, `__name__` est autre chose et les tests sont ignorés. Cela facilite le développement et le déboguage de nouveaux modules avant leur intégration dans un programme plus grand.

Avec MacPython, il y a une étape supplémentaire pour que l'astuce `if __name__` fonctionne. Ouvrez le menu des options des modules en cliquant le triangle noir dans le coin supérieur droit de la fenêtre et assurez−vous que Run as __main__ est coché.

Pour en savoir plus sur l'importation des modules

- La *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique les détails techniques de l'importation de modules (http://www.python.org/doc/current/ref/import.html).

Chapitre 3. Types prédéfinis

Avant de revenir à votre premier programme Python, une petite digression est de rigueur car vous devez absolument connaître les dictionnaires, les tuples et les listes (tout ça !). Si vous être un programmeur Perl, vous pouvez probablement passer rapidemment sur les points concernant les dictionnaires et les listes mais vous devrez quand même faire attention aux tuples.

3.1. Présentation des dictionnaires

Un des types de données fondamentaux de Python est le dictionnaire, qui défini une relation 1 à 1 entre des clés et des valeurs.

En Python, un dictionnaire est comme une table de hachage en Perl. En Perl, les variables qui stockent des tables de hachage débutent toujours par le caractère %. En Python vous pouvez nommer votre variable comme bon vous semble et Python se chargera de la gestion du typage.

Un dictionnaire Python est similaire à une instance de la classe `Hashtable` en Java.

Un dictionnaire Python est similaire à une instance de l'objet `Scripting.Dictionnary` en Visual Basic.

3.1.1. Définition des dictionnaires

Exemple 3.1. Définition d'un dictionnaire

```
>>> d = {"server":"mpilgrim", "database":"master"} ❶
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]                                    ❷
'mpilgrim'
>>> d["database"]                                  ❸
'master'
>>> d["mpilgrim"]                                  ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

❶ D'abord, nous créons un nouveau dictionnaire avec deux éléments que nous assignons à la variable `d`. Chaque élément est une paire clé−valeur et l'ensemble complet des éléments est entouré d'accolades.

❷ `'server'` est une clé et sa valeur associée, référencée par `d["server"]`, est `'mpilgrim'`.

❸ `'database'` est une clé et sa valeur associée, référencée par `d["database"]`, est `'master'`.

❹ Vous pouvez obtenir les valeurs par clé, mais pas les clés à partir de leur valeur. Donc `d["server"]` est `'mpilgrim'`, mais `d["mpilgrim"]` déclenche une exception car `'mpilgrim'` n'est pas une clé.

3.1.2. Modification des dictionnaires

Exemple 3.2. Modification d'un dictionnaire

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" ❶
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
```

```
>>> d["uid"] = "sa"        ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

❶ Vous ne pouvez avoir de clés dupliquées dans un dictionnaire. L'assignation d'une valeur à une clé existante a pour effet d'effacer l'ancienne valeur.

❷ Vous pouvez ajouter de nouvelles paires clé−valeur à tout moment. La syntaxe est identique à celle utilisée pour modifier les valeurs existantes. (Oui, cela vous posera problème si vous essayez d'ajouter de nouvelles valeurs alors que vous ne faites que modifier constamment la même valeur parce que votre clé n'a pas changé de la manière que vous espériez)

Notez que le nouvel élément (clé `'uid'`, valeur `'sa'`) à l'air d'être au milieu. En fait c'est par coïncidence que les éléments avaient l'air d'être dans l'ordre dans le premier exemple, c'est tout autant une coïncidence qu'ils aient l'air dans le désordre maintenant.

Les dictionnaires ne sont liés à aucun concept d'ordonnancement des éléments. Il est incorrect de dire que les élément sont "dans le désordre", il ne sont tout simplement pas ordonnés. C'est une distinction importante qui vous ennuiera lorsque vous souhaiterez accéder aux éléments d'un dictionnaire d'une façon spécifique et reproductible (par exemple par ordre alphabétique des clés). C'est possible, mais cette fonctionalite n'est pas integrée au dictionnaire.

Quand vous utilisez des dictionnaires, vous devez garder à l'esprit le fait que les clés sont sensibles à la casse.

Exemple 3.3. Les clés des dictionnaires sont sensibles à la casse

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value"  ❶
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value"  ❷
>>> d
{'Key': 'third value', 'key': 'other value'}
```

❶ Assigner une valeur a une clé existante remplace l'ancienne valeur par la nouvelle.

❷ Ici la valeur n'est pas assignée à une clé existante parce que les chaînes en Python sont sensibles à la casse, donc `'key'` n'est pas la même chose que `'Key'`. Une nouvelle paire clé/valeur est donc créée dans le dictionnaire, elle peut vous sembler similaire à la précédente, mais pour Python elle est complètement différente.

Exemple 3.4. Mélange de types de données dans un dictionnaire

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3  ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"    ❷
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
```

❶ Les dictionnaires ne servent pas uniquement aux chaînes de caractères. Les valeurs d'un dictionnaire peuvent être de n'importe quel type de données, y compris des chaînes, des entiers, des objets et même d'autres dictionnaires. Au sein d'un même dictionnaire les valeurs ne sont pas forcémment d'un même

type, vous pouvez les mélanger à votre guise.

❷ Les clés d'un dictionnaire sont plus restrictives, mais elles peuvent être des chaînes, des entiers et de quelques autres types encore (nous verrons cela en détail plus tard). Vous pouvez également mélanger divers types de données au sein des clés d'un dictionnaire.

3.1.3. Enlever des éléments d'un dictionnaire

Exemple 3.5. Enlever des éléments d'un dictionnaire

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master',
42: 'douglas', 'retrycount': 3}
>>> del d[42] ❶
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() ❷
>>> d
{}
```

❶ L'instruction `del` vous permet d'effacer des éléments d'un dictionnaire en fonction de leur clé.

❷ La méthode `clear` efface tous les éléments d'un dictionnaire. Notez que l'ensemble fait d'accolades vides signifie un dictionnaire sans éléments.

Pour en savoir plus sur les dictionnaires

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) explique comment utiliser les dictionnaires pour modéliser les matrices creuses (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) a de nombreux exemples de code ayant recours aux dictionnaires (http://www.faqts.com/knowledge−base/index.phtml/fid/541).
- Le Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) explique comment trier les valeurs d'un dictionnaire par leurs clés (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52306).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes des dictionnaires (http://www.python.org/doc/current/lib/typesmapping.html).

3.2. Présentation des listes

Les listes sont le type de données à tout faire de Python. Si votre seule expérience des listes sont les tableaux de Visual Basic ou (à Dieu ne plaise) les datastores de Powerbuilder, accrochez−vous pour les listes Python.

Une liste en Python est comme un tableau Perl. En Perl, les variables qui stockent des tableaux débutent toujours par le caractère `@`, en Python vous pouvez nommer votre variable comme bon vous semble et Python se chargera de la gestion du typage.

Une liste Python est bien plus qu'un tableau en Java (même s'il peut être utilisé comme tel si vous n'attendez vraiment rien de mieux de la vie). Une meilleure analogie serait la classe `ArrayList`, qui peut contenir n'importe quels objets et qui croît dynamiquement au fur et à mesure que de nouveaux éléments y sont ajoutés.

3.2.1. Definition d'une liste

Exemple 3.6. Definition d'une liste

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                       ❷
'a'
>>> li[4]                                       ❸
'example'
```

❶ Premièrement, nous définissons une liste de 5 éléments. Notez qu'ils conservent leur ordre d'origine. Ce n'est pas un accident. Une liste est un ensemble ordonné d'éléments entouré par des crochets.

❷ Une liste peut être utilisée comme un tableau dont l'indice de base est zéro. Le premier élément de toute liste non vide est toujours `li[0]`.

❸ Le dernier élément de cette liste de 5 éléments est `li[4]` car les listes sont toujours indicées à partir de zéro.

Exemple 3.7. Indices de liste négatifs

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] ❶
'example'
>>> li[-3] ❷
'mpilgrim'
```

❶ Un indice négatif permet d'accéder aux éléments à partir de la fin de la liste en comptant à rebours. Le dernier élément de toute liste non vide est toujours `li[-1]`.

❷ Si vous trouvez que les indices négatifs prêtent à confusion, voyez-les comme suit : `li[n] == li[n - len(li)]`. Donc dans cette liste, `li[-3] == li[5 - 3] == li[2]`.

Exemple 3.8. Découpage d'une liste

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  ❶
['b', 'mpilgrim']
>>> li[1:-1] ❷
['b', 'mpilgrim', 'z']
>>> li[0:3]  ❸
['a', 'b', 'mpilgrim']
```

❶ Vous pouvez obtenir un sous-ensemble d'une liste, appelé une "tranche" (*slice*), en spécifiant deux indices. La valeur de retour est une nouvelle liste contenant les éléments de la liste, dans l'ordre, en démarrant du premier indice de la tranche (dans ce cas `li[1]`), jusqu'à au second indice de la tranche non inclu (ici `li[3]`).

❷ Le découpage fonctionne si un ou les deux indices sont négatifs. Pour vous aider, vous pouvez les voir commer ceci : en lisant la liste de gauche à droite, le premier indice spécifie le premier élément que vous désirez et le second indice spécifie le premier élément dont vous ne voulez pas. La valeur de retour est tout ce qui se trouve entre les deux.

❸ Les listes sont indicées à partir de zéro, donc `li[0:3]` retourne les trois premiers éléments de la liste, en démarrant à `li[0]` jusqu'à `li[3]` non inclu.

Exemple 3.9. Raccourci pour le découpage

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] ❶
['a', 'b', 'mpilgrim']
```

```
>>> li[3:] ❷ ❸
['z', 'example']
>>> li[:]  ❹
['a', 'b', 'mpilgrim', 'z', 'example']
```

❶ Si l'indice de tranche de gauche est 0, vous pouvez l'omettre et 0 sera implicite. Donc `li[:3]` est la même chose que `li[0:3]` dans le premier exemple.

❷ De la même manière, si l'indice de tranche de droite est la longueur de la liste, vous pouvez l'omettre. Donc `li[3:]` est pareil que `li[3:5]`, puisque la liste a 5 éléments.

❸ Remarquez la symétrie. Dans cette liste de 5 éléments, `li[:3]` retourne les 3 premiers éléments et `li[3:]` retourne les deux derniers. En fait `li[:n]` retournera toujours les `n` premiers éléments et `li[n:]` le reste, quelle que soit la longueur de la liste.

❹ Si les deux indices sont omis, tous les éléments de la liste sont inclus dans la tranche. Mais ce n'est pas la même chose que la liste `li`; c'est une nouvelle liste qui contient les même éléments. `li[:]` est un raccourci permettant d'effectuer une copie complète de la liste.

3.2.2. Ajout d'éléments à une liste

Exemple 3.10. Ajout d'éléments à une liste

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")               ❶
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")            ❷
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) ❸
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

❶ `append` ajoute un élément à la fin de la liste.

❷ `insert` insère un élément dans la liste. L'argument numérique est l'indice du premier élément qui sera décalé. Notez que les éléments de la liste ne sont pas obligatoirement uniques ; il y a maintenant 2 éléments distincts avec la valeur `'new'`, `li[2]` and `li[6]`.

❸ `extend` concatène des listes. Notez que vous n'appelez pas `extend` avec plusieurs arguments mais bien avec un seul argument qui est une liste. Dans le cas présent, la liste est composée de deux éléments.

Exemple 3.11. Différence entre `extend` et `append`

```
>>> li = ['a', 'b', 'c']
>>> li.extend(['d', 'e', 'f']) ❶
>>> li
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(li)                    ❷
6
>>> li[-1]
'f'
>>> li = ['a', 'b', 'c']
>>> li.append(['d', 'e', 'f']) ❸
>>> li
['a', 'b', 'c', ['d', 'e', 'f']]
>>> len(li)                    ❹
4
>>> li[-1]
```

```
['d', 'e', 'f']
```

❶ Les listes ont deux méthodes, `extend` et `append`, qui semblent faire la même chose, mais sont en fait complètement différentes. `extend` prend un seul argument, qui est toujours une liste et ajoute chacun des éléments de cette liste à la liste originelle.

❷ Ici nous avons une liste de trois éléments (`'a'`, `'b'` et `'c'`) et nous utilisons `extended` pour lui ajouter une liste de trois autres éléments (`'d'`, `'e'` et `'f'`), ce qui nous donne une liste de six éléments.

❸ Par contre, `append` prend un argument, qui peut être de n'importe quel type et l'ajoute simplement à la fin de la liste. Ici, nous appelons `append` avec un argument, qui est une liste de trois éléments.

❹ Maintenant, la liste originelle qui avait trois éléments en contient quatre. Pourquoi quatre ? Parce que le dernier élément que nous venons d'ajouter *est lui−même une liste*. Les listes peuvent contenir n'importe quel type de données, y compris d'autres listes. En fonction du but recherché, faites attention de ne pas utiliser `append` si vous pensez en fait à `extend`.

3.2.3. Recherche dans une liste

Exemple 3.12. Recherche dans une liste

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") ❶
5
>>> li.index("new")     ❷
2
>>> li.index("c")       ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           ❹
False
```

❶ `index` trouve la première occurrence d'une valeur dans la liste et retourne son indice.

❷ `index` trouve la *première* occurrence d'une valeur dans la liste. Dans ce cas, new apparaît à deux reprises dans la liste, `li[2]` et `li[6]`, mais `index` ne retourne que le premier indice, `2`.

❸ Si la valeur est introuvable dans la liste, Python déclenche une exception. C'est sensiblement différent de la plupart des autres langages qui retournent un indice invalide. Si cela peut sembler gênant, c'est en fait une bonne chose car cela signifie que votre programme se plantera à la source même du problème plutôt qu'au moment ou vous tenterez de manipuler l'indice non valide.

❹ Pour tester la présence d'une valeur dans la liste, utilisez `in`, qui retourne `True` si la valeur a été trouvée et `False` dans le cas contraire.

Avant la version 2.2.1, Python n'avait pas de type booléen. Pour compenser cela, Python acceptait pratiquement n'importe quoi dans un contexte requérant un booléen (comme une instruction `if`), en fonction des règles suivantes :

- `0` est faux, tous les autres nombres sont vrai.
- Une chaîne vide (`""`) est faux, toutes les autres chaînes sont vrai.
- Une liste vide (`[]`) est faux, toutes les autres listes sont vrai.
- Un tuple vide (`()`) est faux, tous les autres tuples sont vrai.
- Un dictionnaire vide (`{}`) est faux, tous les autres dictionnaires sont vrai.

Ces règles sont toujours valides en Python 2.3.3 et au−delà, mais vous pouvez maintenant utiliser un véritable booléen, qui a pour valeur `True` ou `False`. Notez la majuscule, ces valeurs comme tout le reste en Python, sont sensibles à la casse.

3.2.4. Suppression d'éléments d'une liste

Exemple 3.13. Enlever des éléments d'une liste

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")   ❶
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")   ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()         ❹
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

❶ `remove` enlève la première occurrence de la valeur de la liste.

❷ `remove` enlève *uniquement* la première occurence de la valeur. Dans ce cas, `new` apparaît à deux reprises dans la liste mais `li.remove("new")` a seulement retiré la première occurrence.

❸ Si la valeur est introuvable dans la liste, Python déclenche une exception. Ce comportement est identique à celui de la méthode `index`.

❹ `pop` est un spécimen intéressant. Il fait deux choses : il enlève le dernier élément de la liste et il retourne la valeur qui a été enlevé. Notez que cela diffère de `li[-1]` qui retourne une valeur mais ne modifie pas la liste et de `li.remove(valeur)` qui altère la liste mais ne retourne pas de valeur.

3.2.5. Utilisation des opérateurs de listes

Exemple 3.14. Opérateurs de listes

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] ❶
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']                ❷
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3              ❸
>>> li
[1, 2, 1, 2, 1, 2]
```

❶ Les listes peuvent être concaténées à l'aide de l'opérateur `+`. `liste = liste + autreliste` est équivalent à `list.extend(autreliste)`. Mais l'opérateur `+` retourne une nouvelle liste concaténée comme une valeur alors que extend modifie une liste existante. Cela implique que `extend` est plus rapide, surtout pour de grandes listes.

❷ Python supporte l'opérateur `+=`. `li += ['two']` est équivalent à `li = li + ['two']`. L'opérateur `+=` fonctionne pour les listes, les chaînes et les entiers. Il peut être surchargé pour fonctionner également avec des classes définies par l'utilisateur (nous en apprendrons plus sur les classes au Chapitre 5).

❸ L'opérateur `*` agit sur les liste comme un répéteur. `li = [1, 2] * 3` est équivalent à `li = [1, 2] + [1, 2] + [1, 2]`, qui concatène les trois listes en une seule.

Pour en savoir plus sur les listes

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) explique les listes et expose le sujet important du passage de listes comme arguments de fonction (http://www.ibiblio.org/obp/thinkCSpy/chap08.htm).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) montre comment utiliser des listes comme des piles ou des files (http://www.python.org/doc/current/tut/node7.html#SECTION00711000000000000000).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions courantes à propos des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/534) et fourni de nombreux exemples de code utilisant des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/540).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes des listes (http://www.python.org/doc/current/lib/typesseq−mutable.html).

3.3. Présentation des tuples

Un *tuple* (n−uplet) est une liste non−mutable. Un fois créé, un tuple ne peut en aucune manière être modifié.

Exemple 3.15. Définition d'un tuple

```
>>> t = ("a", "b", "mpilgrim", "z", "example") ❶
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                       ❷
'a'
>>> t[−1]                                      ❸
'example'
>>> t[1:3]                                     ❹
('b', 'mpilgrim')
```

❶ Un tuple est défini de la même manière qu'une liste sauf que l'ensemble d'éléments est entouré de parenthèses plutôt que de crochets.

❷ Les éléments d'un tuple ont un ordre défini, tout comme ceux d'une liste. Les indices de tuples débutent à zéro, tout comme ceux d'une liste, le premier élément d'un tuple non vide est toujours `t[0]`.

❸ Les indices négatifs comptent à partir du dernier élément du tuple, tout comme pour une liste.

❹ Le découpage fonctionne aussi, tout comme pour une liste. Notez que lorsque vous découpez une liste, vous obtenez une nouvelle liste, lorsque vous découpez un tuple, vous obtenez un nouveau tuple.

Exemple 3.16. Les tuples n'ont pas de méthodes

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")     ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")       ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
```

```
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           ❹
True
```

❶ Vous ne pouvez pas ajouter d'élément à un tuple. Les tuples n'ont pas de méthodes `append` ou `extend`.

❷ Vous ne pouvez pas enlever d'éléments d'un tuple. Les tuples n'ont pas de méthodes `remove` ou `pop`.

❸ Vous ne pouvez pas rechercher d'éléments dans un tuple. Les tuples n'ont pas de méthode `index`.

❹ Vous pouvez toutefois utiliser `in` pour vérifier l'existence d'un élément dans un tuple.

Mais à quoi servent donc les tuples ?

- Les tuples sont plus rapides que les listes. Si vous définissez un ensemble constant de valeurs et que tout ce que vous allez faire est le parcourir, utilisez un tuple au lieu d'une liste.
- Votre code est plus sûr si vous "protégez en écriture" les données qui n'ont pas besoin d'être modifiées. Utiliser un tuple à la place d'une liste revient à avoir une assertion implicite que les données sont constantes et que des mesures spécifiques sont nécéssaires pour modifier cette définition.
- Vous vous souvenez que j'avais dit que que les clés de dictionnaire pouvaient être des entiers, des chaînes et "quelques autres types" ? Les tuples sont un de ces types. Ils peuvent être utilisé comme clé dans un dictionnaire, ce qui n'est pas le cas des listes. En fait, c'est plus compliqué que ça. Les clés de dictionnaire doivent être non−mutables. Les tuples sont non−mutables mais si vous avez un tuple contenant des listes, il est considéré comme mutable et n'est pas utilisable comme clé de dictionnaire. Seuls les tuples de chaînes, de nombres ou d'autres tuples utilisable comme clé peuvent être utilisés comme clé de dictionnaire.
- Les tuples sont utilisés pour le formatage de chaînes, comme nous le verrons bientôt.

Les tuples peuvent être convertis en listes et vice−versa. La fonction prédéfinie `tuple` prends une liste et retourne un tuple contenant les mêmes éléments et la fonction `list` prends un tuple et retourne une liste. En fait, `tuple` gèle une liste et `list` dégèle un tuple.

Pour en savoir plus sur les tuples

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) explique les tuples et montre comment concaténer des tuples (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) vous apprendra à trier un tuple (http://www.faqts.com/knowledge−base/view.phtml/aid/4553/fid/587).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) explique comment définir un tuple avec un seul élément (http://www.python.org/doc/current/tut/node7.html#SECTION007300000000000000000).

3.4. Définitions de variables

Maintenant que vous pensez tout savoir à propos des dictionnaires, des tuples et des listes (hum!), revenons à notre programme d'exemple du Chapitre 2, `odbchelper.py`.

Python dispose de variables locales et globales comme la plupart des autres langages, mais il n'a pas de déclaration explicite des variables. Les variables viennent au monde en se voyant assigner une valeur et sont automatiquement détruites lorsqu'elles se retrouvent hors de portée.

Exemple 3.17. Définition de la variable `myParams`

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
```

```
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Il y a plusieurs points intéressants ici. Tout d'abord, notez l'indentation. Une instruction `if` est un bloc de code et nécessite d'être indenté tout comme une fonction.

Deuxièmement, l'assignation de variable est une commande étalée sur plusieurs lignes avec une barre oblique ("\") servant de marque de continuation de ligne.

Lorsqu'une commande est étalée sur plusieurs lignes avec le marqueur de continuation de ligne ("\"), les lignes suivantes peuvent être indentées de n'importe qu'elle manière, les règles d'indentation strictes habituellement utilisées en Python ne s'appliquent pas. Si votre IDE Python indente automatiquement les lignes continuées, vous devriez accepter ses réglages par défauts sauf raison impérative.

Les expressions entre parenthèses, crochets ou accolades (comme la définition d'un dictionnaire) peuvent être réparties sur plusieurs lignes avec ou sans le caractère de continuation ("\"). Je préfère inclure la barre oblique même lorsqu'elle n'est pas requise car je pense que cela rends le code plus lisible mais c'est une question de style.

Troisièmement, vous n'avez jamais déclaré la variable `myParams`, vous lui avez simplement assigné une valeur. C'est comme en VBScript sans l'option `option explicit`. Heureusement, à l'inverse de VBScript, Python ne permet pas de référencer une variable à laquelle aucune valeur n'a été assigné. Tenter de le faire déclenchera une exception.

3.4.1. Référencer des variables

Exemple 3.18. Référencer une variable non assignée

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
>>> x
1
```

Un jour, vous remercierez Python pour ça.

3.4.2. Assignation simultanée de plusieurs valeurs

Un des raccourcis les plus réjouissants existant en Python est l'utilisation de séquences pour assigner plusieurs valeurs en une fois.

Exemple 3.19. Assignation simultanée de plusieurs valeurs

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v      ❶
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

❶ `v` est un tuple de trois éléments et `(x, y, z)` est un tuple de trois variables. Le fait d'assigner l'un à l'autre assigne chacune des valeurs de `v` a chacune des variables, dans leur ordre respectif.

Ce type d'assignation a de multiples usages. Je souhaite souvent assigner des noms a une série de valeurs. En C, vous utiliseriez `enum` et vous listeriez manuellement chaque constante et la valeur associée, ce qui semble particulièrement fastidieux lorsque les valeurs sont consécutives. En Python, vous pouvez utiliser la fonction prédéfinie `range` avec l'assignation multiples de variables pour assigner rapidement des valeurs consécutives.

Exemple 3.20. Assignation de valeurs consécutives

```
>>> range(7)                                                                    ❶
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) ❷
>>> MONDAY                                                                      ❸
0
>>> TUESDAY
1
>>> SUNDAY
6
```

❶ La fonction prédéfinie `range` retourne une liste d'entiers. Dans sa forme la plus simple, elle prends une borne supérieure et retourne une séquence démarrant à `0` mais n'incluant pas la borne supérieure. (Si vous le souhaitez, vous pouvez spécifier une borne inférieure différente de `0` ou un pas d'incrément différent de `1`. Vous pouvez faire un `print range.__doc__` pour de plus amples détails.)

❷ `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` et `SUNDAY` sont les variables que nous définissons (cet exemple provient du module `calendar`, qui est un petit module amusant qui affiche des calendriers comme le programme `cal` sous UNIX. Le module `calendar` défini des constantes entières pour les jours de la semaine).

❸ A présent, chaque variable possède sa valeur : `MONDAY` vaut `0`, `TUESDAY` vaut `1` et ainsi de suite.

Vous pouvez aussi utiliser l'assignation multiple pour créer des fonctions qui retournent plusieurs valeurs, simplement en retournant un tuple contenant ces valeurs. L'appelant peut le traiter en tant que tuple ou assigner les valeurs à différentes variables. Beaucoup de bibliothèques standard de Python font cela, y compris le module `os` dont nous traiterons au Chapitre 6.

Pour en savoir plus sur les variables

- Le *Python Reference Manual* (http://www.python.org/doc/current/ref/) présente des exemples des cas où vous pouvez omettre le marqueur de continuation (http://www.python.org/doc/current/ref/implicit−joining.html) et où vous devez l'utiliser (http://www.python.org/doc/current/ref/explicit−joining.html).
- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) montre comment utiliser l'assignation multiple pour échanger les valeurs de deux variables (http://www.ibiblio.org/obp/thinkCSpy/chap09.htm).

3.5. Formatage de chaînes

Python supporte le formatage de valeurs en chaînes de caractères. Bien que cela peut comprendre des expression très compliquées, l'usage le plus simple consiste à insérer des valeurs dans des chaînes à l'aide de marques `%s`.

Le formatage de chaînes en Python utilise la même syntaxe que la fonction C `sprintf`.

Exemple 3.21. Présentation du formatage de chaînes

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) ❶
'uid=sa'
```

❶ L'expression entière est évaluée en chaîne. Le premier `%s` est remplacé par la valeur de `k`, le second `%s` est remplacé par la valeur de `v`. Tous les autres caractères de la chaînes (le signe d'égalité dans le cas présent) restent tels quels.

Notez que `(k, v)` est un tuple. Je vous avais dit qu'ils servaient à quelque chose.

Vous pourriez pensez que cela représente beaucoup d'efforts pour le formatage de chaîne se bornait à la concaténation. Il n'y est pas question uniquement de formatage mais également de conversion de types.

Exemple 3.22. Formatage de chaîne et concaténation

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid      ❶
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) ❷
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )           ❸ ❹
Users connected: 6
>>> print "Users connected: " + userCount                 ❺
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

❶ `+` est l'opérateur de concaténation de chaînes.

❷ Dans ce cas trivial, le formatage de chaînes mène au même résultat que la concaténation.

❸ `(userCount, )` est un tuple contenant un seul élément. Oui, la syntaxe est un peu étrange mais il y a un excellente raison : c'est un tuple sans ambiguité aucune. En fait, vous pouvez toujours mettre une virgule après l'élément terminal lors de la définition d'une liste, d'un tuple ou d'un dictionnaire mais cette virgule est obligatoire lors de la définition d'un tuple avec un élément unique. Si ce n'était pas le cas, Python ne pourrait distinguer si `(userCount)` est un tuple avec un seul élément ou juste la valeur `userCount`.

❹ Le formatage de chaîne fonctionne également avec des entiers en spécifiant `%d` au lieu de `%s`.

❺ Si vous tentez de concaténer une chaîne avec un autre type, Python va déclencher une exception. Au contraire du formatage de chaîne, la concaténation ne fonctionne que si tout les objets sont déjà de type chaîne.

Comme la fonction `printf` en C, le formatage de chaînes en Python est un véritable couteau suisse. Il y a des options à profusion et des modificateurs de format spécifiques pour de nombreux types de valeurs.

Exemple 3.23. Formatage de nombres

```
>>> print "Today's stock price: %f" % 50.4625   ❶
50.462500
>>> print "Today's stock price: %.2f" % 50.4625 ❷
50.46
>>> print "Change since yesterday: %+.2f" % 1.5 ❸
+1.50
```

❶ L'option de formatage `%f` considère la valeur comme un nombre décimal et l'affiche avec six chiffres après la virgule.

❷ Le modificateurs ".2" de l'option `%f` tronque la valeur à deux chiffres après la virgule.

❸ On peut également combiner les modificateurs. Ajouter le modificateur `+` affiche le signe positif ou négatif avant la valeur. Notez que le modificateur ".2" est toujours en place et qu'il formate la valeur avec exactement deux chiffres après la virgule.

Pour en savoir plus sur le formatage de liste

- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume tous les caractères spéciaux utilisés pour le formatage de chaînes (http://www.python.org/doc/current/lib/typesseq−strings.html).
- *Effective AWK Programming* (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Top) explique tous les caractères de formatage (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Control+Letters) et des technique de formatage avancées comme le règlage de la largeur ou de la précision et le remplissage avec des zéros (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Format+Modifiers).

3.6. Mutation de listes

Une des caractéristiques les plus puissantes de Python est la *list comprehension* (création fonctionnelle de listes) qui fournit un moyen concis d'appliquer une fonction sur chaque élément d'une liste afin d'en produire une nouvelle.

Exemple 3.24. Présentation des *list comprehensions*

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]      ❶
[2, 18, 16, 8]
>>> li                           ❷
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] ❸
>>> li
[2, 18, 16, 8]
```

❶ Pour comprendre cette ligne, observez là de droite à gauche. `li` est la liste que vous appliquez. Python la parcourt un élément à la fois, en assignant temporairement la valeur de chacun des éléments à la variable `elem`. Python applique ensuite la fonction `elem*2` et ajoute le résultat à la liste retournée.

❷ Notez que les *list comprehensions* ne modifient pas la liste initiale.

❸ Vous pouvez assigner le résultat d'une *list comprehension* à la variable que vous traitez. Python assemble la nouvelle liste en mémoire et assigne le résultat à la variable une fois la transformation terminée.

Voici les *list comprehensions* dans la fonction `buildConnectionString` que nous avons déclaré au Chapitre 2:

```
["%s=%s" % (k, v) for k, v in params.items()]
```

Notez tout d'abord que vous appelez la fonction `items` du dictionnaire `params`. Cette fonction retourne une liste de tuples avec toutes les données stockées dans le dictionnaire.

Exemple 3.25. Les fonctions `keys`, `values` et `items`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()    ❶
['server', 'uid', 'database', 'pwd']
>>> params.values() ❷
```

```
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()  ❸
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

❶ La méthode `keys` d'un dictionnaire retourne la liste de toutes les clés. Cette liste ne suit pas l'ordre dans lequel le dictionnaire a été défini (souvenez−vous, les éléments d'un dictionnaire ne sont pas ordonnés) mais cela reste une liste.

❷ La méthode `values` retourne la liste de toutes les valeurs. La liste est dans le même ordre que celle retournée par`keys`, on a donc `params.values()[n] == params[params.keys[n]]` pour toute valeur de `n`.

❸ La méthode `items` retourne une liste de tuples de la forme `(clé, valeur)`. La liste contient toutes les données stockées dans le dictionnaire.

Voyons maintenant ce que fait `buildConnectionString`. Elle prends une liste, `param.items()`, et crée une nouvelle liste en appliquant une instruction de formatage de chaîne à chacun de ses éléments. La nouvelle liste aura le même nombre d'éléments que `params.items()` mais chaque élément sera une chaîne qui contient à la fois une clé et la valeur qui lui est associée dans le dictionnaire `params`.

Exemple 3.26. *List comprehensions* dans `buildConnectionString`, pas à pas

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
>>> [k for k, v in params.items()]                ❶
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]                ❷
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()] ❸
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

❶ Notez que nous utilisons deux variables pour parcourir la liste `params.items()`. Il s'agit d'un autre usage de l'assignment multiple. Le premier élément de `params.items()` est `('server', 'mpilgrim')`, donc lors de la première itération de la transformation, `k` va prendre la valeur `'server'` et `v` la valeur `'mpilgrim'`. Dans ce cas, nous ignorons la valeur de `v` et placons uniquement la valeur de `k` dans la liste résultante. Cette transformation correspond donc au comportement de `params.keys()`. (Vous n'utiliseriez pas réellement une *list comprehension* comme ceci dans du vrai code; il s'agit d'un exemple exagérément simple pour que vous compreniez ce qui se passe.)

❷ Nous faisons la même chose ici, mais nous ignorons la valeur de `k` de telle sorte que le résultat est équivalent à celui de `params.values()`.

❸ En combinant les deux exemples précédent avec le formatage de chaîne, nous obtenons une liste de chaînes comprenant la clé et la valeur de chaque élément du dictionnaire. Cela ressemble étonnament à la sortie du programme, tout ce qui reste à faire maintenant est la jointure des éléments de cette liste en une seule chaîne.

Pour en savoir plus sur les *list comprehensions*

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d'une autre manière de transformer des listes avec la fonction prédéfinie `map` (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) montre comment emboîter des mutations de listes (http://www.python.org/doc/current/tut/node7.html#SECTION007140000000000000000).

3.7. Jointure de listes et découpage de chaînes

Nous avons une liste de paires clé−valeur sous la forme `clé=valeur` et nous voulons les assembler au sein d'une même chaîne. Pour joindre une liste de chaînes en une seule, nous pouvons utiliser la méthode `join` d'un objet chaîne.

Voici un exemple de jointure de liste provenant de la fonction `buildConnectionString` :

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Une remarque intéressante avant de continuer. Je ne cesse de répéter que les fonctions sont des objets, que les chaînes sont des objets, que tout est objet. Vous pourriez penser que seules les *variables* de type chaîne sont des objets. Mais ce n'est pas le cas, regardez de plus près cet exemple et vous verrez que la chaîne `";"` est elle même un objet dont vous appelez la méthode `join`.

La méthode `join` assemble les éléments d'une liste pour former une chaîne unique, chaque élément étant séparé par un point virgule. Le séparateur n'est pas forcément un point−virgule, il n'est même pas forcément un caractère unique. Il peut être n'importe quelle chaîne.

La méthode `join` ne fonctionne qu'avec des listes de chaînes; elle n'applique pas la conversion de types. La `join`ture d'une liste comprenant au moins un élément non−chaîne déclenchera une exception.

Exemple 3.27. Sortie de `odbchelper.py`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

La chaîne est alors retournée de la fonction `odbchelper` et affichée par le bloc appelant, ce qui vous donne la sortie qui vous a tant émerveillé quand vous avez débuté la lecture de ce chapitre.

Vous vous demandez probablement s'il existe une méthode analogue permettant de découper une chaîne en liste. Et bien sur elle existe, elle porte le nom de `split`.

Exemple 3.28. Découpage d'une chaîne

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")    ❶
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s.split(";", 1) ❷
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

❶ `split` fait l'inverse de `join` en découpant une chaîne en une liste de plusieurs éléments. Notez que le délimiteur (`";"`) est totalement supprimé, il n'apparaît dans aucun des éléments de la liste retournée.

❷ `split` prend en deuxième argument optionnel le nombre de découpages à effectuer ("Des arguments optionnels ?" Vous apprendrez à en définir dans vos propres fonctions au prochain chapitre.)

une_chaîne`.split(`*delimiteur*`, 1)` est une technique utile pour chercher une sous−chaîne dans une chaîne et utiliser tout ce qui précède cette sous−chaîne (le premier élément de la liste retournée) et tout ce qui la suit (le second élément de la liste retournée).

Pour en savoir plus sur les méthodes de chaînes

- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions courantes à propose des chaînes (http://www.faqts.com/knowledge−base/index.phtml/fid/480) et dispose de nombreux exemples de code utilisant des chaînes (http://www.faqts.com/knowledge−base/index.phtml/fid/539).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) récapitule toutes les méthodes de chaînes (http://www.python.org/doc/current/lib/string−methods.html).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `string` (http://www.python.org/doc/current/lib/module−string.html).
- *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) explique pourquoi `join` est une méthode de chaînes (http://www.python.org/cgi−bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) et non une méthode de liste.

3.7.1. Note historique sur les méthodes de chaînes

Lorsque j'ai débuté l'apprentissage de Python, je m'attendais à ce que `join` soit une méthode de liste qui aurait pris un séparateur comme argument. Beaucoup de gens pensent la même chose et il y a une véritable histoire derrière la méthode `join`. Avant Python 1.6, les chaînes n'étaient pas pourvue de toutes ces méthodes si utiles. Il y avait un module `string` séparé qui contenait toutes les fonctions de manipulation de chaînes de caractères, chacune prenant une chaîne comme premier argument. Ces fonctions ont été considérées assez importantes pour être intégrées dans les chaînes elles même, ce qui semblait logique pour des fonctions comme `lower`, `upper` et `split`. Mais beaucoup de programmeurs Python issus du noyau dur émirent des objections quant à la méthode `join` en arguant du fait qu'elle devrait être plutôt une méthode de liste ou tout simplement rester une fonction du module `string` (qui contient encore bien des choses utiles). J'utilise exclusivement la nouvelle méthode `join` mais vous verrez du code écrit des deux façons et si cela vous pose un réel problème, vous pouvez toujours opter pour l'ancienne fonction `string.join`.

3.8. Résumé

A présent, le programme `odbchelper.py` et sa sortie devraient vous paraître parfaitement clairs.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Voici la sortie de `odbchelper.py`:

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Avant de vous plonger dans le chapitre suivant, assurez vous que vous vous sentez à l'aise pour :

- Utiliser l'IDE Python pour tester des expressions de manière interactive
- Ecrire des modules Python et les exécuter depuis votre IDE ou en ligne de commande
- Importer des modules et appeler leurs fonctions
- Déclarer des fonctions et utiliser des `doc string`, des variables locales et une indentation correcte
- Définir des dictionnaires, des tuples et des listes
- Accéder aux attributs et méthodes de tout objet, y compris les chaînes, les listes, les dictionnaires, les fonctions et les modules
- Concaténer des valeur avec le formatage de chaînes
- Utiliser les *lists comprehensions* pour la mutation de listes
- Découper des chaînes en listes et joindre des listes en chaînes

Chapitre 4. Le pouvoir de l introspection

Ce chapitre traite d une des forces de Python : l introspection. Comme vous le savez, tout est objet dans Python, l introspection consiste à considérer des modules et des fonctions en mémoire comme des objets, à obtenir des informations de leur part et à les manipuler. Au cours du chapitre, nous définirons des fonctions sans nom, nous appelerons des fonctions avec les arguments dans le désordre et nous référencerons des fonctions dont nous ne connaissons même pas le nom à l avance.

4.1. Plonger

Voici un programme Python complet et fonctionnel. Vous devriez en comprendre une grande partie rien qu en le lisant. Les lignes numérotées illustrent des concepts traités dans Chapitre 2, *Votre premier programme Python*. Ne vous inquiétez pas si le reste du code a l air intimidant, vous en apprendrez tous les aspects au cours de ce chapitre.

Exemple 4.1. `apihelper.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
def info(object, spacing=10, collapse=1): ❶ ❷ ❸
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":                  ❹ ❺
    print info.__doc__
```

❶ Ce module a une fonction, `info`. Selon sa déclaration de fonction, elle prend trois paramètres : `object`, `spacing` et `collapse`. Les deux derniers sont en fait des paramètres optionnels comme nous le verrons bientôt.

❷ La fonction `info` a une `doc string` multi−lignes qui décrit succintement son usage. Notez qu aucune valeur de retour n est mentionnée, cette fonction sera employée uniquement pour son effet, pas sa valeur.

❸ Le code à l interieur de la fonction est indenté.

❹ L astuce `if __name__` permet à ce programme de faire quelque chose d utile lorsqu il est exécuté tout seul sans intérférence avec son usage comme module pour d autres programmes. Dans ce cas, le programme affiche simplement la `doc string` de la fonction `info`.

❺ L instruction `if` utilise `==` pour la comparaison et ne nécessite pas de parenthèses.

La fonction `info` est conçue pour être utilisée par vous, le programmeur, lorsque vous travaillez dans l IDE Python. Elle prend n importe quel objet qui a des fonctions ou des méthodes (comme un module, qui a des fonction, ou une liste, qui a des méthodes) et affiche les fonctions et leur `doc string`.

Exemple 4.2. Exemple d'utilisation de `apihelper.py`

```
>>> from apihelper import info
```

```
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Par défaut, la sortie est formatée pour être facilement lisible. Les `doc string` multi-lignes sont combinées en une seule longue ligne, mais cette option peut être changée en spécifiant `0` pour l argument *`collapse`*. Si les noms de fonction font plus de 10 caractères, vous pouvez spécifier une valeur plus grande pour l argument *`spacing`*, pour faciliter la lecture.

Exemple 4.3. Utilisation avancée de `apihelper.py`

```
>>> import odbchelper
>>> info(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary Returns string.
>>> info(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary

    Returns string.
```

4.2. Arguments optionnels et nommés

Python permet aux arguments de fonction d avoir une valeur par défaut, si la fonction est appelée sans l argument il a la valeur par défaut. De plus, les arguments peuvent être donnés dans n importe quel ordre en utilisant les arguments nommés. Les procédures stockées de Transact/SQL sous SQL Server peuvent faire la même chose, si vous êtes un as des scripts sous SQL Server, vous pouvez survoler cette partie.

Voici un exemple de `info`, une fonction avec deux arguments optionnels :

```
def info(object, spacing=10, collapse=1):
```

`spacing` et `collapse` sont optionnels car ils ont des valeurs par défaut définies. `object` est obligatoire car il n a pas de valeur par défaut. Si `info` est appelé avec un seul argument, `spacing` prend pour valeur `10` et `collapse` la valeur `1`. Si `info` est appelé avec deux arguments, `collapse` prend encore pour valeur `1`.

Imaginez que vous vouliez spécifier une valeur pour `collapse` mais garder la valeur par défaut pour `spacing`. Dans la plupart des langages, vous ne pouvez pas le faire, vous auriez à spécifier les trois arguments. Mais en Python, les arguments peuvent être spécifiés par leur nom, dans n importe quel ordre.

Exemple 4.4. Appels de `info` autorisés

```
info(odbchelper)                  ❶
info(odbchelper, 12)              ❷
info(odbchelper, collapse=0)      ❸
info(spacing=15, object=odbchelper) ❹
```

❶ Avec un seul argument, `spacing` prend pour valeur `10` et `collapse` `1`.

❷ Avec deux arguments, `collapse` prend pour valeur `1`.

❸ Ici, vous nommez l argument `collapse` explicitement et spécifiez sa valeur. `spacing` prend la valeur par défaut `10`.

❹ Les arguments obligatoires (comme `object`, qui n a pas de valeurs par défaut) peuvent aussi être nommés et les arguments nommés peuvent apparaître dans n importe quel ordre.

Cela a l air confus jusqu à que vous réalisiez que les arguments sont tout simplement un dictionnaire. La manière "normale" d appeler les fonctions sans le nom des arguments est en fait un raccourci dans lequel Python fait correspondre les valeurs avec le nom des arguments dans l ordre dans lequel ils sont spécifiés par la déclaration de fonction. Dans la plupart des cas, vous appellerez le fonctions de la manière "normale", mais vous aurez toujours cette souplesse pour les autres cas.

La seule chose que vous avez à faire pour appeler une fonction est de spécifier une valeur (d une manière ou d une autre) pour chaque argument obligatoire, la manière et l ordre dans lequel vous le faites ne dépendent que de vous.

Pour en savoir plus

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de manière précise de quand et comment les arguments par défaut sont évalués (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000), ce qui est important lorsque la valeur par défaut est une liste ou une expression ayant un effet de bord.

4.3. Utilisation de `type`, `str`, `dir` et autres fonction prédéfinies

Python a un petit ensemble de fonctions prédéfinies très utiles. Toutes les autres fonctions sont réparties dans des modules. C est une décision de conception consciente, afin d éviter au langage de trop grossir comme d autres langages de script (au hasard, Visual Basic).

4.3.1. La fonction `type`

La fonction `type` retourne le type de données d un objet quelconque. Les types possibles sont répertoriés dans le module `types`. C est utile pour les fonctions capables de gérer plusieurs types de données.

Exemple 4.5. Présentation de `type`

```
>>> type(1)                ❶
<type 'int'>
>>> li = []
>>> type(li)               ❷
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)       ❸
<type 'module'>
>>> import types           ❹
>>> type(odbchelper) == types.ModuleType
True
```

❶ `type` prend n importe quel argument et retourne son type de données. Je dis bien n importe lequel : entiers, chaînes, listes, dictionnaires, tuples, fonctions, classes, modules et même types.

❷ `type` peut prendre une variable et retourne son type de données.

❸ `type` fonctionne aussi avec les modules.

❹ Vous pouvez utiliser les constantes du module `types` pour comparer les types des objets. C est ce que fait la fonction `info`, comme nous le verrons bientôt.

4.3.2. La fonction `str`

La fonction `str` convertit des données en chaîne. Tous les types de données peuvent être convertis en chaîne.

Exemple 4.6. Présentation de `str`

```
>>> str(1)          ❶
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen
['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)   ❷
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper) ❸
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)       ❹
'None'
```

❶ Pour des types simples comme les entiers, il semble normal que `str` fonctionne, presque tous les langages ont une fonction de conversion d entier en chaîne.

❷ Cependant, `str` fonctionne pour les objets de tout type. Ici, avec une liste que nous avons construit petit à petit.

❸ `str` fonctionne aussi pour les modules. Notez que la représentation en chaîne du module comprend le chemin du module sur le disque, la votre sera donc différente.

❹ Un aspect subtil mais important du comportement de `str` est qu elle fonctionne pour `None`, la valeur nulle de Python. Elle retourne la chaîne `'None'`. Nous utiliserons cela à notre avantage dans la fonction `info`, comme nous le verrons bientôt.

Au coeur de notre fonction `info`, il y a la puissante fonction `dir`. `dir` retourne une liste des attributs et méthodes de n importe quel objet : module, fonction, chaîne, liste, dictionnaire... à peu près tout.

Exemple 4.7. Introducing `dir`

```
>>> li = []
>>> dir(li)         ❶
['append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)          ❷
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper) ❸
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

❶ `li` est une liste, donc `dir(li)` retourne la liste de toutes les méthodes de liste. Notez que la liste retournée comprend les noms des méthodes sous forme de chaînes, pas les méthoses elles−mêmes.

❷ `d` est un dictionnaire, donc `dir(d)` retourne la liste des noms de méthodes de dictionnaire. Au moins l un de ces noms, `keys`, devrait être familier.

❸ C est ici que cela devient vraiment intéressant. `odbchelper` est un module, donc `dir(odbchelper)` retourne la liste de toutes les choses définies dans le module, y compris le attributs prédéfinis comme

__name__ et __doc__ et tout attribut et méthode que vous définissez. Dans ce cas, `odbchelper` a une seule méthode définie par l utilisateur, la fonction `buildConnectionString` que nous avons étudiée au Chapitre 2.

Enfin, la fonction `callable` prend n importe quel objet et retourne `True` si l objet peut être appelé, sinon `False`. Les objets appelables sont les fonctions, les méthodes de classes ainsi que les classes elles−mêmes (nous verrons les classes au prochain chapitre).

Exemple 4.8. Présentation de `callable`

```
>>> import string
>>> string.punctuation                ❶
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                       ❷
<function join at 00C55A7C>
>>> callable(string.punctuation)      ❸
False
>>> callable(string.join)             ❹
True
>>> print string.join.__doc__         ❺
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

❶ Les fonctions du module `string` sont dépréciées (bien que beaucoup de gens utilisent encore la fonction `join`), mais le module comprend un grand nombre de constantes utiles comme ce `string.punctuation`, qui comprend tous les caractères de ponctuation standards.

❷ `string.join` est une fonction qui effectue la jointure d une liste de chaînes.

❸ `string.punctuation` n est pas appelable, c est une chaîne. (Une chaîne a des méthodes appelables, mais elle n est pas elle−même appelable.)

❹ `string.join` est appelable, c est une fonction qui prend deux arguments.

❺ Tout objet appelable peut avoir une `doc string`. En utilisant la fonction `callable` sur chacun des attributs d un objet, nous pouvons déterminer les attributs qui nous intéressent (méthodes, fonctions et classes) et ce que nous voulons ignorer (constantes etc.) sans savoir quoi que ce soit des objets à l avance.

4.3.3. Fonctions prédéfinies

`type`, `str`, `dir` et toutes les autres fonctions prédéfinies de Python sont regroupés dans un module spécial appelé `__builtin__`. (Il y a deux caractères de soulignement avant et deux après.) Pour vous aider, vous pouvez imaginer que Python exécute automatiquement `from __builtin__ import *` au démarrage, ce qui importe toutes les fonctions prédéfinies (*built−in*) dans l espace de noms pour que vous puissiez les utiliser directement.

L avantage d y penser de cette manière est que vous pouvez accéder à toutes les fonctions et attributs prédéfinis de manière groupée en obtenant des informations sur le module `__builtin__`. Et devinez quoi, nous avons une fonction pour ça, elle s appelle `info`. Essayez vous−même et parcourez la liste maintenant, nous examinerons certaines des fonctions les plus importantes plus tard (certaines des classes d erreur prédéfinies, comme `AttributeError`, devraient avoir l air familier).

Exemple 4.9. Attributs et fonctions prédéfinis

```
>>> from apihelper import info
>>> import __builtin__
>>> info(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
Exception            Common base class for all exceptions.
FloatingPointError   Floating point operation failed.
IOError              I/O operation failed.

[...snip...]
```

Python est fourni avec d excellent manuels de référence que vous devriez parcourir de manière exhaustive pour apprendre tous les modules que Python offre. Mais alors que dans la plupart des langages vous auriez à vous référer constamment aux manuels (ou aux pages man, ou pire, à MSDN) pour vous rappeler l usage de ces modules, Python est en grande partie auto−documenté.

Pour en savoir plus sur les fonctions prédéfinies

- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente toutes les fonctions prédéfinies (http://www.python.org/doc/current/lib/built−in−funcs.html) et toutes les exceptions prédéfinies (http://www.python.org/doc/current/lib/module−exceptions.html).

4.4. Obtenir des références objet avec `getattr`

Vous savez déjà que les fonctions Python sont des objets. Ce que vous ne savez pas, c est que vous pouvez obtenir une référence à une fonction sans connaître son nom avant l exécution, à l aide de la fonction `getattr`.

Exemple 4.10. Présentation de `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                          ❶
<built−in method pop of list object at 010DF884>
>>> getattr(li, "pop")              ❷
<built−in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe")    ❸
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")            ❹
<built−in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")              ❺
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

❶ Ceci retourne une référence à la méthode `pop` de la liste. Ce n est pas un appel à la méthode `pop`, un appel se ferait par `li.pop()`. C est la méthode elle−même.

❷ Ceci retourne également une référence à la méthode `pop`, mais cette fois ci le nom de la méthode est passé comme argument de la fonction `getattr`. `getattr` est une fonction prédéfinie extrèmement utile qui retourne n importe quel attribut de n importe quel objet. Ici, l objet est une liste et l attribut est la méthode `pop`.

❸ Au cas où vous ne voyez pas à quel point c est utile, regardez ceci : la valeur de retour de `getattr` *est* la

méthode, que vous pouvez alors appeler comme si vous aviez tapé `li.append("Moe")` directement. Mais vous n avez pas appelé la fonction directement, vous avez passé le nom de la fonction comme paramètre sous forme de chaîne.

❹ `getattr` fonctionne aussi avec les dictionnaires.

❺ En théorie, `getattr` pourrait fonctionner avec les tuples, mais les tuples n ont pas de méthodes et `getattr` déclenchera une exception quel que soit le nom d attribut que vous lui donnez.

4.4.1. `getattr` et les modules

`getattr` n est pas seulement fait pour les types prédéfinis, il fonctionne aussi avec les modules.

Exemple 4.11. `getattr` dans `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString                ❶
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") ❷
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                         ❸
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                   ❹
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
True
>>> callable(getattr(object, method))               ❺
True
```

❶ Ceci retourne une référence à la fonction `buildConnectionString` du module `odbchelper`, que nous avons étudié au Chapitre 2, *Votre premier programme Python*. (L adresse hexadécimale qui s affiche est spécifique à ma machine, votre sortie sera différente.)

❷ A l aide de `getattr`, nous pouvons obtenir la même référence à la même fonction. En général, `getattr(`*`objet`*`, "`*`attribut`*`")` est équivalent à *`objet.attribut`*. Si *`objet`* est un module, alors *`attribut`* peut être toute chose définie dans le module : une fonction, une classe ou une variable globale.

❸ Voici ce que nous utilisons dans la fonction `info`. `object` est passé en argument à la fonction, `method` est une chaîne, le nom de la méthode ou de la fonction.

❹ Dans ce cas, `method` est le nom d une fonction, ce que nous prouvons en obtenant son `type`.

❺ Puisque `method` est une fonction, elle est callable (appelable).

4.4.2. `getattr` comme sélecteur

Une utilisation usuelle de `getattr` est dans le rôle de sélecteur. Par exemple, si vous avez un programme qui peut produire des données dans différents formats, vous pouvez définir des fonctions différentes pour chaque format de sortie et utiliser une fonction de sélection pour appeler celle qui convient.

Par exemple, imaginons un programme qui affiche des statistiques de consultations d'un site Web aux formats HTML, XML et texte simple. Le choix du format de sortie peut être spécifié depuis la ligne de commande ou stocké dans un fichier de configuration. Un module `statsout` définit trois fonctions, `output_html`, `output_xml` et `output_text`. Ensuite, le programme principal définit une fonction de sortie unique, comme ceci :

Exemple 4.12. Création d'un sélecteur avec `getattr`

```
import statsout

def output(data, format="text"):                                    ❶
    output_function = getattr(statsout, "output_%s" % format)       ❷
    return output_function(data)                                    ❸
```

❶ La fonction `output` prend un argument obligatoire, `data` et un argument optionnel, `format`. Si `format` n'est pas spécifié, il a la valeur par défaut `text` ce qui appelera la fonction de sortie en texte simple.

❷ Nous concaténons l'argument `format` à "output_" pour produire un nom de fonction et allons chercher cette fonction dans le module `statsout`. Cela nous permet d'étendre simplement le programme plus tard pour supporter d'autres formats de sortie, sans changer la fonction de sélection. Il suffit d'ajouter une autre fonction à `statsout` nommée, par exemple, `output_pdf` et de passer "pdf" comme `format` à la fonction `output`.

❸ Maintenant, nous pouvons simplement appeler la fonction de sortie comme toute autre fonction. La variable `output_function` est une référence à la fonction appropriée du module `statsout`.

Avez vous vu le problème dans l'exemple précédent ? Il y a un couplage très lâche entre chaînes et fonctions et il n'y a aucune vérification d'erreur. Que se passe−t−il si l'utilisateur passe un format pour lequel aucune fonction correspondante n'est définie dans le module `statsout`? Et bien, `getattr` retournera `None`, qui sera assigné à `output_function` au lieu d'une fonction valide et à la ligne suivante, qui tente d'appeler cette fonction inexistante, plantera et déclenchera une exception. C'est un problème.

Heureusement `getattr` prend un troisième argument optionnel, une valeurpar défaut.

Exemple 4.13. Valeurs par défaut de `getattr`

```
import statsout

def output(data, format="text"):
    output_function = getattr(statsout, "output_%s" % format, statsout.output_text)
    return output_function(data) ❶
```

❶ Cet appel de fonction est assuré de fonctionner puisque nous avons ajouté un troisième argument à l'appel à `getattr`. Le troisième argument est une valeur par défaut qui est retournée si l'attribut ou la méthode spécifié par le second argument n'est pas trouvé.

Comme vous pouvez le voir, `getattr` est très puissant. C'est le coeur même de l'introspection et vous en verrez des exemples encore plus puissants dans des prochains chapitres.

4.5. Filtrage de listes

Comme vous le savez, Python a des moyens puissant de mutation d une liste en une autre, au moyen des *list comprehensions* (Section 3.6, «Mutation de listes»). Cela peut être associé à un mécanisme de filtrage par lequel certains éléments sont modifiés alors que d autres sont totalement ignorés.

Voici la syntaxe du filtrage de liste :

```
[mapping−expression for element in source−list if filter−expression]
```

C est une extension des *list comprehensions* que vous connaissez et appréciez. Les deux premiers tiers sont identiques, la dernière partie, commençant par le `if`, est l expression de filtrage. Une expression de filtrage peut être n importe quelle expression qui s évalue en vrai ou faux (ce qui en Python peut être presque tout). Tout élément pour lequel l expression de filtrage s évalue à vrai sera inclu dans la liste à transformer. Tous les autres éléments seront ignorés, il ne passeront jamais par l expression de mutation et ne seront pas inclus dans la liste retournée.

Exemple 4.14. Présentation du filtrage de liste

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]      ❶
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]        ❷
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
>>> [elem for elem in li if li.count(elem) == 1] ❸
['a', 'mpilgrim', 'foo', 'c']
```

❶ L expression de mutation est ici très simple (elle retourne juste la valeur de chaque élément), observez plutôt attentivement l expression de filtrage. Au fur et à mesure que Python parcours la liste, il soumet chaque élément à l expression de filtrage, si l expression s évalue à vrai, l élément passe par l expression de mutation et le résultat est inclu dans la liste de résultat. Ici, on filtre toutes les chaînes d un seul caractère, il ne reste donc que les chaînes plus longues.

❷ Ici, on filtre une valeur spécifique : `b`. Notez que cela filtre toutes les occurences de `b`, puisqu à chaque fois qu il apparaît, l expression de filtrage s évaluera à faux.

❸ `count` est une méthode de listes qui retourne le nombre d occurences d une valeur dans la liste. On pourrait penser que ce filtre élimine les doublons de la liste, retournant une liste contenant seulement un exemplaire de chaque valeur. Mais en fait, les valeurs qui apparaissent deux fois dans la liste initiale (ici `b` et `d`) sont totalement éliminées. Il y a des moyens de supprimer les doublons d une liste mais le filtrage n est pas la solution.

Revenons à cette ligne de `apihelper.py`:

```
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
```

Cela à l air complexe et ça l est, mais la structure de base est la même. L expression complète renvoie une liste qui est assignée à la variable `methodList`. La première moitié de l expression est la mutation de liste. L expression de mutation est une expression d identité, elle retourne la valeur de chaque élément. `dir(object)` retourne une liste des attributs et méthodes de `object`, c est à cette liste que vous appliquez la mutation. La seule nouveauté est l expression après le `if`.

L expression de filtrage à l air impressionant, mais elle n est pas si terrible. Vous connaissez déjà `callable`, `getattr` et `in`. Comme vous l avez vu dans la section précédente, l expression `getattr(object, method)` retourne un objet fonction si `object` est un module et si `method` est le nom d une fonction de ce module.

Donc, cette expression prend un objet, appelé `object`, obtient une liste des noms de ses attributs, méthodes, fonctions et quelques autres choses, puis filtre cette liste pour éliminer ce qui ne nous intéresse pas. Cette élimination se fait en prenant le nom de chaque attribut/méthode/fonction et en obtenant une référence vers l objet véritable, grâce à la fonction `getattr`. On vérifie alors si cet objet est appelable, ce qui sera le cas pour toutes les méthodes et fonctions prédéfinies (comme la méthode `pop` d une liste) ou définies par l utilisateur (comme la fonction `buildConnectionString` du module `odbchelper`). Nous ne nous intéressons pas aux autres attributs, comme l attribut `__name__` qui existe pour tout module.

Pour en savoir plus sur le filtrage de liste

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d une autre manière de filtrer les listes en utilisant la fonction prédéfinie `filter` (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).

4.6. Particularités de and et or

En Python, and et or appliquent la logique booléenne comme vous pourriez l attendre, mais ils ne retournent pas de valeurs booléennes, ils retournent une des valeurs comparées.

Exemple 4.15. Présentation de and

```
>>> 'a' and 'b'          ❶
'b'
>>> '' and 'b'           ❷
''
>>> 'a' and 'b' and 'c'  ❸
'c'
```

❶ Lorsqu on utilise and les valeurs sont évaluées dans un contexte booléen de gauche à droite. 0, '', [], (), {} et None valent faux dans ce contexte, tout le reste vaut vrai.[1] Si toutes les valeurs valent vrai dans un contexte booléen, and retourne la dernière valeur. Ici and évalue 'a', qui vaut vrai, puis 'b', qui vaut vrai et retourne 'b'.

❷ Si une des valeurs vaut faux and retourne la première valeur fausse. Ici '' est la première valeur fausse.

❸ Toutes les valeurs sont vrai, donc and retourne la dernière valeur, 'c'.

Exemple 4.16. Présentation de or

```
>>> 'a' or 'b'           ❶
'a'
>>> '' or 'b'            ❷
'b'
>>> '' or [] or {}       ❸
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()      ❹
'a'
```

❶ Lorsqu on utilise or, les valeurs sont évaluées dans un contexte booléen de gauche à droite, comme pour and. Si une des valeurs vaut vrai, or la retourne immédiatement. Dans ce cas 'a' est la première valeur vraie.

❷ or évalue '', qui vaut faux, puis 'b', qui vaut vrai et retourne 'b'.

❸ Si toutes les valeurs valent faux, or retourne la dernière valeur. or évalue '', qui vaut faux, puis [], qui vaut faux, puis {}, qui vaut faux et retourne {}.

❹ Notez que or continue l évaluation seulement jusqu à ce qu il trouve une valeur vraie, le reste est ignoré. C est important si certaines valeurs peuvent avoir un effet de bord. Ici, la fonction sidefx n est jamais appelée, car or évalue 'a', qui vaut vrai et retourne 'a' immédiatement.

Si vous êtes un programmeur C, vous êtes certainement familier de l expression ternaire *bool* `? a : b`, qui s évalue à a si *bool* vaut vrai et à b dans le cas contraire. Le fonctionnement de and et or en Python vous permet d accomplir la même chose.

4.6.1. Utilisation de l'astuce and-or

Exemple 4.17. Présentation de l astuce and-or

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b ❶
'first'
>>> 0 and a or b ❷
'second'
```

❶ Cette syntaxe ressemble à celle de l expression ternaire *`bool ? a : b`* de C. L expression est évaluée de gauche à droite, donc le `and` est évalué en premier. `1 and 'first'` s évalue à `'first'`, puis `'first' or 'second'` s évalue à `'first'`.

❷ `0 and 'first'` s évalue à `0`, puis `0 or 'second'` s évalue à `'second'`.

Cependant, puisque cette expression en Python est simplement de la logique booléenne et non un dispositif spécial du langage, il y a une différence très, très importante entre l astuce `and-or` en Python et la syntaxe *`bool ? a : b`* en C. Si `a` vaut faux, l expression ne fonctionnera pas comme vous vous y attendez. (Vous devinez que cela m a déjà joué des tours. Et plus d une fois !)

Exemple 4.18. Quand l astuce `and-or` échoue

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b        ❶
'second'
```

❶ Puisque `a` est une chaîne vide, ce que Python évalue à faux dans un contexte booléen, `1 and ''` s évalue à `''`, puis `'' or 'second'` s évalue à `'second'`. Zut ! Ce n est pas ce que nous voulions.

L astuce `and-or`, *`bool and a or b`*, ne fonctionne pas comme l expression ternaire de C *`bool ? a : b`* quand `a` s évalue à faux dans un contexte booléen.

La véritable astuce cachée derrière l astuce `and-or`, c est de s assurer que `a` ne vaut jamais faux. Une manière habituelle de le faire est de changer `a` en `[a]` et `b` en `[b]` et de prendre le premier élément de la liste retournée, qui sera soit `a` soit `b`.

Exemple 4.19. L astuce `and-or` en toute sécurité

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] ❶
''
```

❶ Puisque `[a]` est une liste non-vide, il ne vaut jamais faux. Même si `a` est `0` ou `''` ou une autre valeur fausse, la liste `[a]` vaut vrai puisqu elle a un élément.

On peut penser que cette astuce apporte plus de complication que d avantages. Après tout, on peut obtenir le même résultat avec une instruction `if`, alors pourquoi s embarasser de tout ces problèmes ? Mais dans de nombreux cas, le choix se fait entre deux valeurs constantes et donc vous pouvez utiliser cette syntaxe plus simple sans vous inquiéter puisque vous savez que `a` vaudra toujours vrai. Et même si vous devez utiliser la version sûre plus complexe, il y a parfois de bonnes raisons de le faire, il y a des cas en Python où les instructions `if` ne sont pas autorisées, comme dans les fonctions `lambda`.

Pour en savoir plus sur l'astuce `and-or`

- Le Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) traite des alternatives à l astuce `and-or` (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52310).

4.7. Utiliser des fonctions `lambda`

Python permet une syntaxe intéressante qui vous laisse définir des mini−fonctions d une ligne à la volée. Empruntées à Lisp, ces fonctions dites `lambda` peuvent être employées partout où une fonction est nécéssaire.

Exemple 4.20. Présentation des fonctions `lambda`

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2  ❶
>>> g(3)
6
>>> (lambda x: x*2)(3) ❷
6
```

❶ Voici une fonction `lambda` qui fait la même chose que la fonction ordinaire précédente. Notez la syntaxe condensée : il n y a pas de parenthèses autour de la liste d arguments et le mot−clé `return` est manquant (il est implicite, la fonction complète ne pouvant être qu une seule expression). Remarquez aussi que la fonction n a pas de nom, mais qu elle peut être appelée à travers la variable à laquelle elle est assignée.

❷ Vous pouvez utiliser une fonction `lambda` sans l assigner à une variable. Ce n est pas forcément très utile, mais cela démontre qu une fonction `lambda` est simplement une fonction en ligne.

Plus généralement, une fonction `lambda` est une fonction qui prend un nombre quelconque d arguments (y compris des arguments optionnels) et retourne la valeur d une expression unique. Les fonctions `lambda` ne peuvent pas contenir de commandes et elles ne peuvent contenir plus d une expression. N essayez pas de mettre trop de choses dans une fonction `lambda`, si vous avez besoin de quelque chose de complexe, définissez plutôt une fonction normale et faites la aussi longue que vous voulez.

Les fonctions `lambda` sont une question de style. Les utiliser n est jamais une nécessité, partout où vous pouvez les utiliser, vous pouvez utiliser une fonction ordinaire. Je les utilise là où je veux incorporer du code spécifique et non réutilisable sans encombrer mon code de multiples fonctions d une seule ligne.

4.7.1. Les fonctions `lambda` dans le monde réel

Voici les fonctions `lambda` dans `apihelper.py`:

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

Il y a plusieurs chose à noter ici. D abord, nous utilisons la forme simple de l astuce `and−or`, ce qui est sûr car une fonction `lambda` vaut toujours vrai dans un contexte booléen. (Cela ne veut pas dire qu une fonction `lambda` ne peut retourner faux. La fonction est toujours vrai, sa valeur de retour peut être vrai ou fausse.)

Ensuite, nous utilisons la fonction `split` sans arguments. Vous l avez déjà vu employée avec un ou deux arguments, sans arguments elle utilise les espaces comme séparateur.

Exemple 4.21. `split` sans arguments

```
>>> s = "this   is\na\ttest"  ❶
```

```
>>> print s
this   is
a      test
>>> print s.split()           ❷
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) ❸
'this is a test'
```

❶ Voici une chaîne multi−lignes définie à l aide de caractères d échappement au lieu de triples guillemets. `\n` est le retour chariot et `\t` le caractère de tabulation.

❷ `split` sans arguments fait la séparation sur les espaces. Trois espaces, un retour chariot ou un caractère de tabulation reviennent à la même chose.

❸ Vous pouvez normaliser les espaces en utilisant `split` sur une chaîne et en la joignant par `join` avec un espace simple comme délimiteur. C est ce que fait la fonction `info` pour replier les `doc string` sur une seule ligne.

Mais que fait donc exactement cette fonction `info` avec ces fonctions `lambda`, ces `split` et ces astuces `and−or` ?

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

`processFunc` est maintenant une fonction, mais la fonction qu elle est dépend de la valeur de la variable `collapse`. Si `collapse` vaut vrai, `processFunc(`*`string`*`)` repliera les espaces, sinon, `processFunc(`*`string`*`)` retournera son argument sans le modifier.

Pour faire la même chose dans un langage moins robuste, tel que Visual Basic, vous auriez sans doute créé une fonction prenant une chaîne et un argument *`collapse`* qui aurait utilisé une instruction `if` pour décider de replier les espaces ou non, puis aurait retourné la valeur appropriée. Cela serait inefficace car la fonction devrait prendre en compte tous les cas possibles. A chaque fois que vous l appeleriez, elle devrait décider si elle doit replier l espace avant de pouvoir vous donner ce que vous souhaitez. En Python, vous pouvez retirer cette prise de décision de la fonction et définir une fonction `lambda` taillée sur mesure pour vous donner ce que vous voulez et seulement cela. C est plus efficace, plus élégant et moins sujet à des erreurs dans l ordre des arguments.

Pour en savoir plus sur les fonctions `lambda`

- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) traite de l utilisation de `lambda` pour faire des appels de fonction indirects (http://www.faqts.com/knowledge−base/view.phtml/aid/6081/fid/241).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) montre comment accéder à des variables extérieures de l intérieur d une fonction `lambda` (http://www.python.org/doc/current/tut/node6.html#SECTION006740000000000000000). (La PEP 227 (http://python.sourceforge.net/peps/pep−0227.html) explique comment cela va changer dans les futures versions de Python.)
- La *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) a des exemples de code obscur monoligne utilisant `lambda` (http://www.python.org/cgi−bin/faqw.py?query=4.15&querytype=simple&casefold=yes&req=search).

4.8. Assembler les pièces

La dernière ligne du code, la seule que nous n ayons pas encore déconstruite, est celle qui fait tout le travail. Mais arrivé à ce point, le travail est simple puisque tous les éléments dont nous avons besoin sont disponibles. Les dominos sont en place, il ne reste qu à les faire tomber.

Voici le plat de résistance de `apihelper.py`:

```
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])
```

Notez que ce n est qu une commande, répartie sur plusieurs lignes sans utiliser le caractère de continuation ("\"). Vous vous rappelez quand j ai dit que certaines expressions peuvent être divisées en plusieurs lignes sans utiliser de *backslash* ? Une *list comprehension* est une expression de ce type car toute l expression est entourée de crochets.

Maintenant étudions l expression de la fin vers le début. L'instruction

```
      for method in methodList
```

nous montre qu il s agit d une *list comprehension*. Comme vous le savez, `methodList` est une liste de toutes les méthodes qui nous intéressent dans `object`. Nous parcourons donc cette liste avec `method`.

Exemple 4.22. Obtenir une `doc string` dynamiquement

```
>>> import odbchelper
>>> object = odbchelper                    ❶
>>> method = 'buildConnectionString'       ❷
>>> getattr(object, method)                ❸
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__  ❹
Build a connection string from a dictionary of parameters.

    Returns string.
```

❶ Dans la fonction `info`, `object` est l objet pour lequel nous demandons de l aide, passé en argument.

❷ Pendant que nous parcourons la `methodList`, `method` est le nom de la méthode en cours.

❸ En utlisant la fonction `getattr`, nous obtenons une référence à la fonction *`method`* du module *`object`*.

❹ Maintenant, afficher la `doc string` de la méthode est facile.

La pièce suivante du puzzle est l utilisation de `str` sur la `doc string`. Comme vous vous rappelez peut−être, `str` est une fonction prédéfinie pour convertir des données en chaîne. Mais une `doc string` est toujours une chaîne, alors pourquoi utiliser `str` ? La réponse est que toutes les fonctions n ont pas de `doc string`, et que l attribut `__doc__` de celles qui n en ont pas renvoi `None`.

Exemple 4.23. Pourquoi utiliser `str` sur une `doc string` ?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__       ❶
>>> foo.__doc__ == None   ❷
True
>>> str(foo.__doc__)      ❸
'None'
```

❶ Nous pouvons facilement définir une fonction qui n a pas de `doc string`, sont attribut `__doc__` est `None`. Attention, si vous évaluez directement l attribut `__doc__`, l'IDE Python n affiche rien du tout, ce qui est logique si vous y réflechissez mais n est pas très utile.

❷ Vous pouvez vérifier que la valeur de l attribut `__doc__` est bien `None` en faisant directement la comparaison.

❸ Lorsque l on utilise la fonction `str`, elle prend la valeur nulle et en retourne une représentation en chaîne, `'None'`.

En SQL, vous devez utiliser `IS NULLmethod` au lieu de `= NULL` pour la comparaison d une valeur nulle. En Python, vous pouvez utiliser aussi bien `== None` que `is None`, mais `is None` est plus rapide.

Maintenant que nous sommes sûrs d obtenir une chaîne, nous pouvons passer la chaîne à `processFunc`, que nous avons déjà défini comme une fonction qui replie ou non les espace. Maintenant vous voyez qu il était important d utiliser `str` pour convertir une valeur `None` en une représentation en chaîne. `processFunc` attend une chaîne comme argument et appelle sa méthode `split`, ce qui échouerait si nous passions `None`, car `None` n a pas de méthode `split`.

En remontant en arrière encore plus loin, nous voyons que nous utilisons encore le formatage de chaîne pour concaténer la valeur de retour de `processFunc` avec celle de la méthode `ljust` de `method`. C est une nouvelle méthode de chaîne que nous n avons pas encore rencontré.

Exemple 4.24. Présentation de la méthode `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) ❶
'buildConnectionString         '
>>> s.ljust(20) ❷
'buildConnectionString'
```

❶ `ljust` complète la chaîne avec des espaces jusqu à la longueur donnée. La fonction `info` l utilise pour afficher sur deux colonnes et aligner les `doc string` de la seconde colonne.

❷ Si la longueur donnée est plus petite que la longueur de la chaîne, `ljust` retourne simplement la chaîne sans la changer. Elle ne tronque jamais la chaîne.

Nous avons presque terminé. Ayant obtenu le nom de méthode complété d espaces de la méthode `ljust` et la `doc string` (éventuellement repliée sur une ligne) de l appel à `processFunc`, nous concaténons les deux pour obtenir une seule chaîne. Comme nous faisons une mutation de `methodList`, nous obtenons une liste de chaînes. En utlisant la méthode `join` de la chaîne `"\n"`, nous joignons cette liste en une chaîne unique, avec chaque élément sur une ligne et affichons le résultat..

Exemple 4.25. Affichage d une liste

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li) ❶
a
b
c
```

❶ C est aussi une astuce de débogage utile lorsque vous travaillez avec des liste. Et en Python, vous travaillez toujours avec des listes.

C est la dernière pièce du puzzle. Le code devrait maintenant être parfaitement compréhensible.

```
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                    for method in methodList])
```

4.9. Résumé

Le programme `apihelper.py` et sa sortie devraient maintenant être parfaitement clairs.

```
def info(object, spacing=10, collapse=1):
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":
    print info.__doc__
```

Voici la sortie de `apihelper.py` :

```
>>> from apihelper import info
>>> li = []
>>> info(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Avant de plonger dans le chapitre suivant, assurez vous que vous vous sentez à l aise pour :

- Définir et appeler des fonctions avec des arguments optionnels et nommés
- Utiliser `str` pour convertir une valeur quelconque en chaîne
- Utiliser `getattr` pour obtenir des références à des fonctions et autres attributs dynamiquement
- Etendre la syntaxe des *list comprehensions* pour faire du filtrage de liste
- Identifier l astuce `and-or` et l utiliser de manière sûre
- Définir des fonctions `lambda`
- Assigner des fonctions à des variables et appeler ces fonctions en référençant les variables. Je n insisterais jamais assez : cette manière de penser est essentielle pour faire progresser votre compréhension de Python. Vous verrez des applications plus complexe de ce concept tout au long de ce livre.

---

[1] Presque tout, en fait. Par défaut, les instances de classes valent vrai dans un contexte booléen mais vous pouvez définir des méthodes spéciales de votre classe pour faire qu une instance vale faux. Vous apprendrez tout sur les classes et les méthodes spéciales au Chapitre 5.

Chapitre 5. Les objets et l'orienté objet

Ce chapitre, comme la plupart de ceux qui le suivent, a trait à la programmation orientée objet en Python.

5.1. Plonger

Voici un programme Python complet et fonctionnel. Lisez les `doc string` du module, des classes et des fonctions pour avoir un aperçu de ce que ce programme fait et de son fonctionnement. Comme d'habitude ne vous inquiétez pas de ce que vous ne comprenez pas, la suite du chapitre est là pour vous l'expliquer.

Exemple 5.1. `fileinfo.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Framework for getting filetype−specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key−value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
```

```
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]): ❶
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

❶ La sortie de ce programme dépend des fichiers qui se trouvent sur votre disque dur. Pour avoir une sortie pertinente, vous devrez changer le chemin pour qu'il pointe vers un répertoire de fichiers MP3 sur votre machine.

Voici la sortie que j'ai obtenu sur ma machine. Votre sortie sera différente, sauf si, par une surprenante coïncidence, vous partagez exactement mes goûts musicaux..

```
album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=31
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine

album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER****Trance from Hell
genre=31
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO****THE BEST GOA
genre=31
```

```
name=/music/_singles/kairo.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan

album=
artist=The Cynic Project
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject

album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

5.2. Importation de modules avec `from` *`module`* `import`

Python fournit deux manières d'importer les modules. Les deux sont utiles et vous devez savoir quand les utiliser. Vous avez déjà vu la première, `import` *`module`*, à la Section 2.4, «Tout est objet». La deuxième manière accomplit la même action avec des différences subtiles mais importante dans son fonctionnement.

Voici la syntaxe de base de `from` *`module`* `import` :

```
from UserDict import UserDict
```

Cela ressemble à la syntaxe `import` *`module`* que vous connaissez, mais avec une différence importante : les attributs et et les méthodes du module importé `types` sont importés directement dans l'espace de noms local, ils sont donc disponible directement sans devoir les qualifier avec le nom du module. Vous pouvez importer des éléments précis ou utiliser `from` *`module`* `import *` pour tout importer.

`from` *`module`* `import *` en Python est comme `use` *`module`* in Perl. `import` *`module`* en Python est comme `require` *`module`* en Perl.

`from` *`module`* `import *` en Python est comme `import` *`module`*`.*` en Java. `import` *`module`* en Python est comme `import` *`module`* en Java.

Exemple 5.2. `import` *`module`* *vs.* `from` *`module`* `import`

```
>>> import types
>>> types.FunctionType             ❶
<type 'function'>
>>> FunctionType                   ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
```

```
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType ❸
>>> FunctionType                   ❹
<type 'function'>
```

❶ Le module `types` ne contient aucune méthode, seulement des attributs pour chaque type d'objet Python. Notez que l'attribut `FunctionType` doit être qualifié avec le nom de module, `types`.

❷ `FunctionType` lui−même n'a pas été défini dans cet espace de noms, il n'existe que dans le contexte de `types`.

❸ Cette syntaxe permet l'importation de l'attribut `FunctionType` du module `types` directement dans l'espace de noms local.

❹ Maintenant `FunctionType` peut être référé directement, sans mentionner `types`.

Quand faut−il utiliser `from *module* import` ?

- Quand vous devez accéder fréquemment aux attributs et méthodes et que vous ne voulez pas taper le nom de module sans arrêt, utilisez `from *module* import`.
- Quand vous voulez n'importer que certains attributs et méthodes, utilisez `from *module* import`.
- Quand le module contient des attributs ou des fonctions ayant des noms déjà utilisés dans votre module, vous devez utiliser `import *module*` pour éviter les conflits de nom.

En dehors de ces cas, c'est une question de style et vous verrez du code Python écrit des deux manières.

Utilisez `from module import *` avec modération, il rend plus difficile de déterminer l'origine d'une fonction ou d'un attribut, ce qui rend le débogage et la refactorisation plus difficiles.

Pour en savoir plus sur l'importation de module

- eff−bot (http://www.effbot.org/guides/) a d'autres choses à ajouter à propos de `import *module*` et `from *module* import` (http://www.effbot.org/guides/import−confusion.htm).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de techniques d'import avancées, y compris `from *module* import *` (http://www.python.org/doc/current/tut/node8.html#SECTION008410000000000000000).

5.3. Définition de classes

Python est entièrement orienté objet : vous pouvez définir vos propres classes, hériter de vos classes ou des classes prédéfinies et instancier les classes que vous avez défini.

Définir une classe en Python est simple, comme pour les fonctions, il n'y a pas de définition séparée d'interface. Vous définissez simplement la classe et commencez à coder. Une classe Python commence par le mot réservé `class` suivi du nom de la classe. Techiquement c'est tout ce qui est requis, une classe n'hérite pas obligatoirement d'une autre.

Exemple 5.3. La classe Python la plus simple

```
class Loaf: ❶
    pass    ❷ ❸
```

❶ Le nom de cette classe est `Loaf` et elle n'hérite d'aucune autre classe. Chaque mot d'un nom de classe prend habituellement une majuscule, `DeCetteManiere`, mais c'est une simple convention et pas une nécéssité.

❷

Cette classe ne définit aucune méthode ni attribut, mais pour respecter la syntaxe, il est nécéssaire d'avoir quelque chose dans la définition, nous utilisons donc `pass`. C'est un mot réservé de Python qui signifie simplement "circulez, il n'y a rien à voir". C'est une instruction qui ne fait rien et c'est un bon marqueur lorsque vous écrivez un squelette de fonction ou de classe.

❸ Vous l'aurez sans doute deviné, tout est indenté dans une classe, comme le code d'une fonction, d'une instruction `if`, d'une boucle `for` etc. La première ligne non indenté ne fait plus partie de la classe.


L'instruction `pass` de Python est comme une paire d'accolades vide (`{}`) en Java ou C.

Bien sûr, dans des cas réels la plupart des classes hériteront d'autres classes et elles définiront leurs propres classes, méthodes et attributs. Mais comme vous l'avez vu, il n'y a rien qu'une classe doit absolument avoir en dehors d'un nom. En particulier, les programmeurs C++ s'étonneront sans doute que les classes Python n'aient pas de constructeurs et de destructeurs explicites. Les classes Python ont quelque chose de semblable à un constructeur : la méthode `__init__`.

Exemple 5.4. Définition de la classe `FileInfo`

```
from UserDict import UserDict

class FileInfo(UserDict): ❶
```

❶ En Python, l'ancêtre d'une classe est simplement indiqué entre parenthèses immédiatement après le nom de la classe. La classe `FileInfo` est hérite donc de la classe `UserDict` (qui a été importée du module `UserDict`). `UserDict` est une classe qui se comporte comme un dictionnaire, vous permettant pratiquement de dériver le type de données dictionnaire et d'y ajouter votre propre comportement (il y a des classes semblables `UserList` et `UserString` qui vous permettent de dériver les listes et les chaînes). Il y a un peu de magie noire derrière tout cela, nous la démystifierons plus loin dans ce chapitre lorsque nous explorerons la classe `UserDict` plus en détail.


En Python, l'ancêtre d'une classe est simplement indiqué entre parenthèses immédiatement après le nom de la classe. Il n'y a pas de mot clé spécifique comme `extends` en Java.

Python supporte l'héritage multiple. Entre les parenthèses qui suivent le nom de classe, vous pouvez indiquer autant de classes ancêtres que vous le souhaitez, séparées par des virgules.

5.3.1. Initialisation et écriture de classes

Cet exemple montre l'initialisation de la classe `FileInfo` avec la méthode `__init__`.

Exemple 5.5. Initialisation de la classe `FileInfo`

```
class FileInfo(UserDict):
    "store file metadata"                    ❶
    def __init__(self, filename=None): ❷ ❸ ❹
```

❶ Les classes peuvent aussi (et le devraient) avoir une `doc string`, comme les modules et les fonctions.

❷ `__init__` est appelé immédiatement après qu'une instance de la classe est créée. Il serait tentant mais incorrect de l'appeler le constructeur de la classe. Tentant, parceque ça ressemble à un constructeur (par convention, `__init__` est la première méthode définie de la classe), ça se comporte comme un constructeur (c'est le premier morceau de code exécuté dans une nouvelle instance de la classe) et que ça sonne pareil ("init" fait penser à quelque chose comme un constructeur). Incorrect, parce qu'au moment ou `__init__` est appelé, l'objet à déjà été créé et qu vous avez déjà une référence valide à la nouvelle instance de la classe. Mais `__init__` est ce qui se rapproche le plus d'un constructeur en Python et remplit en gros le même rôle.

❸ Le premier argument de chaque méthode de classe, y compris `__init__`, est toujours une référence à l'instance actuelle de la classe. Par convention, cet argument est toujours nommé `self`. Dans la méthode `__init__`, `self` fait référence à l'objet nouvellement créé, dans les autres méthodes de classe, il fait référence à l'instance dont la méthode a été appelée. Bien que vous deviez spécifier `self` explicitement lorsque vous définissez la méthode, vous ne devez *pas* le spécifier lorsque vous appelez la méthode, Python l'ajoutera pour vous automatiquement.

❹ Une méthode `__init__` peut prendre n'importe quel nombre d'arguments et tout comme pour les fonctions, les arguments peuvent être définis avec des valeurs par défaut, ce qui les rend optionnels lors de l'appel. Ici `filename` a une valeur par défaut de `None`, la valeur nulle de Python.

Par convention, le premier argument d'une méthode de classe (la référence à l'instance en cours) est appelé `self`. Cet argument remplit le rôle du mot réservé `this` en C++ ou Java, mais `self` n'est pas un mot réservé de Python, seulement une convention de nommage. Cependant, veuillez ne pas l'appeler autre chose que `self`, c'est une très forte convention.

Exemple 5.6. Ecriture de la classe `FileInfo`

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)        ❶
        self["name"] = filename        ❷
                                       ❸
```

❶ Certain langage pseudo−orientés objet comme Powerbuilder ont un concept d'"extension" des constructeurs et autres évènements, dans lequel la méthode de l'ancêtre est appelée automatiquement avant que la méthode du descendant soit exécutée. Python n'a pas ce comportement, vous devez appeler la méthode appropriée de l'ancêtre explicitement.

❷ Je vous ai dit que cette classe se comportait comme un dictionnaire, en voici le premier signe. Nous assignons l'argument `filename` comme valeur de la clé `name` de cet objet.

❸ Notez que la méthode `__init__` ne retourne jamais de valeur.

5.3.2. Quand utiliser `self` et `__init__`

Lorsque vous définissez vos méthodes de classe, vous *devez* indiquer explicitement `self` comme premier argument de chaque méthode, y compris `__init__`. Quand vous appelez une méthode d'une classe ancêtre depuis votre classe, vous *devez* inclure l'argument `self`. Mais quand vous appelez votre méthode de classe de l'extérieur, vous ne spécifiez rien pour l'argument self, vous l'omettez complètement et Python ajoute automatiquement la référence d'instance. Je me rends bien compte qu'on s'y perd au début, ce n'est pas réellement incohérent même si cela peut sembler l'être car cela est basé sur une distinction (entre méthode liée et non liée) que vous ne connaissez pas pour l'instant.

Ouf. Je sais bien que ça fait beaucoup à absorber, mais vous ne tarderez pas à comprendre tout ça. Toutes les classes Python fonctionnent de la même manière, donc quand vous en avez appris une, vous les connaissez toutes. Mais même si vous oubliez tout le reste souvenez vous de ça, car ça vous jouera des tours :

Les méthodes `__init__` sont optionnelles, mais quand vous en définissez une, vous devez vous rappeler d'appeler explicitement la méthode `__init__` de l'ancêtre de la classe. C'est une règle plus générale : quand un descendant veut étendre le comportement d'un ancêtre, la méthode du descendant doit appeler la méthode de l'ancêtre explicitement au moment approprié, avec les arguments appropriés.

Pour en savoir plus sur les classes Python

- *Learning to Program* (http://www.freenetpages.co.uk/hp/alan.gauld/) a une introduction en douceur aux classes (http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm).
- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) montre comment utiliser des classes pour modéliser des types composés (http://www.ibiblio.org/obp/thinkCSpy/chap12.htm).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) a un aperçu en profondeur des classes, des espaces de noms et de l'héritage (http://www.python.org/doc/current/tut/node11.html).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond à des questions courantes à propos des classes (http://www.faqts.com/knowledge−base/index.phtml/fid/242).

5.4. Instantiation de classes

L'instanciation de classes en Python est simple et directe. Pour instancier une classe, appelez simplement la classe comme si elle était une fonction, en lui passant les arguments que la méthode `__init__` définit. La valeur de retour sera l'objet nouvellement créé.

Exemple 5.7. Création d'une instance de `FileInfo`

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") ❶
>>> f.__class__                                        ❷
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__                                          ❸
'store file metadata'
>>> f                                                  ❹
{'name': '/music/_singles/kairo.mp3'}
```

❶ Nous créons une instance de la classe `FileInfo` (définie dans le module `fileinfo`) et assignons l'instance nouvellement créée à la variable `f`. Nous passons un paramètre, `/music/_singles/kairo.mp3`, qui sera l'argument `filename` de la méthode `__init__` de `FileInfo`.

❷ Chaque instance de classe à un attribut prédéfini, `__class__`, qui est la classe de l'objet (notez que la représentation de cet attribut comprend l'adresse physique de l'instance sur ma machine, votre sortie sera différente). Les programmeurs Java sont sans doute familiers de la classe `Class`, qui contient des méthodes comme `getName` et `getSuperclass` permettant d'obtenir les métadonnées d'un objet. En Python, ce type de métadonnées est accessible directement par le biais de l'objet lui−même à travers des attributs comme `__class__`, `__name__` et `__bases__`.

❸ Vous pouvez accéder à la `doc string` de l'instance comme pour une fonction ou un module. Toutes les instances d'une classe partagent la même `doc string`.

❹ Rappelez vous quand la méthode `__init__` a assigné son argument `filename` à `self["name"]`. Voici le résultat. Les arguments que nous passons lorsque nous créons une instance de classe sont envoyés directement à la méthode `__init__` (en même temps que la référence à l'objet, `self`, que Python ajoute automatiquement).

En Python, vous appelez simplement une classe comme si c'était une fonction pour créer une nouvelle instance de la classe. Il n'y a pas d'opérateur `new` explicite comme pour C++ ou Java.

5.4.1. Ramasse−miettes

Si créer des instances est simple, les détruire est encore plus simple. En général, il n'y a pas besoin de libérer explicitement les instances, elles sont libérées automatiquement lorsque les variables auxquelles elles sont assignées sont hors de portée. Les fuites mémoire sont rares en Python.

Exemple 5.8. Tentative d'implémentation d'une fuite mémoire

```
>>> def leakmem():
...     f = fileinfo.FileInfo('/music/_singles/kairo.mp3') ❶
...
>>> for i in range(100):
...     leakmem()                                          ❷
```

❶ A chaque fois que la fonction `leakmem` est appelée, nous créons une instance de `FileInfo` et l'assignons à la variable `f`, qui est une variable locale à la fonction. La fonction s'achève sans jamais libérer `f`, vous pourriez donc vous attendre à une fuite mémoire, mais vous auriez tort. Lorsque la fonction se termine, la variable locale `f` est hors de portée. A ce moment, il n'y a plus de référence à l'instance nouvellement créée de `FileInfo` (puisque nous ne l'avons jamais assignée à autre chose qu'à `f`), Python détruit alors l'instance pour nous.

❷ Peu importe le nombre de fois que nous appelons la fonction `leakmem`, elle ne provoquera jamais de fuite mémoire puisque Python va détruire à chaque fois la nouvelle instance de la classe `FileInfo` avant le retour de `leakmem`.

Le terme technique pour cette forme de ramasse−miettes est "comptage de référence". Python maintien une liste des références à chaque instance créée. Dans l'exemple ci−dessus, il n'y avait qu'une référence à l'instance de `FileInfo` : la variable locale `f`. Quand la fonction se termine, la variable `f` sort de la portée, le compteur de référence descend alors à `0` et Python détruit l'instance automatiquement.

Dans des versions précédentes de Python, il y avait des situations où le comptage de référence échouait et Python ne pouvait pas nettoyer derrière vous. Si vous créiez deux instances qui se référençaient mutuellement (par exemple une liste doublement chaînée où chaque noeud a un pointeur vers le noeud prochain et le précédent dans la liste), aucune des deux instances n'était jamais détruite car Python pensait (correctement) qu'il y avait toujours une référence à chaque instance. Python 2.0 a une forme additionnele de ramasse−miettes appelée "mark−and−sweep" qui est assez intelligente pour remarquer ce blocage et nettoyer correctement les références circulaires.

En tant qu'ancien étudiant en Philosophie, cela me dérange de penser que les choses disparaissent quand personne ne les regarde, mais c'est exactement ce qui se passe en Python. En général, vous pouvez simplement ignorer la gestion mémoire et laisser Python nettoyer derrière vous.

Pour en savoir plus sur le ramasse−miettes

- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume les attributs prédéfinis comme `__class__` (http://www.python.org/doc/current/lib/specialattrs.html).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `gc` module (http://www.python.org/doc/current/lib/module−gc.html), qui vous donne un contrôle de bas niveau sur le ramasse−miettes de Python.

5.5. `UserDict` : une classe enveloppe

Comme vous l'avez vu, `FileInfo` est une classe qui se comporte comme un dictionnaire. Pour voir ça plus en profondeur, regardons la classe `UserDict` dans le module `UserDict`, qui est l'ancêtre de notre classe `FileInfo`. Cela n'a rien de spécial, la classe est écrite en Python et stockée dans un fichier `.py`, tout comme notre code. En fait, elle est stockée dans le répertoire `lib` de votre installation Python.

Dans l'IDE ActivePython sous Windows, vous pouvez ouvrir rapidement n'importe quel module dans votre chemin de bibliothèques avec File−>Locate... (**Ctrl−L**).

Exemple 5.9. Definition de la classe `UserDict`

```
class UserDict:                                 ❶
    def __init__(self, dict=None):              ❷
        self.data = {}                          ❸
        if dict is not None: self.update(dict) ❹ ❺
```

❶ Notez que `UserDict` est une classe de base, elle n'hérite d'aucune classe.

❷ Voici la méthode `__init__` que nous avons redéfini dans la classe `FileInfo`. Notez que la liste d'arguments dans cette classe ancêtre est différente de celle du descendant. Cela ne pose pas de problème, chaque classe dérivée peut avoir sa propre liste d'arguments, tant qu'elle appelle la méthode de l'ancêtre avec les arguments corrects. Ici, la classe ancêtre a un moyen de définir des valeurs initiales (en passant un dictionnaire à l'argument `dict`) ce que notre `FileInfo` n'exploite pas.

❸ Python supporte les données attributs (appelés "variables d'instance" en Java et Powerbuilder, "variables membres" en C++), qui sont des données propres à une instance spécifique de la classe. Dans ce cas, chaque instance de `UserDict` aura un attribut de données `data`. Pour référencer cet attribut depuis du code extérieur à la classe, vous devez le qualifier avec le nom de l'instance, *`instance`*`.data`, de la même manière que vous qualifiez une fonction avec son nom de module. Pour référencer un attribut de données depuis la classe, nous utilisons `self` pour le qualifier. Par convention, tous les données attributs sont initalisées à des valeurs raisonnables dans la méthode `__init__`. Cependant, ce n'est pas obligatoire, puisque les données attributs, comme les variables locales viennent à existence lorsqu'on leur assigne une valeur pour la première fois.

❹ La méthode `update` est un duplicateur de dictionnaire. Elle copie toutes les clés et valeurs d'un dictionnaire à l'autre. Cela n'efface *pas* le dictionnaire de destination si il a déjà des clés, celles qui sont présente dans le dictionnaire source seront récrites, mais les autres ne seront pas touchées. Considérez `update` comme une fonction de fusion, pas de copie.

❺ Voici une syntaxe que vous n'avez peut−être pas vu auparavant (je ne l'ai pas employé dans les exemples de ce livre). C'est une instruction `if`, mais au lieu d'avoir un bloc indenté commençant à la ligne suivante, il y a juste une instruction unique sur la même ligne après les deux points. C'est une syntaxe tout à fait légale, c'est juste un raccourci lorsque vous n'avez qu'une instruction dans un bloc (comme donner une instruction unique sans accolades en C++). Vous pouvez employer cette syntaxe ou vous pouvez avoir du code indenté sur les lignes suivantes, mais vous ne pouvez pas mélanger les deux dans le même bloc.

Java et Powerbuilder supportent la surcharge de fonction par liste d'arguments : une classe peut avoir différentes méthodes avec le même nom mais avec un nombre différent d'arguments ou des arguments de type différent. D'autres langages (notamment PL/SQL) supportent même la surcharge de fonction par nom d'argument : une classe peut avoir différentes méthodes avec le même nom et le même nombre d'arguments du même type mais avec des noms d'arguments différents. Python ne supporte ni l'une ni l'autre, il n'a tout simplement aucune forme de surcharge de fonction. Les méthodes sont définies uniquement par leur nom et il ne peut y avoir qu'une méthode par classe avec le même nom. Donc si une classe descendante a une méthode `__init__`, elle redéfinit *toujours* la méthode `__init__` de la classe ancêtre, même si la descendante la définit avec une liste d'arguments différente. Et la même règle s'applique pour toutes les autres méthodes.

Guido, l'auteur originel de Python, explique la redéfinition de méthode de cette manière : "Les classes dérivées peuvent redéfinir les méthodes de leur classes de base. Puisque les méthodes n'ont pas de privilèges spéciaux lorsqu'elles appellent d'autres méthodes du même objet, une méthode d'une classe de base qui appelle une autre méthode définie dans cette même classe de base peut en fait se retrouver à appeler une méthode d'une classe dérivée qui la redéfini (pour les programmeurs C++ cela veut dire qu'en Python toutes les méthodes sont virtuelles)." Si cela n'a pas de sens pour vous (personellement, je m'y perd complètement) vous pouvez ignorer la question. Je me suis juste dit que je ferais circuler l'information.

Assignez toujours une valeur initiale à toutes les données attributs d'une instance dans la méthode `__init__`. Cela vous épargera des heures de débogage plus tard, à la poursuite d'exceptions `AttributeError` pour cause de référence à des attributs non−initialisés (et donc non−existants).

Exemple 5.10. Méthodes ordinaires de `UserDict`

```
    def clear(self): self.data.clear()          ❶
    def copy(self):                             ❷
        if self.__class__ is UserDict:          ❸
            return UserDict(self.data)
        import copy                             ❹
        return copy.copy(self)
    def keys(self): return self.data.keys()     ❺
    def items(self): return self.data.items()
    def values(self): return self.data.values()
```

❶ `clear` est une méthode de classe ordinaire, elle est disponible publiquement et peut être appelée par n'importe qui. Notez que `clear`, comme toutes les méthodes de classe, a pour premier argument `self` (rappelez−vous que vous ne mentionnez pas `self` lorsque vous appelez la méthode, Python l'ajoute pour vous). Notez aussi la technique de base employé par cette classe enveloppe : utiliser un véritable dictionnaire (`data`) comme données attributs, définir toutes les méthodes d'un véritable dictionnaire et rediriger chaque méthode vers la méthode du véritable dictionnaire (au cas où vous l'auriez oublié, la méthode `clear` d'un dictionnaire supprime toutes ses clés et leurs valeurs associées).

❷ La méthode `copy` d'un véritable dictionnaire retourne un nouveau dictionnaire qui est un double exact de l'original (avec les mêmes paires clé−valeur). Mais `UserDict` ne peut pas simplement rediriger la méthode vers `self.data.copy`, car cette méthode retourne un véritable dictionnaire, alors que nous voulons retourner une nouvelle instance qui soit de la même classe que `self`.

❸ Nous utilisons l'attribut `__class__` pour voir si `self` est un `UserDict` et, dans ce cas, tout va bien puisque nous savons comment copier un `UserDict` : il suffit de créer un nouveau `UserDict` et de lui passer le dictionnaire véritable qu'est `self.data`.

❹ Si `self.__class__` n'est pas un `UserDict`, alors `self` doit être une classe dérivée de `UserDict` (par exemple `FileInfo`), dans ce cas c'est plus compliqué. `UserDict` ne sait pas comment faire une copie exacte d'un de ses descendants. Il pourrait y avoir, par exemple, d'autres données attributs définies dans la classe dérivée, ce qui nécessiterait de les copier tous. Heureusement Python est fourni avec un module qui remplit cette tâche, le module `copy`. Je ne vais pas entrer ici dans les détails (bien que ce soit très intéressant et vaille la peine que vous y jetiez un coup d'œil). Il suffit de dire que `copy` peut copier un objet Python quelconque et que c'est comme cela que nous l'employons ici.

❺ Le reste des méthodes est sans difficulté, les appels sont redirigés vers les méthodes de `self.data`.

Dans les versions de Python antérieures à la 2.2, vous ne pouviez pas directement dériver les types de données prédéfinis comme les chaînes, les listes et les dictionnaires. Pour compenser cela, Python est fourni avec des classes enveloppes qui reproduisent le comportement de ces types de données prédéfinis : `UserString`, `UserList` et `UserDict`. En utilisant un mélange de méthodes ordinaires et spéciales, la classe `UserDict` fait une excellente imitation d'un dictionnaire, mais c'est juste une classe comme les autres, vous pouvez donc la dériver pour créer des classes personalisées semblables à un dictionnaire comme `FileInfo`. En Python 2.2 et suivant, vous pourriez récrire l'exemple de ce chapitre de manière à ce que `FileInfo` hérite directement de `dict` au lieu de `UserDict`. Cependant, vous devriez quand même lire l'explication du fonctionnement de `UserDict` au cas où vous auriez besoin d'implémenter ce genre d'objet enveloppe ou au cas où vous auriez à travailler avec une version de Python antérieure à la 2.2.

En Python il est possible de dériver une classe directement du type de données prédéfini `dict`, comme dans l'exemple suivant. Il y a trois différence avec la version dérivée de `UserDict`.

Exemple 5.11. Dériver une classe directement du type prédéfini `dict`

```
class FileInfo(dict):                   ❶
    "store file metadata"
    def __init__(self, filename=None):  ❷
        self["name"] = filename
```

❶ La première différence est que nous n'avons pas besoin d'importer le module `UserDict`, puisque `dict` est un type prédéfini et donc toujours disponible. La seconde est que nous dérivons notre classe de `dict` directement et non de `UserDict.UserDict`.

❷ La troisième différence est subtile mais importante. À cause de la manière dont `UserDict` fonctionne en interne, nous devons appeler explicitement sa méthode `__init__` pour l'initialiser correctement. `dict` ne fonctionne pas de la même manière, ce n'est pas une enveloppe et il ne demande pas d'initialisation explicite.

Pour en savoir plus sur `UserDict`

- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `UserDict` (http://www.python.org/doc/current/lib/module−UserDict.html) et le module `copy` (http://www.python.org/doc/current/lib/module−copy.html).

5.6. Méthodes de classe spéciales

En plus des méthodes de classe ordinaires, il y a un certain nombre de méthodes spéciales que les classes Python peuvent définir. Au lieu d'être appelées directement par votre code (comme les méthodes ordinaires) les méthodes spéciales sont appelées pour vous par Python dans des circonstances particulières ou quand une syntaxe spécifique est utilisée.

Comme vous l'avez vu dans la section précédente, les méthodes ordinaires nous ont permis en grande partie d'envelopper un dictionnaire dans une classe. Mais les méthodes ordinaires seules ne suffisent pas parce qu'il y a beaucoup de choses que vous pouvez faire avec un dictionnaire en dehors d'appeler ses méthodes. Pour commencer vous pouvez lire (get) et écrire (set) des éléments à l'aide d'une syntaxe qui ne fait pas explicitement appel à des méthodes. C'est là que les méthodes de classe spéciales interviennent : elle fournissent un moyen de faire correspondre la syntaxe n'appelant pas de méthodes à des appels de méthodes.

5.6.1. Lire et écrire des éléments

Exemple 5.12. La méthode spéciale `__getitem__`

```
    def __getitem__(self, key): return self.data[key]

>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("name") ❶
'/music/_singles/kairo.mp3'
>>> f["name"]             ❷
'/music/_singles/kairo.mp3'
```

❶ La méthode spéciale `__getitem__` à l'air simple. Comme les méthodes ordinaires `clear`, `keys` et `values`, elle ne fait que rediriger vers le dictionnaire pour obtenir sa valeur. Mais comment est−elle appelée ? Vous pouvez appeler `__getitem__` directement, mais en pratique vous ne le ferez pas, je le fais ici seulement pour vous montrer comment ça marche. La bonne manière d'utiliser `__getitem__` est d'obtenir de Python qu'il fasse l'appel pour vous.

❷ Cela à l'apparence exacte de la syntaxe que l'on utilise pour obtenir une valeur d'un dictionnaire et cela retourne bien la valeur que l'on attend. Mais il y a un chaînon manquant : en coulisse, Python convertit cette syntaxe en un appel de méthode `f.__getitem__("name")`. C'est pourquoi `__getitem__` est une méthode de classe spéciale, non seulement vous pouvez l'appeler vous−même, mais vous pouvez faire en sorte que Python l'appelle pour vous grâce à la syntaxe appropriée.

Bien sûr, Python a une méthode spéciale `__setitem__` pour accompagner `__getitem__`, comme nous le montrons dans l'exemple suivant.

Exemple 5.13. La méthode spéciale `__setitem__`

```
    def __setitem__(self, key, item): self.data[key] = item

>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31) ❶
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32            ❷
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

❶ Comme la méthode `__getitem__`, `__setitem__` redirige simplement l'appel au véritable dictionnaire `self.data`. Et comme pour `__getitem__`, vous n'appelez pas directement cette méthode en général, Python appelle `__setitem__` pour vous lorsque vous utilisez la bonne syntaxe.

❷ Cela ressemble à la syntaxe habituelle d'utilisation d'un dictionnaire, mais en fait `f` est une classe faisant de son mieux pour passer pour un dictionnaire et `__setitem__` est un élément essentiel de cette apparence. Cette ligne de code appelle en fait `f.__setitem__("genre", 32)` en coulisse.

`__setitem__` est une méthode de classe spéciale car elle est appelée pour vous, mais c'est quand même une méthode de classe. Nous pouvons la redéfinir dans une classe descendante tout aussi facilement qu'elle a été définie dans `UserDict`. Cela nous permet de définir des classes qui se comportent en partie comme des dictionnaires, mais qui ont leur propre comportement dépassant le cadre d'un simple dictionnaire.

Ce concept est à la base de tout le *framework* que nous étudions dans ce chapitre. Chaque type de fichier peut avoir une classe de manipulation qui sait comment obtenir des méta−données d'un type particulier de fichier. Une fois certains attributs (comme le nom et l'emplacement du fichier) connus, la classe de manipulation sait comment obtenir les autres attributs automatiquement. Cela se fait en redéfinissant la méthode `__setitem__`, en cherchant des clés particulières et en ajoutant un traitement supplémentaire quand elles sont trouvées.

Par exemple, `MP3FileInfo` est un descendant de `FileInfo`. Quand le nom (`name`) d'un `MP3FileInfo` est défini, cela ne change pas seulement la valeur de la clé `name` (comme pour l'ancêtre `FileInfo`), mais déclenche la recherche de balises MP3 et définit tout un ensemble de clés.

Exemple 5.14. Redéfinition de `__setitem__` dans `MP3FileInfo`

```
    def __setitem__(self, key, item):          ❶
        if key == "name" and item:             ❷
            self.__parse(item)                 ❸
        FileInfo.__setitem__(self, key, item)  ❹
```

❶ Notez que notre méthode `__setitem__` est définie exactement comme la méthode héritée. C'est important car Python appellera la méthode pour nous et qu'il attend un certain nombre d'arguments (techniquement parlant, les noms des arguments n'ont pas d'importance, seulement leur nombre).

❷ Voici le point crucial de toute la classe `MP3FileInfo` : si nous assignons une valeur à la clé `name`, alors nous voulons faire quelque chose en plus.

❸ Le traitement supplémentaire que nous faisons pour les noms (`name`) est encapsulé dans la méthode `__parse`. C'est une autre méthode de classe définie dans `MP3FileInfo` et quand nous l'appelons nous la qualifions avec `self`. Un appel à `__parse` tout court chercherait une fonction ordinaire définie hors de la classe, ce qui n'est pas ce que nous voulons, appeler `self.__parse` cherchera une méthode définie dans la classe. Cela n'a rien de nouveau, c'est de la même manière que l'on fait référence aux données attributs.

❹ Après avoir fait notre traitement supplémentaire, nous voulons appeler la méthode héritée. Rappelez−vous que Python ne le fait jamais pour vous, vous devez le faire manuellement. Notez que nous appelons l'ancêtre immédiat, `FileInfo`, même si il n'a pas de méthode `__setitem__`. Cela fonctionne parce que Python va remonter la hierarchie d'heritage jusqu'à ce qu'il trouve une classe avec la méthode que nous appelons, cette ligne finira donc par trouver et appeler la méthode `__setitem__` définie dans `UserDict`.

Lorsque vous accédez à des données attributs dans une classe, vous devez qualifier le nom de l'attribut : `self.`*`attribute`*. Lorsque vous appelez d'autres méthodes dans une classe, vous devez qualifier le nom de la méthode : `self.`*`method`*.

Exemple 5.15. Setting an `MP3FileInfo`'s `name`

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()                    ❶
>>> mp3file
{'name':None}
>>> mp3file["name"] = "/music/_singles/kairo.mp3"       ❷
>>> mp3file
{'album': 'Rave Mix', 'artist': '***DJ MARY−JANE***', 'genre': 31,
'title': 'KAIRO****THE BEST GOA', 'name': '/music/_singles/kairo.mp3',
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3" ❸
>>> mp3file
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title': 'Sidewinder',
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

❶ D'abord nous créons une instance de `MP3FileInfo` sans lui passer de nom de fichier (nous pouvons le faire parce que l'argument `filename` de la méthode `__init__` est optionnel). Comme `MP3FileInfo` n'a pas de méthode `__init__` propre, Python remonte la hierarchie d'héritage et trouve la méthode `__init__` de `FileInfo`. Cette méthode `__init__` appelle manuellement la méthode `__init__` de `UserDict` puis définit la clé `name` à la valeur de `filename`, qui est `None` puisque nous n'avons passé aucun nom de fichier. Donc `mp3file` est au début un dictionnaire avec une clé, `name`, dont la valeur est `None`.

❷ Maintenant les choses sérieuses commencent. Définir la clé `name` de `mp3file` déclenche la méthode `__setitem__` de `MP3FileInfo` (pas `UserDict`), qui remarque que nous définissons la clé `name` avec une valeur réelle et appelle `self.__parse`. Bien que nous n'ayons pas encore vu le contenu de la méthode `__parse`, vous pouvez voir à partir de la sortie qu'elle définit plusieurs autres clés : `album`, `artist`, `genre`, `title`, `year` et `comment`.

❸ Modifier la clé `name` recommencera le même processus : Python appelle `__setitem__`, qui appelle `self.__parse`, qui définit toutes les autres clés.

5.7. Méthodes spéciales avancées

Il y a d'autres méthodes spéciales que `__getitem__` et `__setitem__`. Certaines vous laissent émuler des fonctionnalité dont vous ignorez encore peut−être tout.

Cet exemple montre certaines des autres méthodes spéciales de `UserDict`.

Exemple 5.16. D'autres méthodes spéciales dans `UserDict`

```
def __repr__(self): return repr(self.data)      ❶
def __cmp__(self, dict):                        ❷
    if isinstance(dict, UserDict):
        return cmp(self.data, dict.data)
    else:
        return cmp(self.data, dict)
def __len__(self): return len(self.data)        ❸
def __delitem__(self, key): del self.data[key] ❹
```

❶ `__repr__` est une méthode spéciale qui est appelée lorsque vous appelez `repr(`*`instance`*`)`. La fonction `repr` est une fonction prédéfinie qui retourne une représentation en chaîne d'un objet. Elle fonctionne pour tout objet, pas seulement les instances de classes. En fait, vous êtes déjà familier de `repr`, même si vous l'ignorez. Dans la fenêtre interactive, lorsque vous tapez juste un nom de variable et faites **Entrée**, Python utilise `repr` pour afficher la valeur de la variable. Créez un dictionnaire `d` avec des données, puis faites `print repr(d)` pour le voir par vous même.

❷ `__cmp__` est appelé lorsque vous comparez des instances de classe. En général, vous pouvez comparer deux objets Python quels qu'ils soient, pas seulement des instances de classe, en utilisant `==`. Il y a des règles qui définissent quand les types de données prédéfinis sont considérés égaux. Par exemple, les dictionnaires sont égaux quand ils ont les mêmes clés et valeurs, les chaînes sont égales quand elles ont la même longueur et contiennent la même séquence de caractères. Pour les instances de classe, vous pouvez définir la méthode `__cmp__` et écrire la logique de comparaison vous−même et vous pouvez ensuite utiliser `==` pour comparer des instances de votre classe, Python appelera votre méthode spéciale `__cmp__` pour vous.

❸ `__len__` est appelé lorsque vous appelez `len(`*`instance`*`)`. La fonction `len` est une fonction prédéfinie qui retourne la longueur d'un objet. Elle fonctionne pour tout objet pour lequel il est envisageable de penser qu'il a une longueur. La `len` d'une chaîne est son nombre de caractères, la `len` d'un dictionnaire est son nombre de clés et la `len` d'une liste ou tuple est son nombre d'éléments. Pour les instances de classe, définissez la méthode `__len__` et écrivez le calcul de longueur vous−même, puis appelez `len(`*`instance`*`)` et Python appelera votre méthode spéciale `__len__` pour vous.

❹ `__delitem__` est appelé lorsque vous appelez `del` *`instance`*`[`*`key`*`]`, ce qui, vous vous en rappelez peut−être, est le moyen de supprimer des éléments individuels d'un dictionnaire. Quand vous utilisez `del` sur une instance de classe, Python appelle la méthode spéciale `__delitem__` pour vous.

En Java, vous déterminez si deux variables de chaînes référencent la même zone mémoire à l'aide de `str1 == str2`. On appelle cela *identité* des objets et la syntaxe Python en est `str1 is str2`. Pour comparer des valeurs de chaînes en Java, vous utiliseriez `str1.equals(str2)`, en Python, vous utiliseriez `str1 == str2`. Les programmeurs Java qui ont appris que le monde était rendu meilleur par le fait que `==` en Java fasse une comparaison par identité plutôt que par valeur peuvent avoir des difficultés à s'adapter au fait que Python est dépourvu d'un tel piège.

Vous trouvez peut−être que ça fait beaucoup de travail pour faire avec une classe ce qu'on peut faire avec un type de données prédéfini. Et c'est vrai que tout serait plus simple (et la classe `UserDict` serait inutile) si on pouvait hériter d'un type de données prédéfini comme un dictionnaire. Mais même si vous pouviez le faire, les méthodes spéciales seraient toujours utiles, car elles peuvent être utilisées dans n'importe quelle classe, pas seulement dans une classe enveloppe comme `UserDict`.

Les méthodes spéciales permettent à *toute classe* de stocker des paires clé−valeur comme un dictionnaire, simplement en définissant la méthode `__setitem__`. *Toute classe* peut se comporter comme une séquence, simplement en définissant la méthode `__getitem__`. Toute classe qui définit la méthode `__cmp__` peut être comparée avec `==`. Et si votre classe représente quelque chose qui a une longueur, ne créez pas une méthode `GetLength`, définissez la méthode `__len__` et utilisez `len(`*`instance`*`)`.

Alors que les autres langages orientés objet ne vous laissent définir que le modèle physique d'un objet ("cet objet a une méthode `GetLength`"), les méthodes spéciales de Python comme `__len__` vous permettent de définir le modèle logique d'un objet ("cet objet a une longueur").

Il y a de nombreuses autres méthodes spéciales. Un ensemble de ces méthodes permet aux classes de se comporter comme des nombres, permettant l'addition, la soustraction et autres opérations arithmétiques sur des instances de classe (l'exemple type en est une classe représentant les nombres complexes, nombres ayant à la fois un composant réel et imaginaire). La méthode `__call__` permet à une classe de se comporter comme une fonction, ce qui permet d'appeler une instance de classe directement. Il y a aussi d'autres méthodes spéciales permettant aux classes d'avoir des données attributs en lecture seule ou en écriture seule, nous en parlerons dans des chapitres à venir.

Pour en savoir plus sur les méthodes de classe spéciales

- La *Python Reference Manual* (http://www.python.org/doc/current/ref/) documente toutes les méthodes spéciales de classe (http://www.python.org/doc/current/ref/specialnames.html).

5.8. Attributs de classe

Vous connaissez déjà les données attributs, qui sont des variables appartenant à une instance particulière d'une classe. Python permet aussi les attributs de classe, qui sont des variables appartenant à la classe elle−même.

Exemple 5.17. Présentation des attributs de classe

```
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

>>> import fileinfo
>>> fileinfo.MP3FileInfo                ❶
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap     ❷
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built−in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()          ❸
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built−in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

❶ `MP3FileInfo` est la classe elle−même, pas une instance particulière de cette classe.

❷ `tagDataMap` est un attribut de classe : littéralement, un attribut de la classe. Il est diponible avant qu'aucune instance de la classe n'ait été créée.

❸ Les attributs de classe sont disponibles à la fois par référence directe à la classe et par référence à une instance

quelconque de la classe.

En Java, les variables statiques (appelées attributs de classe en Python) aussi bien que les variables d'instance (appelées données attributs en Python) sont définies immédiatement après la définition de la classe (avec le mot−clé `static` pour les premières). En Python, seuls les attributs de classe peuvent être définis à cet endroit, les données attributs sont définies dans la méthode `__init__`.

Les attributs de classe peuvent être utilisés comme des constantes au niveau de la classe (ce qui est la manière dont nous les utilisons dans `MP3FileInfo`), mais ils ne sont pas vraiment des constantes. Vous pouvez également les modifier.

Il n'y a pas de constantes en Python. Tout est modifiable en faisant un effort. C'est en accord avec un des principes essentiels de Python : un mauvais comportement doit être découragé mais pas interdit. Si vous voulez vraiment changer la valeur de `None`, vous pouvez le faire, mais ne venez pas vous plaindre que votre code est impossible à déboguer.

Exemple 5.18. Modification des attributs de classe

```
>>> class counter:
...     count = 0                       ❶
...     def __init__(self):
...         self.__class__.count += 1   ❷
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count                       ❸
0
>>> c = counter()
>>> c.count                             ❹
1
>>> counter.count
1
>>> d = counter()                       ❺
>>> d.count
2
>>> c.count
2
>>> counter.count
2
```

❶ `count` est un attribut de la classe `counter`.

❷ `__class__` est un attribut prédéfini de toute instance de classe (de toute classe). C'est une référence à la classe dont `self` est une instance (dans ce cas, la classe `counter`).

❸ Comme `count` est un attribut de classe, il est disponible par référence directe à la classe, avant que nous ayions créé une instance de la classe.

❹ Créer une instance de la classe appelle la méthode `__init__`, qui incrémente l'attribut de classe `count` de `1`. Cela affecte la classe elle−même, pas seulement l'instance nouvellement créée.

❺ Créer une seconde instance incrémente à nouveau l'attribut de classe `count`. Vous constatez que l'attribut de classe est partagé par la classe et toutes ses instances.

5.9. Fonctions privées

Comme la plupart des langages, Python possède le concept d'éléments privées :

- des fonctions privées, qui ne peuvent être appelées de l'extérieur de leur module ;

- des méthodes de classe privées, qui ne peuvent être appelées de l'extérieur de leur classe ;
- des attributs privés, qui ne peuvent être accédé de l'extérieur de leur classe.

Contrairement à la plupart des langages, le caractère privé ou public d'une fonction, d'une méthode ou d'un attribut est déterminé en Python entièrement par son nom.

Si le nom d'une fonction, d'une méthode de classe ou d'un attribut commence par (mais ne se termine pas par) deux caractères de soulignement il s'agit d'un élément privé, tout le reste est public. Python ne possède pas de concept de méthode *protégée* (accessible seulement à la même classe ou a ses descendants). Les méthodes d'une classe sont ou privées (accessibles seulement à la même classe) ou publiques (accessibles à tout le monde).

`MP3FileInfo` a deux méthodes : `__parse` et `__setitem__`. Comme nous en avons déjà discuté, `__setitem__` est une méthode spéciale, vous l'appelez normalement indirectement en utilisant la syntaxe de dictionnaire sur une instance de classe, mais elle est publique et vous pouvez l'appeler directement (même d'en dehors du module `fileinfo`) si vous avez une bonne raison de le faire. Par contre `__parse` est privée, car elle a deux caractères de soulignement au début de son nom.

En Python, toutes les méthodes spéciales (comme `__setitem__`) et les attributs prédéfinis (comme `__doc__`) suivent une convention standard : il commencent et se terminent par deux caractères de soulignement. Ne nommez pas vos propres méthodes et attributs de cette manière, cela n'apporterait que de la confusion pour vous et les autres.

Exemple 5.19. Tentative d'appel d'une méthode privée

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3") ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
      AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

❶ Si vous essayez d'appeler une méthode privée, Python lévera une exception un peu trompeuse disant que la méthode n'existe pas. Bien sûr elle existe, mais elle est privée, donc elle n'est pas accessible d'en dehors de la classe. A proprement parler, les méthodes privées sont accessibles d'en dehors de leur classe, mais pas *facilement* accessibles. Rien n'est vraiment privé en Python, en interne les noms des méthodes et attributs privés sont camouflés à la volée pour les rendre inaccessibles par leur nom d'origine. Vous pouvez accéder à la méthode `__parse` de la classe `MP3FileInfo` par le nom `_MP3FileInfo__parse`. Vous êtes prié de trouver ça intéressant mais de promettre de ne jamais, jamais l'utiliser dans votre code. Les méthodes privées ont une bonne raison de l'être, mais comme beaucoup d'autres choses en Python, leur caractère privé est en dernière instance une question de convention et non d'obligation.

Pour en savoir plus sur les fonctions privées

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite du fonctionnement des variables privées (http://www.python.org/doc/current/tut/node11.html#SECTION00116000000000000000).

5.10. Résumé

Voila pour ce qui est des chicanes techniques des objets. Vous verrez une application réelle des méthodes de classe spéciales au Chapitre 12, qui utilise `getattr` pour créer un mandataire d'un service web distant.

Le prochain chapitre continuera d'utiliser ce programme d'exemple pour explorer d'autres concepts de Python comme

les exceptions, les objets−fichiers et les boucles `for`.

Avant de plonger dans le prochain chapitre, assurez−vous que vous vous sentez à l'aise pour :

- Importer des modules en utilisant `import *module*` ou `from *module* import`
- Definir et instancier des classes
- Definir des méthodes `__init__` et autres méthodes spéciales et comprendre quand elles sont appelées
- Dériver de `UserDict` pour définir des classes qui se comportent comme des dictionnaires
- Définir des données attributs et des attributs de classe et comprendre la différence entre les deux
- Définir des méthodes privées

Chapitre 6. Traitement des exceptions et utilisation de fichiers

Dans ce chapitre vous plongerez dans les exceptions, les objets−fichiers, les boucles `for` et les modules `os` et `sys`. Si vous avez utilisé les exceptions dans un autre langage de programmation, vous pouvez survoler la première section pour avoir une idée de la syntaxe de Python. Assurez−vous de reprendre une lecture détaillée pour l'utilisation des fichiers.

6.1. Traitement des exceptions

Comme beaucoup de langages orientés objet, Python gère les exception à l'aide de blocs `try...except`.

Python utilise `try...except` pour gérer les exceptions et `raise` pour les générer. Java et C++ utilisent `try...catch` pour gérer les exceptions et `throw` pour les générer.
Les exceptions sont partout en Python, pratiquement chaque module de la librairie standard Python les utilise et Python lui−même en déclenchera dans de nombreuses circonstances différentes. Vous les avez déjà vu à plusieurs reprises tout au long de ce livre.

- Accéder à une clé non−existante d'un dictionnaire déclenche une exception `KeyError`.
- Chercher une valeur non−existante dans une liste déclenche une exception `ValueError`.
- Appeler une méthode non−existante déclenche une exception `AttributeError`.
- Référencer une variable non−existante déclenche une exception `NameError`.
- Mélanger les types de données sans conversion déclenche une exception `TypeError`.

Dans chacun de ces cas, nous ne faisions qu'expérimenter à l'aide de l'IDE Python : une erreur se produisait, l'exception était affichée (éventuellement, en fonction de votre IDE, dans un rouge détonnant) et c'était tout. C'est ce que l'on appelle une exception *non−gérée*, lorsque l'exception a été déclenchée, il n'y avait pas de code pour la prendre en charge explicitement, elle est donc remontée jusqu'à Python qui l'a traité selon la méthode par défaut, qui est d'afficher une information de débogage et d'abandonner. Dans l'IDE ce n'est pas un problème, mais si cela arrivait pendant le déroulement d'un de vos programmes Python réels, le programme dans son ensemble serait arrété.

Cependant, une exception ne doit pas forcément entrainer le plantage complet d'un programme. Les exceptions, lorsqu'elles sont déclenchées, peuvent être *gérées*. Parfois une exception se produit parcequ'il y a réellement un bogue dans votre code (comme tenter d'accéder à une variable qui n'existe pas), mais souvent, une exception est un évènement que vous pouvez prévoir. Si vous ouvrez un fichier, il peut ne pas exister, si vous vous connectez à une base de données, elle peut être indisponible, ou peut−être n'avez−vous pas les droits nécéssaires pour y accéder. Si vous savez qu'une ligne de code est susceptible de déclencher un exception, vous devriez gérer l'exception avec un bloc `try...except`.

Exemple 6.1. Ouverture d'un fichier inexistant

```
>>> fsock = open("/notthere", "r")      ❶
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
       IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
...     fsock = open("/notthere")       ❷
... except IOError:                     ❸
...     print "The file does not exist, exiting gracefully"
... print "This line will always print" ❹
```

```
The file does not exist, exiting gracefully
        This line will always print
```

❶ En utilisant la fonction prédéfinie `open`, nous pouvons ouvrir un fichier en lecture (nous verrons `open` plus en détail dans la section suivante). Mais le fichier n'existe pas, ce qui déclenche une exception `IOError`. Comme nous n'avons pas fourni de gestionnaire pour l'exception `IOError`, Python se contente d'afficher des informations de débogage et abandonne.

❷ Nous allons essayer d'ouvrir le même fichier non−existant, mais cette fois à l'intérieur d'un bloc `try...except`.

❸ Quand la méthode `open` déclenche une exception `IOError`, nous sommes prêts. La ligne `except IOError:` intercepte l'exception et exécute notre propre bloc de code, qui en l'occurence ne fait qu'afficher un message d'erreur plus agréable.

❹ Une fois qu'une exception a été traitée, le traitement continue normalement à la première ligne après le bloc `try...except`. Notez que cette ligne sera toujours affichée, qu'une exception se produise ou pas. Si vous aviez vraiment un fichier appelé `notthere` dans votre répertoire racine, l'appel a `open` réussirait, la clause `except` serait ignorée, mais cette ligne serait quand même exécutée.

Les exceptions peuvent sembler hostiles (après tout, si vous ne les interceptez pas, votre programme plante), mais réflechissez à l'alternative. Voudriez−vous plutôt un objet−fichier inutilisable pointant vers un fichier non−existant ? De toute manière, vous auriez quand même à vérifier sa validité, sinon votre programme produirait des erreurs bizarres plus loin dont vous auriez à retrouver la source. Je suis sûr que vous avez déjà fait cela, ce n'est pas drôle. Avec les exceptions, les erreurs se produisent immédiatement et vous pouvez les gérer de manière standardisée à la source du problème.

6.1.1. Utilisation d'exceptions pour d'autres cas que la gestion d'erreur

Il y a de nombreux autres usages pour les exceptions en dehors de la prise en compte de véritables conditions d'erreurs. Un des usages commun dans la bibliothèque standard Python est d'essayer d'importer un module, puis de vérifier si cela à marché. Importer un module qui n'existe pas déclenchera une exception `ImportError`. Vous pouvez utiliser cela pour définir des niveaux multiples de fonctionnalité basé sur la disponibilité des modules à l'exécution, ou pour supporter plusieurs plateformes (dans ce cas le code spécifique à chaque plateforme est séparé dans différents modules).

Vous pouvez aussi définir vos propre exceptions en créant un classe qui hérite de la classe prédéfinie `Exception` et déclencher vos exceptions avec l'instruction `raise`. Cela dépasse le champ de cette section, voyez la section "Pour en savoir plus" si vous êtes intéressé.

L'exemple suivant démontre l'utilisation d'une exception pour le support d'une fonctionnalité spécifique à une plate−forme. Ce code vient du module `getpass`, un module enveloppe pour obtenir un mot de passe de l'utilisateur. Obtenir un mot de passe est fait de manière différente sous UNIX, Windows et Mac OS, mais ce code encapsule toutes ces différences.

Exemple 6.2. Support de fonctionnalités propre à une plate−forme

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS                 ❶
except ImportError:
    try:
        import msvcrt                       ❷
    except ImportError:
        try:
            from EasyDialogs import AskPassword   ❸
```

```
            except ImportError:
                getpass = default_getpass ❹
            else:                         ❺
                getpass = AskPassword
        else:
            getpass = win_getpass
    else:
        getpass = unix_getpass
```

❶ `termios` est un module spécifique à UNIX qui fournit un contrôle de bas niveau sur le terminal d'entrée. Si ce module n'est pas disponible (parcequ'il n'est pas sur votre système ou que votre système ne le supporte pas), l'import échoue et Python déclenche une exception `ImportError`, que nous interceptons.

❷ OK, nous n'avons pas `termios`, essayons donc `msvcrt`, qui est un module spécifique à Windows qui fournit une API pour de nombreuses fonctions utiles des services d'exécution de Microsoft Visual C++. Si l'import échoue, Python déclenche une exception `ImportError`, que nous interceptons.

❸ Si les deux premiers n'ont pas marché, nous essayons d'importer une fonction de `EasyDialogs`, qui est un module spécifique à Mac OS qui fournit des fonctions pour afficher des boîtes de dialogue de différents types. Encore une fois, si cette import échoue, Python déclenche une exception `ImportError`, que nous interceptons.

❹ Aucun de ces modules spécifiques n'est disponible (ce qui est possible puisque Python a été porté sur de nombreuses plateformes), nous devons donc nous replier sur la fonction de saisie de mot de passe par défaut (qui est définie ailleurs dans le module `getpass`). Remarquez ce que nous faison là : nous assignons la fonction `default_getpass` à la variable `getpass`. Si vous lisez la documentation officielle de `getpass`, elle vous dit que le module `getpass` définit une fonction `getpass`. C'est comme ça qu'il le fait, en assignant `getpass` à la bonne fonction pour votre plateforme. Quand vous appelez ensuite la fonction `getpass`, vous appelez en fait une fonction spécifique à la plateforme que ce code a mis en place pour vous. Vous n'avez pas à vous soucier de la plateforme sur laquelle votre code est exécuté, appelez `getpass`, qui fera ce qu'il faut.

❺ Un bloc `try...except` peut avoir une clause `else`, comme une instruction `if`. Si aucune exception n'est déclenchée dans le bloc `try`, la clause `else` est exécutée à la suite. Dans ce cas, cela veut dire que l'import `from EasyDialogs import AskPassword` a fonctionné et donc nous assignons `getpass` à la fonction `AskPassword`. Chacun des autres blocs `try...except` a une clause `else` similaire pour assigner `getpass` à la bonne fonction lorsque nous trouvons un `import` qui marche.

Pour en savoir plus sur le traitement des exceptions

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de la définition et du déclenchement de vos propres exceptions et de la gestion de plusieurs exceptions à la fois (http://www.python.org/doc/current/tut/node10.html#SECTION00104000000000000000).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les exceptions prédéfinies (http://www.python.org/doc/current/lib/module−exceptions.html).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module getpass (http://www.python.org/doc/current/lib/module−getpass.html).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `traceback` (http://www.python.org/doc/current/lib/module−traceback.html), qui fournit un accès de bas niveau aux attributs d'une exception après qu'elle ait été déclenchée.
- La *Python Reference Manual* (http://www.python.org/doc/current/ref/) traite du fonctionnement interne du bloc `try...except` (http://www.python.org/doc/current/ref/try.html).

6.2. Les objets−fichier

Python a une fonction prédéfinie, `open`, pour ouvrir un fichier sur le disque. `open` retourne un objet−fichier qui possède des méthodes et des attributs pour obtenir des informations et manipuler le fichier ouvert.

Exemple 6.3. Ouverture d'un fichier

```
>>> f = open("/music/_singles/kairo.mp3", "rb") ❶
>>> f                                           ❷
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode                                      ❸
'rb'
>>> f.name                                      ❹
'/music/_singles/kairo.mp3'
```

❶ La méthode `open` peut prendre jusqu'à trois paramètres : un nom de fichier, un mode et un paramètre de tampon. Seul le premier, le nom de fichier, est nécéssaire, les deux autres sont optionnels. Si le mode n'est pas spécifié, le fichier est ouvert en mode texte pour la lecture. Ici nous ouvrons le fichier en mode binaire pour la lecture (`print open.__doc__` affiche une bonne explication de tous les modes possibles).

❷ La fonction `open` retourne un objet (arrivé à ce point cela ne doit pas vous surprendre). Un objet−fichier à plusieurs attributs utiles.

❸ L'attribut `mode` d'un objet−fichier vous indique dans quel mode le fichier a été ouvert.

❹ L'attribut `name` d'un objet−fichier vous indique le nom du fichier qui a été ouvert.

6.2.1. Lecture d'un fichier

Une fois un fichier ouvert, la première chose que l'on peut faire est de le lire, comme nous allons le voir dans l'exemple suivant.

Exemple 6.4. Lecture d'un fichier

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()                ❶
0
>>> f.seek(-128, 2)         ❷
>>> f.tell()                ❸
7542909
>>> tagData = f.read(128)   ❹
>>> tagData
'TAGKAIRO****THE BEST GOA          ***DJ MARY-JANE***
Rave Mix                      2000http://mp3.com/DJMARYJANE     \037'
>>> f.tell()                ❺
7543037
```

❶ Un objet−fichier maintien des informations d'état sur le fichier qui est ouvert. La méthode `tell` d'un objet−fichier vous indique la position actuelle dans le fichier ouvert. Comme nous n'avons encore rien fait de ce fichier la position actuelle est `0`, le début du fichier.

❷ La méthode `seek` d'un objet−fichier permet de se déplacer dans le fichier ouvert. Le deuxième paramètre précise ce que le premier signifie : `0` pour un déplacement à une position absolue (en partant du début du fichier), `1` pour une position relative (en partant de la position actuelle) et `2` pour une position relative à la fin du fichier. Puisque les balises MP3 que nous recherchons sont stockés à la fin du fichier, nous utilisons `2` et nous déplaçons à `128` octets de la fin du fichier.

❸ La méthode `tell` confirme que la position actuelle a changé.

❹ La méthode `read` lit un nombre d'octets spécifié du fichier ouvert et retourne une chaîne contenant les données lues. Le paramètre optionnel précise le nombre maximal d'octets à lire. Si aucun paramètre n'est spécifié, `read` lit jusqu'à la fin du fichier. (Nous aurions pu taper simplement `read()` ici, puisque nous savons exactement où nous sommes dans le fichier et que nous lisons en fait les 128 derniers octets.) Les données lues sont assignées à la variable `tagData` et la position actuelle est mise à jour en fonction du

nombre d'octets lus.

❺ La méthode `tell` confirme que la position actuelle a changé. Si vous faites le calcul, vous verrez qu'après que nous ayons lu 128 octets, la position a été incrémenté de 128.

6.2.2. Fermeture d'un fichier

Les fichiers ouverts consomment des ressources système et, en fonction du mode d'ouverture, peuvent ne pas être accessibles à d'autres programmes. Il est donc important de fermer les fichiers dès que vous ne les utilisez plus.

Exemple 6.5. Fermeture d'un fichier

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed       ❶
False
>>> f.close()      ❷
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed       ❸
True
>>> f.seek(0)      ❹
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close()      ❺
```

❶ L'attribut `closed` d'un objet−fichier indique si l'objet pointe un fichier ouvert ou non. Dans ce cas, le fichier est toujours ouvert (`closed` vaut `False`).

❷ Pour fermer un fichier, appelez la méthode `close` de l'objet−fichier. Cela libère le verrou (s'il existe) que vous avez sur le fichier, purge les tampons en écriture (s'ils existent) et libère les ressources système.

❸ L'attribut `closed` confirme que le fichier est fermé.

❹ Ce n'est pas parce que le fichier est fermé que l'objet fichier cesse d'exister. La variable `f` continuera d'exister jusqu'à ce qu'elle soit hors de portée ou qu'elle soit supprimée manuellement. Cependant aucune des méthodes de manipulation d'un fichier ouvert ne marchera après que le fichier ait été fermé, elles déclencheront toutes une exception.

❺ Appeler `close` sur un objet−fichier dont le fichier est déjà fermé ne déclenche *pas* d'exception mais échoue silencieusement.

6.2.3. Gestion des erreurs d'entrée/sortie

Maintenant vous en avez vu assez pour comprendre le code de gestion de fichier dans le programme d'exemple `fileinfo.py` du chapitre précédent. L'exemple suivant montre comment ouvrir et lire un fichier de manière sûre en gérant les erreurs.

Exemple 6.6. Les objets−fichier dans `MP3FileInfo`

```
try:                                    ❶
    fsock = open(filename, "rb", 0)     ❷
    try:
        fsock.seek(-128, 2)             ❸
        tagdata = fsock.read(128)       ❹
    finally:                            ❺
        fsock.close()
    .
    .
    .
except IOError:                         ❻
    pass
```

❶ Comme l'ouverture et la lecture de fichiers est risquée et peut déclencher une exception, tout ce code est enveloppé dans un bloc `try...except` (alors, l'indentation standardisée n'est-elle pas admirable ? C'est là que l'on commence a vraiment l'apprécier).

❷ La fonction `open` peut déclencher une exception `IOError` (peut-être que le fichier n'existe pas).

❸ La méthode `seek` peut déclencher une exception `IOError` (peut-être que le fichier fait moins de 128 octets).

❹ La méthode `read` peut déclencher une exception `IOError` (peut-être que le disque a un secteur défectueux ou le fichier est sur le réseau et le réseau est en rideau).

❺ Voilà qui est nouveau : un bloc `try...finally`. Une fois le fichier ouvert avec succès par la fonction `open`, nous voulons être absolument sûrs que nous le refermons, même si une exception est déclenchée par les méthodes `seek` ou `read`. C'est à cela que sert un bloc `try...finally` : le code du bloc `finally` sera *toujours* exécuté, même si une exception est déclenchée dans le bloc `try`. Pensez-y comme à du code qui est exécuté "au retour", quoi qu'il se soit passé "en route".

❻ Enfin, nous gérons notre exception `IOError`. Cela peut être l'exception `IOError` déclenchée par l'appel à `open`, `seek`, ou `read`. Ici, nous ne nous en soucions vraiment pas car tout ce que nous faisons et d'ignorer l'erreur et de continuer (rappelez-vous que `pass` est une instruction Python qui ne fait rien). C'est tout à fait légal, "gérer" une exception peut vouloir dire explicitement ne rien faire. Cela compte quand même comme une exception gérée et le traitement va reprendre normalement à la prochaine ligne de code après le bloc `try...except`.

6.2.4. Ecriture dans un fichier

Nous pouvons bien sûr écrire dans un fichier, cela se fait en grande partie de la même manière que pour la lecture. Il y a deux modes d'ouverture de base :

- Le mode *Append* (ajout) pour ajouter des données à la fin du fichier.
- Le mode *Write* (écriture) pour écraser le contenu du fichier.

Les deux modes créeront le fichier automatiquement s'il n'existe pas encore, il n'y a donc pas besoin de logique du type "si le fichier de journalisation n'existe pas, créer un nouveau fichier vide et l'ouvrir en écriture." Il suffit d'ouvrir le fichier pour commencer à y écrire.

Exemple 6.7. Ecriture dans un fichier

```
>>> logfile = open('test.log', 'w') ❶
>>> logfile.write('test succeeded') ❷
>>> logfile.close()
>>> print file('test.log').read()   ❸
test succeeded
>>> logfile = open('test.log', 'a') ❹
>>> logfile.write('line 2')
>>> logfile.close()
```

```
>>> print file('test.log').read()    ❺
test succeededline 2
```

❶ Nous commençons par créer un nouveau fichier `test.log` ou l'écraser s'il existe et l'ouvrir en écriture (le second paramètre `"w"` signifie ouverture en écriture). Effectivement, c'est aussi dangeureux que ça en a l'air. J'espère que vous n'aviez rien de précieux dans ce fichier, parce que maintenant c'est effacé.

❷ Vous pouvez écrire dans le fichier ouvert à l'aide de la méthode `write` de l'objet−fichier retourné par `open`.

❸ `file` est un synonyme de `open`. Ici, nous ouvrons le fichier, lisons son contenu et l'imprimons en une seule ligne.

❹ Nous savons que `test.log` existe (puisque nous venons juste d'écrire dedans), donc nous pouvons l'ouvrir pour ajouter des données (le paramètre `"a"` signifie ouverture pour ajout). En fait, nous pourrions le faire même si le fichier n'existait pas, puisque l'ouverture du fichier en mode ajout crée le fichier si nécéssaire. Mais le mode ajout n'endommagera *jamais* le contenu du fichier.

❺ Comme vous pouvez le voir, la ligne d'origine aussi bien que la nouvelle ligne ajoutée sont maintenant dans `test.log`. Notez également que le retour à la ligne n'est pas inclus. Puisque nous n'en avons pas explicitement écrit dans le fichier, le fichier n'en contient pas. Nous pouvons écrire un retour à la ligne avec le caractère `"\n"`. Puisque nous ne l'avons pas fait, tout ce que nous avons écrit finit sur la même ligne.

Pour en savoir plus sur l'utilisation de fichiers

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de la lecture et de l'écriture de fichiers, y compris comment lire un fichier ligne par ligne dans une liste (http://www.python.org/doc/current/tut/node9.html#SECTION009210000000000000000).
- eff−bot (http://www.effbot.org/guides/) traite de l'efficience et de la performance de différentes manières de lire un fichier (http://www.effbot.org/guides/readline−performance.htm).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions courantes à propose des fichiers (http://www.faqts.com/knowledge−base/index.phtml/fid/552).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes de l'objet−fichier (http://www.python.org/doc/current/lib/bltin−file−objects.html).

6.3. Itérations avec des boucles `for`

Comme la plupart des langages, Python a des boucles `for`. La seule raison pour laquelle vous ne les avez pas vues jusqu à maintenant est que Python sait faire tellement d autre choses que vous n en avez pas besoin aussi souvent.

La plupart des autres langages n ont pas de type de données liste aussi puissant que celui de Python, vous êtes donc amené à faire beaucoup de travail à la main, spécifier un début, une fin et un pas pour définir une suite d entiers ou de caractères ou d autres entités énumérables. Mais en Python une boucle `for` parcourt simplement une liste, de la même manière que les *list comprehensions* fonctionnent.

Exemple 6.8. Présentation des boucles `for`

```
>>> li = ['a', 'b', 'e']
>>> for s in li:          ❶
...     print s           ❷
a
b
e
>>> print "\n".join(li)   ❸
a
b
e
```

❶ La syntaxe d une boucle `for` est similaire aux *list comprehensions*. `li` est une liste et `s` prend successivement la valeur de chaque élément, en commençant par le premier.

❷ Comme une instruction `if` ou n importe quel autre bloc indenté, une boucle `for` peut contenir autant de lignes de codes que vous le voulez.

❸ Voici la raison pour laquelle vous n aviez pas encore vu la boucle `for` : nous n en avions pas eu besoin. C est incroyable la fréquence à laquelle nous utilisons les boucles `for` dans d autres langages alors que ce que nous voudrions vraiment et un `join` ou une *list comprehension*.

Faire un compteur "normal" (selon les critères de Visual Basic) pour la boucle `for` est également simple.

Exemple 6.9. Compteurs simples

```
>>> for i in range(5):                  ❶
...     print i
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(li)):            ❷
...     print li[i]
a
b
c
d
e
```

❶ Comme nous l avons vu dans l'Exemple 3.20, «Assignation de valeurs consécutives», `range` produit une liste d entiers que nous pouvons parcourir. Je sais que ça peut sembler étrange, mais c est parfois (et j insiste sur le *parfois*) utile d avoir une boucle sur un compteur.

❷ Ne faites jamais ça. C est un style de pensée Visual Basic. Libérez−vous en. Parcourez simplement la liste comme dans l exemple précédent.

Les boucles `for` ne sont seulement faites pour les compteurs simples. Elles peuvent parcourir de nombreuses choses. Voici un exemple d'utilisation d'une boucle `for` pour parcourir un dictionnaire.

Exemple 6.10. Parcourir un dictionnaire

```
>>> import os
>>> for k, v in os.environ.items():       ❶ ❷
...     print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
>>> print "\n".join(["%s=%s" % (k, v)
...     for k, v in os.environ.items()]) ❸
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
```

❶ `os.environ` est un dictionnaire des variables d environnement définies dans votre système. Sous Windows ce sont vos variables utilisateur et système. Sous UNIX ce sont les variables exportées par le script de démarrage de votre shell. Sous Mac OS il n y a pas de notion de variables d environnement, ce dictionnaire est donc vide.

❷ `os.environ.items()` retourne une liste de tuples : `[(`*`key1`*`, `*`value1`*`), (`*`key2`*`, `*`value2`*`), ...]`. La boucle `for` parcourt cette liste. A la première itération, il assigne *`key1`* à `k` et *`value1`* à `v`, donc `k` = `USERPROFILE` et `v` = `C:\Documents and Settings\mpilgrim`. A la seconde, `k` reçoit la deuxième clé, `OS` et `v` la valeur correspondante, `Windows_NT`.

❸ Avec l assignement multiple de variable et les *list comprehensions*, vous pouvez entièrement remplacer la boucle `for` par une seule instruction. Le choix d une des deux formes dans votre code est une question de style personnel. J aime ce style parce qu il rend clair que ce que nous faisons est une mutation d un dictionnaire en une liste, puis de joindre cette liste en une chaîne unique. D autres programmeurs préfèrent la forme de la boucle `for`. Notez que la sortie est la même dans les deux cas, bien que cette version−ci soit légèrement plus rapide car il n y a qu une instruction `print` au lieu d une par itération.

Maintenant nous pouvons examiner l'usage de la boucle `for` dans la classe `MP3FileInfo` du programme d'exemple `fileinfo.py` présenté au Chapitre 5.

Exemple 6.11. Boucle `for` dans `MP3FileInfo`

```
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}                          ❶
    .
    .
    .
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items(): ❷
                    self[tag] = parseFunc(tagdata[start:end])             ❸
```

❶ `tagDataMap` est un attribut de classe qui définit les balises que nous recherchons dans un fichier MP3 file. Les balises sont stockées dans des champ de longueur fixe, une fois que nous avons lu les derniers 128 octets du fichier, les octets 3 à 32 contiennent toujours le titre de la chanson, 33−62 le nom de l artiste, 63−92 le nom de l album etc. Notez que `tagDataMap` est un dictionnaire de tuples et que chaque tuple contient deux entiers et une référence de fonction.

❷ Ceci à l air compliqué, mais ne l est pas. La structure des variables de `for` correspond à la structure des éléments de la liste retournée par `items`. Rappelez−vous, `items` retourne une liste de tuples de la forme `(`*`key`*`, `*`value`*`)`. Le premier élément de cette liste est `("title", (3, 33, <function stripnulls>))`, donc à la première itération de la boucle `tag` reçoit `"title"`, `start` reçoit `3`, `end` reçoit `33` et `parseFunc` reçoit la fonction `stripnulls`.

❸ Maintenant que nous avons extrait tous les paramètres pour une balise MP3 unique, sauvegarder les données de la balise data est simple. Nous découpons `tagdata` de `start` à `end` pour obtenir les véritables données de cette balise, nous appelons `parseFunc` pour le traitement final des données et assignons le résultat comme valeur de la clé `tag` dans le pseudo−dictionnaire `self`. Après itération de tous les éléments de `tagDataMap`, `self` a les valeurs de toutes les balises et vous savez à quoi ça ressemble.

6.4. Utilisation de `sys.modules`

Les modules, comme tout le reste en Python, sont des objets. Une fois qu'il a été importé, vous pouvez toujours obtenir une référence à un module à travers le dictionnaire global `sys.modules`.

Exemple 6.12. Présentation de `sys.modules`

```
>>> import sys                          ❶
>>> print '\n'.join(sys.modules.keys()) ❷
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
```

❶ Le module `sys` contient des informations système, comme la version de Python que vous utilisez (`sys.version` ou `sys.version_info`) et des options système comme la profondeur maximale de récursion autorisée (`sys.getrecursionlimit()` et `sys.setrecursionlimit()`).

❷ `sys.modules` est un dictionnaire qui contient tous les modules qui ont été importés depuis que Python a été démarré. La clé est le nom de module, la valeur est l objet module. Notez que cela comprend plus que les modules que *votre* programme a importé. Python charge certains modules au démarrage et si vous êtes dans une IDE Python, `sys.modules` contient tous les modules importés par tous les programmes que vous avez exécutés dans l IDE.

Cet exemple présente l'utilisation de `sys.modules`.

Exemple 6.13. Utilisation de `sys.modules`

```
>>> import fileinfo          ❶
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

❶ Au fur et à mesure que des nouveaux modules sont importés, ils sont ajoutés à `sys.modules`. Cela explique pourquoi importer le même module deux fois est très rapide : Python a déjà chargé et mis en cache le module dans `sys.modules`, donc l importer une deuxième fois n est qu une simple consultation de dictionnaire.

❷ A partir du nom (sous forme de chaîne) de n importe quel module déjà importé, vous pouvez obtenir une référence au module lui−même du dictionnaire `sys.modules`.

L'exemple suivant montre l'utilisation de l'attribut de classe `__module__` avec le dictionnaire `sys.modules` pour obtenir une référence vers le module dans lequel la classe est définie.

Exemple 6.14. L attribut de classe `__module__`

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__                ❶
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__] ❷
<module 'fileinfo' from 'fileinfo.pyc'>
```

❶ Chaque classe Python a un attribut de classe `__module__` prédéfini, dont la valeur est le nom du module dans lequel la classe est définie.

❷ En combinant cela au dictionnaire `sys.modules` vous pouvez obtenir une référence au module dans lequel la classe est définie.

Maintenant vous pouvez comprendre l'utilisation de `sys.modules` dans `fileinfo.py`, le programme d'exemple présenté au Chapitre 5. Cet exemple montre cette partie du code.

Exemple 6.15. `sys.modules` dans `fileinfo.py`

```
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):       ❶
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]        ❷
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo ❸
```

❶ Ceci est une fonction avec deux arguments, `filename` est obligatoire, mais `module` est optionnel et est par défaut le module qui contient la classe `FileInfo`. Cela peut sembler peu efficace si on pense que Python évalue l expression `sys.modules` à chaque fois que la fonction est appelée. En fait, Python n évalue les expressions par défaut qu une fois, la première fois que le module est importé. Comme nous le verrons plus loin nous n appelons jamais cette fonction avec un argument `module`, `module` sert donc de constante au niveau de la fonction.

❷ Nous détaillerons cette ligne plus tard, après avoir étudié le module `os`. Pour l instant retenez simplement que `subclass` obtient le nom d un classe, comme `MP3FileInfo`.

❸ Vous connaissez déjà `getattr`, qui obtient une référence a un objet par son nom. `hasattr` est une fonction complémentaire qui vérifie si un objet possède un attribut particulier. Dans le cas présent, si un module possède une classe particulière (bien que cela fonctionne pour tout objet et tout attribut, tout comme `getattr`). En français, cette ligne de code dit "si le module a la classe nommée par `subclass` alors la retourner, sinon retourner la classe de base `FileInfo`".

Pour en savoir plus

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de quand et comment les arguments par défaut sont évalués (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `sys` (http://www.python.org/doc/current/lib/module−sys.html).

6.5. Travailler avec des répertoires

Le module `os.path` a de nombreuses fonctions pour manipuler les chemins de fichiers et de répertoires. Ici nous voulons gérer les chemins et lister le contenu d'un répertoire.

Exemple 6.16. Construction de noms de chemins

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3") ❶ ❷
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")   ❸
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")                         ❹
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python") ❺
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

❶ `os.path` est une référence à un module, quel module exactement dépend de la plateforme que vous utilisez. Tout comme `getpass` encapsule les différences entre plateforme en assignant à `getpass` une fonction spécifique à la plateforme, `os` encapsule les différences entre plateformes en assignant à `path` un module spécifique à la plateforme.

❷ La fonction `join` de `os.path` construit un nom de chemin à partir d un ou de plusieurs noms de chemins partiels. Dans ce cas simple il ne fait que concaténer des chaînes (notez que traiter des noms de chemins sous Windows est ennuyeux car le backslash force à utiliser le caractère d échappement).

❸ Dans cette exemple un peu moins simple, `join` ajoute un backslash supplémentaire au nom de chemin avant de le joindre au nom de fichier. J étais ravi quand j ai découvert cela car `addSlashIfNecessary` est une des petites fonctions stupides que je dois toujours écrire quand je construis ma boîte à outil dans un nouveau langage. *N écrivez pas* cette petite fonction stupide en Python, des gens intelligents l ont déjà fait pour vous.

❹ `expanduser` développe un nom de chemin qui utilise ~ pour représenter le répertoire de l utilisateur. Cela fonctionne sur toutes les plateformes où les utilisateurs ont un répertoire propre comme Windows, UNIX et Mac OS X, c'est sans effet sous Mac OS.

❺ En combinant ces techniques, vous pouvez facilement construire des noms de chemins pour les répertoires et les fichiers contenus dans le répertoire utilisateur.

Exemple 6.17. Division de noms de chemins

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")                 ❶
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3") ❷
>>> filepath                                                     ❸
'c:\\music\\ap'
>>> filename                                                     ❹
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)          ❺
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

❶ La fonction `split` divise un nom de chemin complet et retourne un tuple contenant le chemin et le nom de fichier. Vous vous rappelez quand je vous ai dit que vous pouviez utiliser l assignement multiple de variables pour retourner des valeurs multiples d une fonction ? Et bien `split` est une de ces fonctions.

❷

Nous assignons la valeur de retour de la fonction `split` à un tuple de deux variables. Chaque variable reçoit la valeur de l élément correspondant du tuple retourné.

❸ La première variable, `filepath`, reçoit la valeur du premier élément du tuple retourné par `split`, le chemin du fichier.

❹ La seconde variable, `filename`, reçoit la valeur du second élément du tuple retourné par `split`, le nom de fichier.

❺ `os.path` contient aussi une fonction `splitext`, qui divise un nom de fichier et retourne un tuple contenant le nom de fichier et l extension. Nous utilisons la même technique pour assigner chacun d entre eux à des variables séparées.

Exemple 6.18. Liste des fichiers d un répertoire

```
>>> os.listdir("c:\\music\\_singles\\")                ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)                               ❷
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin',
'docbook', 'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS',
'MSDOS.SYS', 'Music', 'NTDETECT.COM', 'ntldr', 'pagefile.sys',
'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname)
...      if os.path.isfile(os.path.join(dirname, f))] ❸
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname)
...      if os.path.isdir(os.path.join(dirname, f))]  ❹
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

❶ La fonction `listdir` prend un nom de chemin et retourne une liste du contenu du répertoire.

❷ `listdir` retourne à la fois les fichiers et les répertoires, sans indiquer lequel est quoi.

❸ Vous pouvez utiliser le filtrage de liste et la fonction `isfile` du module `os.path` pour séparer les fichiers des répertoires. `isfile` prend un nom de chemin et retourne 1 si le chemin représente un fichier et 0 dans le cas contraire. Ici, nous utilisons `os.path.join` pour nous assurer que nous avons un nom de chemin complet, mais `isfile` marche aussi avec des chemins partiels, relatifs au répertoire en cours. Vous pouvez utiliser `os.getcwd()` pour obtenir le répertoire en cours.

❹ `os.path` a aussi une fonction `isdir` qui retourne 1 si le chemin représente un répertoire et 0 dans le cas contraire. Vous pouvez l utiliser pour obtenir une liste des sous−répertoires d un répertoire.

Exemple 6.19. Liste des fichiers d un répertoire dans `fileinfo.py`

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]          ❶ ❷
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList] ❸ ❹ ❺
```

❶ `os.listdir(directory)` retourne une liste de tous les fichiers et répertoires de `directory`.

❷

En parcourant la liste avec `f`, nous utilisons `os.path.normcase(f)` pour normaliser la casse en fonction des paramètres par défaut du système d exploitation. `normcase` est une petite fonction utile qui compense le problème des systèmes d exploitation insensibles à la casse qui pensent que `mahadeva.mp3` et `mahadeva.MP3` sont le même fichier. Par exemple, sous Windows et Mac OS, `normcase` convertit l ensemble du nom de fichier en minuscules, sous les systèmes compatibles UNIX, elle retourne le nom de fichier inchangé.

❸ En parcourant la liste normalisée avec `f` à nouveau, nous utilisons `os.path.splitext(f)` pour diviser chaque nom de fichier en nom et extension.

❹ Pour chaque fichier, nous regardons si l extension est dans la liste d extensions de fichier qui nous intéressent (`fileExtList`, qui a été passé à la fonction `listDirectory`).

❺ Pour chaque fichier qui nous intéresse, nous utilisons `os.path.join(directory, f)` pour construire le chemin de fichier complet. Nous retournons une liste de noms de chemin complets.

A chaque fois que c est possible, vous devriez utiliser les fonction de `os` et `os.path` pour les manipulations de fichier, de répertoire et de chemin. Ces modules enveloppent des modules spécifiques aux plateformes, les fonctions comme `os.path.split` marchent donc sous UNIX, Windows, Mac OS et toute autre plateforme supportée par Python.

Il existe une autre manière d'obtenir le contenu d'un répertoire. Elle est très puissante et utilise le type de jokers qui vous sont familier si vous utilisez la ligne de commande.

Exemple 6.20. Liste du contenu d'un répertoire avec `glob`

```
>>> os.listdir("c:\\music\\_singles\\")                    ❶
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3',
'kairo.mp3', 'long_way_home1.mp3', 'sidewinder.mp3',
'spinning.mp3']
>>> import glob
>>> glob.glob('c:\\music\\_singles\\*.mp3')                ❷
['c:\\music\\_singles\\a_time_long_forgotten_con.mp3',
'c:\\music\\_singles\\hellraiser.mp3',
'c:\\music\\_singles\\kairo.mp3',
'c:\\music\\_singles\\long_way_home1.mp3',
'c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\_singles\\s*.mp3')               ❸
['c:\\music\\_singles\\sidewinder.mp3',
'c:\\music\\_singles\\spinning.mp3']
>>> glob.glob('c:\\music\\*\\*.mp3')                       ❹
```

❶ Comme nous l'avons vu plus haut, `os.listdir` prend simplement un chemin de répertoire et retourne la liste de tous les fichiers et répertoires qu'il contient.

❷ Le module `glob`, par contre, prend un joker et retourne le chemin complet de tous les fichiers et répertoires qui lui correspondent. Ici, le joker est un chemin de répertoire plus "*.mp3", c'est à dire tous les fichiers `.mp3`. Notez que chaque élément de la liste retournée contient le chemin complet du fichier.

❸ Voici le joker pour trouver tous les fichiers d'un répertoire qui commencent par "s" et finissent par ".mp3".

❹ Maintenant considerez le scénario suivant : vous avez un répertoire `music`, contenant plusieurs sous−répertoires, avec des fichiers `.mp3` dans chaque sous−répertoire. Vous pouvez obtenir une liste de tous ces fichiers avec un seul appel à `glob`, en utilisant deux jokers à la fois. Un des jokers est `"*.mp3"` (qui correspond aux fichiers `.mp3`) et l'autre est *à l'intérieur du chemin de répertoire*, ce qui correspond aux sous−répertoires de `c:\music`. C'est une énorme puissance contenue dans une fonction à l'air faussement simple !

Pour en savoir plus sur le module `os`

- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions sur le module `os` (http://www.faqts.com/knowledge−base/index.phtml/fid/240).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `os` (http://www.python.org/doc/current/lib/module−os.html) et le module `os.path` (http://www.python.org/doc/current/lib/module−os.path.html).

6.6. Assembler les pièces

A nouveau, tous les dominos sont en place. Nous avons vu comment chaque ligne de code fonctionne. Maintenant prenons un peut de recul pour voir comment tout cela s assemble.

Exemple 6.21. `listDirectory`

```
def listDirectory(directory, fileExtList):                                  ❶
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList]                   ❷
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]): ❸
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:] ❹
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo ❺
    return [getFileInfoClass(f)(f) for f in fileList]                       ❻
```

❶ `listDirectory` est l attraction principale de ce module. Elle prend un répertoire (`c:\music\_singles\` dans mon cas) et une liste d extensions intéressantes (comme `['.mp3']`) et elle retourne une liste d instances de classe qui se comportent comme des dictionnaires et qui contiennent des métadonnées concernant chaque fichier intéressant de ce répertoire. Et elle le fait en une poignée de ligne simples et directes.

❷ Comme nous l avons vu dans la section précédente, cette ligne de code permet d obtenir une liste de noms de chemin complets de tous les fichiers de `directory` qui ont une extension de fichier intéressante (comme spécifiée par `fileExtList`).

❸ Les programmeurs Pascal à l ancienne les connaissent bien, mais la plupart des gens me jettent un regard vide quand je leur dit que Python supporte les *fonctions imbriquées* −− littéralement une fonction à l intérieur d une fonction. La fonction imbriquée `getFileInfoClass` peut seulement être appelée de la fonction dans laquelle elle est définie, `listDirectory`. Comme pour toute autre fonction, vous n avez pas besoin d une déclaration d interface ou de quoi que ce soit d autre, définissez juste la fonction et écrivez−la.

❹ Maintenant que vous avez vu le module `os`, cette ligne devrait être plus compréhensible. Elle obtient l extension du fichier (`os.path.splitext(filename)[1]`), la force en majuscules (`.upper()`), découpe le point (`[1:]`) et construit un nom de classe en formatant la chaîne. Donc, `c:\music\ap\mahadeva.mp3` devient `.mp3`, puis `.MP3`, puis `MP3` et enfin `MP3FileInfo`.

❺ Ayant construit le nom de la classe qui doit manipuler ce fichier, nous vérifions si cette classe existe dans ce module. Si c est le cas, nous retournons la classe, sinon, nous retournons la classe de base, `FileInfo`. C est un point très important : *cette fonction retourne une classe*. Pas une instance de classe, mais la classe elle−même.

❻ Pour chaque fichier dans notre liste de "fichiers intéressants" (`fileList`), nous appelons `getFileInfoClass` avec le nom de fichier (`f`). Appeler `getFileInfoClass(f)` retourne une classe, nous ne savons pas exactement laquelle mais cela ne nous intéresse pas. Nous créons alors une instance de cette

classe (quelle qu elle soit) et passons le nom du fichier (encore `f`), à la méthode `__init__`. Comme nous l avons vu auparavant dans ce chapitre, la méthode `__init__` de `FileInfo` définit `self["name"]`, ce qui déclenche `__setitem__`, qui est redéfini dans la classe descendante (`MP3FileInfo`) comme une fonction traitant le fichier de manière à en extraire les métadonnées. Nous faisons cela pour tous les fichiers intéressants et retournons une liste des instances ainsi créées.

Notez que `listDirectory` est complètement générique. Il ne sait pas à l avance quels types de fichiers iI va obtenir, ou quelles sont les classes qui pourraient triter ces fichiers. Il inspecte le répertoire à la recherche de fichiers à traiter, puis recourt à l introspection sur son propre module pour voir quelles classes de traitement (comme `MP3FileInfo`) sont définies. Vous pouvez étendre ce programme pour gérer d autres types de fichiers simplement en définissant une classe portant un nom approprié : `HTMLFileInfo` pour les fichiers HTML, `DOCFileInfo` pour les fichiers `.doc` de Word, etc. `listDirectory` les prendra tous en charge, sans modification, en se déchargeant du traitement proprement dit sur les classes appropriées et en assemblant les résultats.

6.7. Résumé

Le programme `fileinfo.py`, introduit au Chapitre 5; devrait maintenant être parfaitement clair.

```
"""Framework for getting filetype-specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
```

```
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

Avant de plonger dans le chapitre suivant, assurez vous que vous vous sentez à l aise pour :

- Intercepter les exceptions avec `try...except`
- Protéger les ressources externes avec `try...finally`
- Lire dans des fichiers
- Assigner des valeurs multiples en une fois dans une boucle `for`
- Utiliser le module `os` pour tous vos besoins de manipulation de fichiers indépendamment de la plateforme
- Instancier des classes de type inconnu dynamiquement en traitant les classes comme des objets

Chapitre 7. Expressions régulières

Les expressions régulières sont un moyen puissant et standardisé de rechercher, remplacer et analyser du texte à l aide de motifs complexes de caractères. Si vous avez utilisé les expressions régulières dans d autres langages (comme Perl), vous pouvez sauter cette section et lire uniquement la présentation du module `re` (http://www.python.org/doc/current/lib/module−re.html) pour avoir une vue d ensemble des fonctions disponibles et de leurs arguments.

7.1. Plonger

Les objets−chaîne Strings ont des méthodes pour rechercher (`index`, `find` et `count`), remplacer (`replace`) et analyser (`split`) mais elles sont limitées aux cas les plus simples. Les méthodes de recherche tentent de trouver une chaîne unique et prédéfinie et elles sont toujours sensibles à la casse. Pour faire une recherche non sensible à la casse sur une chaîne `s`, vous devez appeler `s.lower()` ou `s.upper()` et vous assurer que vos chaînes de recherche sont dans la casse correspondante. Les méthodes `replace` et `split` ont les mêmes restrictions.

Si ce que vous essayez de faire peut être accompli avec les fonctions de chaînes, utilisez−les. Elles sont rapides et faciles à comprendre et il y a beaucoup d'avantages à un code rapide, simple et lisible. Mais si vous vous rendez compte que vous utilisez un grand nombre de fonctions de chaînes différentes avec des instruction `if` pour les cas particulier, ou si vous les associez à des fonctions `split` et `join` et à des *list comprehension* de manière complexe et illisible, vous devez vous tourner vers les expressions régulières.

Bien que la syntaxe des expressions régulières soit compacte et différente du code ordinaire, le résultat peut être *plus* lisible qu une solution à la main avec une longue séquence de fonctions de chaînes. Il y a même une manière d inclure des commentaires dans les expressions régulières pour les documenter.

7.2. Exemple : adresses postales

Cette série d exemples est inspirée d un problème réel que j ai eu au cours de mon travail, l extraction et la standardisation d adresses postales exportées d un ancien système avant de les importer dans un nouveau système (vous voyez, je n invente rien, c est réellement utile). L exemple suivant montre comment j ai abordé ce problème.

Exemple 7.1. Reconnaître la fin d une chaîne

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')                ❶
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')                ❷
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')  ❸
'100 NORTH BROAD RD.'
>>> import re                               ❹
>>> re.sub('ROAD$', 'RD.', s)               ❺ ❻
'100 NORTH BROAD RD.'
```

❶ Mon but était de standardiser les adresses de manière à ce que `'ROAD'` soit toujours abrégé en `'RD.'`. Au premier abord, je pensais que ce serait assez simple pour utiliser uniquement la méthode de chaîne `replace`. Après tout, toutes les données étaient déjà en majuscules, donc les erreurs de casses ne seraient pas un problème. De plus, la chaîne de recherche, `'ROAD'`, était une constante. Pour cet exemple trompeusement simple, `s.replace` fonctionne effectivement.

❷ Malheureusement, la vie est pleine de contre−exemples et je découvrais assez rapidemment celui−ci. Le problème ici est que `'ROAD'` apparaît deux fois dans l adresse, d abord comme partie du nom de la rue `'BROAD'` et ensuite comme mot isolé. La méthode `replace` trouve ces deux occurences et les remplace aveuglément, rendant l adresse illisible.

❸ Pour résoudre le problème des adresses comprenant plus d une sous−chaîne `'ROAD'`, nous pourrions recourir à quelque chose de ce genre : ne rechercher et remplacer `'ROAD'` que dans les 4 derniers caractères de l adresse (`s[−4:]`) et ignorer le début de la chaîne (`s[:−4]`). Mais on voit bien que ça commence à être embrouillé. Par exemple, le motif dépend de la longueur de la chaîne que nous remplaçons (si nous remplaçons `'STREET'` par `'ST.'`, nous devons écrire `s[:−6]` et `s[−6:].replace(...)`). Aimeriez−vous revenir à ce code dans six mois et devoir le débugger ? En ce qui me concerne, certainement pas.

❹ Il est temps de recourir aux expressions régulières. En Python, toutes les fonctionalités en rapport aux expressions régulières sont contenues dans le module `re`.

❺ Regardez le premier paramètre, `'ROAD$'`. C est une expression régulière très simple qui ne reconnaît `'ROAD'` que s il apparaît à la fin d une chaîne. Le symbole `$` signifie "fin de la chaîne" (il y a un caractère correspondant, l accent circonflexe `^`, qui signifie "début de la chaîne").

❻ A l aide de la fonction `re.sub`, nous recherchons dans la chaîne `s` l expression régulière `'ROAD$'` et la remplaçons par `'RD.'`. Cela correspond à `ROAD` à la fin de la chaîne `s`, mais ne correspond *pas* au `ROAD` faisant partie du mot `BROAD`, puisqu il est au milieu de `s`.

En continuant mon travail de reformatage d adresses, je decouvrais bientôt que le modèle précédent, reconnaître `'ROAD'` à la fin de l adresse, ne suffisait pas, car toutes les adresses n incluaient pas d identifiant pour la rue. Certaines finissaient simplement par le nom de la rue. La plupart du temps, je m en sortais sans problème, mais si le nom de la rue était `'BROAD'`, alors l expression régulière reconnaissait `'ROAD'` à la fin de la chaîne dans le mot `'BROAD'`. Ce n était pas ce que je voulais.

Exemple 7.2. Reconnaître des mots entiers

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)  ❶
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)  ❷
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)  ❸
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ❹
'100 BROAD RD. APT 3'
```

❶ Ce que je voulais *vraiment* était de reconnaître `'ROAD'` quand il était à la fin de la chaîne *et* qu il était un mot isolé, pas une partie de mot. Pour exprimer cela dans une expressions régulière, on utilise `\b`, qui signifie "une limite de mot doit apparaître ici". En Python, c'est rendu plus compliqué par le fait que le caractère `'\'`, qui est le caractère d échappement, doit lui−même être précédé du caractère d échappement (c'est ce qui est parfois appelé la *backslash plague* et c est une des raison pour lesquelles les expressions régulières sont plus faciles à utliser en Perl qu en Python. Par contre, Perl mélange les expressions régulières et la syntaxe du langage, donc si vous avez un bogue, il peut être difficile de savoir si c est une erreur dans la syntaxe ou dans l expression régulière).

❷ Pour éviter la *backslash plague*, vous pouvez utiliser ce qu on appelle une chaîne brute, en préfixant la chaîne par la lettre `r`. Cela signale à Python que cette chaîne doit être traitée sans échappement, `'\t'` est un caractère de tabulation, mais `r'\t'` est réellement un caractère *backslash* `\` suivi de la lettre `t`. Je vous conseille de toujours utiliser des chaînes brutes lorsque vous employez des expressions régulières, sinon cela devient confus très vite (et les expressions régulières peuvent devenir suffisament confuses par elles−mêmes).

❸ **soupir** Malheureusement, je découvrais rapidement d autres cas qui contredisaient mon raisonnement. Dans le cas présent, l adresse contenait le mot isolé `'ROAD'` mais il n était pas à la fin de la chaîne, car l adresse avait un numéro d appartement après l identifiant de la rue. Comme `'ROAD'` n était pas tout à la fin de la chaîne, il n était pas identifié, donc l appel de `re.sub` s achèvait sans rien remplacer, j obtenais en retour la chaîne d origine, ce qui n était pas le but recherché.

❹ Pour résoudre ce problème, j enlevais le caractère `$`et ajoutais un deuxième `\b`. L expression régulière signifiait alors "reconnaître `'ROAD'` lorsqu il est un mot isolé, n importe où dans la chaîne", que ce soit à la fin, au début ou quelque part au milieu.

7.3. Exemple : chiffres romains

Vous avez certainement déjà vu des chiffres romains, par exemple dans Astérix[2]

En chiffres romains, il y a sept caractères qui sont répétés et combinés de différentes manières pour représenter des nombres.

- `I` = `1`
- `V` = `5`
- `X` = `10`
- `L` = `50`
- `C` = `100`
- `D` = `500`
- `M` = `1000`

Voici les règles générales pour construire des nombres romains :

- Les caractères sont additifs. `I` est `1`, `II` est `2` et `III` est `3`. `VI` est `6` (littéralement "`5` et `1`"), `VII` est `7` et `VIII` est `8`.
- Les caractères en un (`I`, `X`, `C`, and `M`) peuvent être répétés jusqu à trois fois. A `4`, vous devez soustraire du prochain caractère en cinq. Vous ne pouvez pas représenter `4` par `IIII`, au lieu de ça il est représenté par `IV` ("`1` de moins que `5`"). `40` s écrit `XL` ("`10` de moins que `50`"), `41` s écrit `XLI`, `42` `XLII`, `43` `XLIII` et `44` `XLIV` ("`10` de moins que `50`, puis `1` de moins que `5`").
- De manière similaire, à `9`, vous devez soustraire du prochain caractère en un : `8` est `VIII` mais `9` est `IX` ("`1` de moins que `10`"), pas `VIIII` (puisque le caractère `I` ne peut être répété quatre fois). `90` est `XC` et `900` `CM`.
- Les caractères en cinq ne peuvent être répétés. `10` est toujours représenté par `X`, jamais par `VV`. `100` est toujours `C`, jamais `LL`.

7.3.1. Rechercher les milliers

Qu est−ce qui serait nécessaire pour vérifier qu une chaîne de caractères quelconque constitue des chiffres romains valides ? Nous allons opérer caractère par caractère. Puisque les chiffres romains sont toujours écrits du plus grand vers le plus petit, nous allons commencer par les plus grands : les milliers. Pour les nombres supérieurs à 1000, les milliers sont représentés par une série de caractères `M`.

Exemple 7.3. Rechercher les milliers

```
>>> import re
>>> pattern = '^M?M?M?$'          ❶
>>> re.search(pattern, 'M')       ❷
<SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')      ❸
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')     ❹
```

```
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM') ❺
>>> re.search(pattern, '')     ❻
<SRE_Match object at 0106F4A8>
```

❶ Ce motif a trois parties :

- `^` − reconnaît ce qui suit uniquement en début de chaîne. Si ce n était pas spécfié, le motif reconnaîtrait les `M` où qu ils soient, ce qui n est pas ce que nous voulons. Nous voulons être sûrs que les `M`, s il y en a dans la chaîne, sont à son début.
- `M?` − reconnaît un `M` optionnel. Comme nous le répétons trois fois, nous reconnaissons 0 à 3 `M` se suivant.
- `$` − reconnaît ce qui précède uniquement à la fin de la chaîne. Lorsqu il est combiné avec `^` en début de motif, cela signifie que le motif doit correspondre à la chaîne entière, sans autres caractères avant ou après les `M`.

❷ L essence du module `re` est la fonction `search`, qui prend une expression régulière (`pattern`) et une chaîne (`'M'`) qu elle va tenter de faire correspondre. Si une correspondance est trouvée, `search` retourne un objet ayant diverses méthodes permettant de décrire la correspondance, sinon, `search` retourne `None`, la valeur nulle de Python. Tout ce qui nous intéresse à ce stade est de savoir si le motif est reconnu, ce que nous pouvons dire rien qu en regardant la valeur retournée par `search`. `'M'` est reconnu par cette expression régulière car le premier `M` optionnel correspond et que le second et troisième `M` sont ignorés.

❸ `'MM'` est reconnu puisque les premier et deuxième `M` optionnels correspondent et que le troisième est ignoré.

❹ `'MMM'` est reconnu puisque les trois `M` correspondent.

❺ `'MMMM'` n est pas reconnu. Les trois `M` correspondent, mais l expression régulière précise la fin de chaîne (par le caractère `$`) et la chaîne ne s arrête pas là (à cause du quatrième `M`). Donc `search` retourne `None`.

❻ Un élément intéressant est qu une chaîne vide est reconnue par l expression régulière, puisque tous les `M` sont optionnels.

7.3.2. Rechercher les centaines

Les centaines présentent plus de difficultés que les milliers car elles peuvent être exprimées de plusieurs manières mutuellement exclusives, en fonction de leur valeur.

- `100` = `C`
- `200` = `CC`
- `300` = `CCC`
- `400` = `CD`
- `500` = `D`
- `600` = `DC`
- `700` = `DCC`
- `800` = `DCCC`
- `900` = `CM`

Il y a donc quatre motifs possibles :

- `CM`
- `CD`
- 0 à 3 `C` (0 si les centaines valent 0)
- `D`, suivi de 0 à 3 `C`

Les deux derniers motifs peuvent être combinés en :

- un `D` optionnel, suivi de 0 à 3 `C`

L exemple suivant montre comment valider les centaines en chiffres romains.

Exemple 7.4. Rechercher les centaines

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ❶
>>> re.search(pattern, 'MCM')             ❷
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')              ❸
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')          ❹
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')            ❺
>>> re.search(pattern, '')                ❻
<SRE_Match object at 01071D98>
```

❶ Ce motif commence de la même manière que le précédent, en vérifiant le début de chaîne (^), puis les milliers (`M?M?M?`). Ensuite vient la nouvelle partie, entre parenthèses, qui définit un ensemble de trois motifs mutuellement exclusifs séparés par des barres verticales : `CM`, `CD`, and `D?C?C?C?` (qui est un `D` optionnel suivi de 0 à 3 `C` optionnels). Le processeur d expressions régulières teste chacun de ces motifs dans l ordre (de gauche à droite), prend le premier qui correspond et ignore le reste.

❷ `'MCM'` est reconnu car le premier `M` correspond, que le second et troisième `M` sont ignorés et que `CM` correspond (et donc les motifs `CD` et `D?C?C?C?` ne sont même pas examinés). `MCM` est la représentation de `1900`.

❸ `'MD'` est reconnu car le premier `M` correspond, les deuxième et troisième `M` sont ignorés et que le motif `D?C?C?C?` reconnaît `D` (chacun des trois `C` est optionnel et est ignoré). `MD` est la représentation de `1500`.

❹ `'MMMCCC'` est reconnu car les trois `M` correspondent et que le motif `D?C?C?C?` reconnaît `CCC` (le `D` est optionnel et est ignoré). `MMMCCC` est la représentation de `3300`.

❺ `'MCMC'` n est pas reconnu. Le premier `M` correspond, les deuxième et troisième `M` sont ignorés et le `CM` correspond, mais le `$` ne correspond pas car nous ne sommes pas encore à la fin de la chaîne (il nous reste le caractère `C` à évaluer). Le `C` ne correspond *pas* comme partie du motif `D?C?C?C?` car le motif `CM` a déja été reconnu et qu ils sont mutuellement exclusifs.

❻ Fait intéressant, une chaîne vide est toujours reconnue par ce motif, car tous les `M` sont optionnels et sont ignorés et que la chaîne vide est reconnue par le motif `D?C?C?C?` dans lequel tous les caractères sont optionnels et sont ignorés.

Ouf ! Vous voyez à quel point les expressions régulières peuvent devenir compliquées ? Et nous n avons vu que les milliers et les centaines. Heureusement, si vous avez suivi jusque là, les dizaines sont relativement simples puisqu elles suivent exactement le même motif. Mais continuons en examinant une autre manière d exprimer ce motif.

7.4. Utilisation de la syntaxe `{n,m}`

Dans la section précédente, nous avons vu un motif dans lequel le même caractère pouvait être répété jusqu à trois fois. Il y a une autre manière d exprimer cela dans les expressions régulière, que certaines personnes trouvent plus lisible. D abord, revenons sur la méthode que nous avons utilisé dans l exemple précédent.

Exemple 7.5. L ancienne méthode : chaque caractère est optionnel

```
>>> import re
>>> pattern = '^M?M?M?$'
```

```
>>> re.search(pattern, 'M')    ❶
<_sre.SRE_Match object at 0x008EE090>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM')   ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM')  ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMMM') ❹
>>>
```

❶ Cette chaîne est reconnue : le motif reconnaît le début de la chaîne, puis le premier `M` optionnel, mais pas de second ni de troisième `M` (ce qui est correct puisqu'ils sont optionnels), puis la fin de la chaîne.

❷ Le motif reconnaît le début de la chaîne, puis le premier et le second `M` optionnels, mais pas de troisième `M` (ce qui est correct puisqu'il est optionnel), puis la fin de la chaîne.

❸ Le motif reconnaît le début de la chaîne, puis les trois `M` optionnels, puis la fin de la chaîne.

❹ Le motif reconnaît le début de la chaîne, puis les trois `M` optionnels, mais pas la fin de la chaîne (puisqu'il reste un `M`), la chaîne n'est donc pas reconnue et `None` est retourné.

Exemple 7.6. La nouvelle méthode : de `n` à `m`

```
>>> pattern = '^M{0,3}$'       ❶
>>> re.search(pattern, 'M')    ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MM')   ❸
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM')  ❹
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ❺
>>>
```

❶ Ce motifie signifie : "reconnaître le début de la chaîne, puis de zéro à trois caractères `M`, puis la fin de la chaîne." Le 0 et le 3 peuvent être n importe quel nombre, si nous voulons reconnaître au moins un, mais pas plus de trois caractères `M`, nous pouvons écrire `M{1,3}`.

❷ Le motif reconnaît le début de la chaîne, puis un `M` sur trois possibles, puis la fin de la chaîne.

❸ Le motif reconnaît le début de la chaîne, puis deux `M` sur trois possibles, puis la fin de la chaîne.

❹ Le motif reconnaît le début de la chaîne, puis trois `M` sur trois possibles, puis la fin de la chaîne.

❺ Le motif reconnaît le début de la chaîne, puis trois `M` sur trois possibles, puis *ne reconnaît pas* la fin de la chaîne. L expression régulière permet jusqu'à trois caractères `M` avant la fin de la chaîne, mais il y en a quatre, donc la chaîne n'est pas reconnue et `None` est retourné.

Il n y a aucun moyen de déterminer par un programme que deux expressions régulières sont équivalentes. Le mieux que vous puissiez faire est d écrire de nombreux cas de test pour vérifier que leur comportements sont identiques pour les entrées pertinentes. Nous discuterons plus en détail l écriture de cas de tests plus loin dans le livre.

7.4.1. Rechercher les dizaines et les unités

Maintenant, nous allons étendre l expression régulière pour prendre en compte les dizaines et les unités. L exemple suivant montre la recherche des dizaines.

Exemple 7.7. Rechercher les dizaines

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL')    ❶
```

```
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML')     ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX')    ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXX')  ❹
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLXXXX') ❺
>>>
```

❶ Le motif reconnaît le début de la chaîne, le premier `M` optionnel, puis `CM`, puis `XL`, puis la fin de la chaîne. Rappelez−vous que la syntaxe `(A|B|C)` signifie "reconnaître un seul parmi A, B et C". `XL` est reconnu, donc `XC` et `L?X?X?X?` sont ignorés, puis la find de la chaîne est reconnue. `MCML` est la représentation en chiffres romains de `1940`.

❷ Le motif reconnaît le début de la chaîne, le premier `M` optionnel, puis `CM`, puis `L?X?X?X?`. Pour `L?X?X?X?`, il reconnaît `L` et saute les trois caractères `X` optionnels. Il reconnaît ensuite la fin de la chaîne. `MCML` est la représentation en chiffres romains de `1950`.

❸ Le motif reconnaît le début de la chaîne, le premier `M` optionnel, puis `CM`, puis le `L` optionnel et le premier `X` optionnel, saute les trois caractères `X` optionnels, puis reconnaît la fin de la chaîne. `MCMLX` est la représentation en chiffres romains de `1960`.

❹ Le motif reconnaît le début de la chaîne, le premier `M` optionnel, puis `CM`, puis le `L` optionnel et les trois caractères `X`, puis la fin de la chaîne. `MCMLXXX` est la représentation en chiffres romains de `1980`.

❺ Le motif reconnaît le début de la chaîne, le premier `M` optionnel, puis `CM`, puis le `L` optionnel et les trois caractères `X`, puis *ne reconnaît pas* la fin de la chaîne puisqu'il y a encore un `X` non pris en charge. Donc la chaîne n'est pas reconnue et `None` est retourné. `MCMLXXXX` n'est pas un représentation en chiffres romains valide.

L expression pour les unités suit le même motif. Je vous épargne les détails et ne vous montre que le résultat final.

```
>>> pattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

A quoi est−ce que ça resemble en utilisant la syntaxe alternative avec `{n,m}` ? L exemple suivant montre la nouvelle syntaxe.

Exemple 7.8. Validation des chiffres romains avec `{n,m}`

```
>>> pattern = '^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
>>> re.search(pattern, 'MDLV')            ❶
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMDCLXVI')        ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII') ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'I')               ❹
<_sre.SRE_Match object at 0x008EEB48>
```

❶ Le motif reconnaît le début de la chaîne, puis un sur un maximum de quatre caractères `M`, puis `D?C{0,3}`. Pour cette expression il reconnaît le `D` optionnel et zéro sur un maximum de trois caractères `C`. Ensuite il reconnaît `L?X{0,3}` avec le `L` optionnel et zéro sur un maximum de trois caractères `X`. Puis il reconnaît `V?I{0,3}` avec le `V` optionnel et zéro sur un maximum de trois caractères `I`, puis la fin de la chaîne. `MDLV` est la représentation en chiffres romains de `1555`.

❷ Le motif reconnaît le début de la chaîne, puis deux sur un maximum de quatre caractères `M`, puis `D?C{0,3}` avec un `D` et un sur un maximum de trois caractères `C`. Ensuite il reconnaît `L?X{0,3}` avec un `L` et un sur un maximum de trois caractères `X`. Puis il reconnaît `V?I{0,3}` avec un `V` and et un sur un maximum de trois

caractères `I`, puis la fin de la chaîne. `MMDCLXVI` est la représentation en chiffres romains de `2666`.

❸ Le motif reconnaît le début de la chaîne, puis quatre sur un maximum de quatre caractères `M`, puis `D?C{0,3}` avec un `D` et trois sur un maximum de trois caractères `C`. Ensuite il reconnaît `L?X{0,3}` avec un `L` et trois sur un maximum de trois caractères `X`. Puis il reconnaît `V?I{0,3}` avec un `V` et trois sur un maximum de trois caractères `I`, puis la fin de la chaîne. `MMMMDCCCLXXXVIII` est la représentation en chiffres romains de `3888` et c'est le chiffre romain le plus long que vous pouvez écrire sans syntaxe étendue.

❹ Regardez bien. Le motif reconnaît le début de la chaîne, puis zéro sur un maximum de quatre `M`, puis `D?C{0,3}` en sautant le `D` optionnel et zéro sur un maximum de trois `C`. Ensuite il reconnaît `L?X{0,3}` en sautant le `L` optionnel et zéro sur un maximum de trois `X`. Puis il reconnaît `V?I{0,3}` en sautant le `V` optionnel et un sur un maximum de trois `I`, puis la fin de la chaîne.

Si vous avez suivi tout cela et que vous l'avez compris du premier coup, vous vous en sortez mieux que moi au début. Maintenant, imaginez devoir comprendre une expression régulière écrite par quelqu'un d'autre au mileu d'un fonction critique d'un programme de grande taille. Ou imaginez simplement de devoir revenir sur une de vos propres expressions régulières quelques mois plus tard. Je l'ai fait et ce n'est pas une partie de plaisir.

Dans la prochaine section vous explorerez une syntaxe alternative qui rendra possible la maintenance de vos expressions régulières.

7.5. Expressions régulières détaillées

Jusqu'à maintenant, vous n'avez vu que ce que j'appellerais des expressions régulières "compactes". Comme vous l'avez vu, elles sont difficiles à lire et même si vous comprenez ce qu'une d'entre elles fait, rien n'assure que vous pourrez la comprendre dans six mois. Ce qu'il faut, c'est une documentation intégrée.

Python fournit pour cela les *expressions régulières détaillées* (*verbose regular expressions*). Une expression régulière détaillée diffère d'une expression régulière compacte de deux manières :

- Les espaces sont ignorés. Les espaces, tabulations et retours chariot ne sont pas reconnus comme espaces, tabulations et retours chariot. Il ne sont pas reconnus du tout (si vous voulez reconnaître un espace dans une expression régulière détaillée, vous devez le faire précéder d'un caractère d'échapement "\").
- Les commentaires sont ignorés. Un commentaire dans une expression régulière détaillée est comme un commentaire dans du code Python : il commence par un caractère # et se poursuit jusqu'à la fin de la ligne. Dans le cas présent, c'est un commentaire à l'intérieur d'une chaîne de caractères multi−lignes plutôt que dans du code source, mais cela fonctionne de la même manière.

Cela sera plus clair avec un exemple. Revenons à l'expression régulière compacte avec laquelle nous avons travaillé et transformons−la en une expression régulière détaillée. L'exemple suivant montre comment.

Exemple 7.9. Expressions régulières intégrant des commentaires

```
>>> pattern = """
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    """
>>> re.search(pattern, 'M', re.VERBOSE)                ❶
<_sre.SRE_Match object at 0x008EEB48>
```

```
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE)          ❷
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MMMMDCCCLXXXVIII', re.VERBOSE) ❸
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M')                             ❹
```

❶ La chose la plus importante à se rappeler lorsqu'on utilise des expressions régulières détaillées est qu'il faut passer un argument supplémentaire : `re.VERBOSE` est une constante définie dans le module `re` qui signale que le motif doit être traité comme une expresion régulière détaillée. Comme vous le voyez, ce motif comprend beaucoup d'espaces (qui sont tous ignorés) et plusieurs commentaires (qui sont tous ignorés). Une fois enlevés les espaces et les commentaires, on obtient exactement la même expression régulière que nous avons vu à la section précédente, mais elle est beaucoup plus lisible.

❷ Le motif reconnaît le début de la chaîne, puis un sur un maximum de quatre `M`, puis `CM`, puis `L` et trois sur un maximum de trois `X`. Ensuite `IX` et la fin de la chaîne.

❸ Le motif reconnaît le début de la chaîne, puis quatre sur un maximum de quatre `M`, puis `D` et trois sur un maximum de trois `C`. Ensuite `L` et trois sur un maximum de trois `X`, puis `V` et trois sur un maximum de trois `I` et la fin de la chaîne.

❹ Rien n'est reconnu. Pourquoi ? Parce que le drapeau `re.VERBOSE` n'est pas mis et donc la fonction `re.search` traite le motif comme une expression régulière compacte, dans laquelle les espaces et les commentaires sont pris en compte. Python ne peut pas savoir si une expression régulière est détaillée ou non. Python considère que chaque expression régulière est compacte, à moins que vous ne spécifiez qu'elle est détaillée.

7.6. Etude de cas : reconnaissance de numéros de téléphone

Jusqu'ici nous nous sommes concentrés sur la reconnaissance de motifs complets, le motif est reconnu ou non. Mais les expressions régulières sont beaucoup plus puissantes que cela. Lorsqu'une expression régulière reconnaît un motif, nous pouvons sélectionner certaines parties du motif. Nous pouvons savoir ce qui a été reconnu et à quel endroit.

Cet exemple est tiré d'un autre problème réel que j'ai eu au cours de mon travail précédent. Le problème : la reconnaissance de numéros de téléphone au Etats−Unis. Le client voulait que la saisie se fasse librement (dans un champ unique), mais voulait stocker le code régional, l'indicatif, le numéro et une extension optionnelle séparément dans la base de données. Je parcourais le Web et trouvais de nombreux exemples d'expressions régulières qui avaient pour but de faire cela, mais aucune n'était assez souple.

Voici les numéros de téléphone qu'il fallait que j'accepte :

- `800−555−1212`
- `800 555 1212`
- `800.555.1212`
- `(800) 555−1212`
- `1−800−555−1212`
- `800−555−1212−1234`
- `800−555−1212x1234`
- `800−555−1212 ext. 1234`
- `work 1−(800) 555.1212 #1234`

Quelle diversité ! Dans chacun de ces cas, je devais savoir que le code régional était `800`, l'indicatif `555` et le reste du numéro `1212`. Pour les numéros avec extension, je devais savoir que celle−ci était `1234`.

Nous allons développer une solution pour la reconnaissance des numéros de téléphone. Cet exemple montre la première étape.

Exemple 7.10. Trouver des numéros

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') ❶
>>> phonePattern.search('800-555-1212').groups()            ❷
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')                ❸
>>>
```

❶ Lisez toujours les expressions régulières de la gauche vers la droite. Celle-ci reconnaît le début de la chaîne, puis `(\d{3})`. Que veut dire ce `\d{3}`? Le `{3}` signifie "reconnaître exactement trois chiffres", c'est une variante de la `syntaxe {n,m}` que nous avons vu plus haut. `\d` signifie "n'importe quel chiffre" (de `0` à `9`). En le mettant entre parenthèses, nous disons "reconnais exactement trois chiffres, *puis identifie-les comme un groupe que je peux rappeler plus tard*". Ensuite, le motif reconnaît un tiret, puis un autre groupe de trois chiffres, un autre tiret, un groupe de quatre chiffres et la fin de la chaîne.

❷ Pour accéder aux groupes que l'expression régulière à identifiés, utilisez la méthode `groups()` de l'objet que la fonction `search` retourne. Elle retournera un tuple du nombre de groupes définis dans l'expression régulières. Dans ce cas, nous avons défini trois groupes, deux de trois chiffres et un de quatre.

❸ Cette expression régulière n'est pas la réponse finale car elle ne prend pas en compte les numéros de téléphone avec une extension à la fin. Pour cela, nous allons devoir la modifier.

Exemple 7.11. Trouver l'extension

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') ❶
>>> phonePattern.search('800-555-1212-1234').groups()            ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234')                     ❸
>>>
>>> phonePattern.search('800-555-1212')                          ❹
>>>
```

❶ Cette expression régulière est pratiquement identique à la précédente. Elle reconnaît le début de la chaîne, puis un groupe identifié de trois chiffres, puis un tiret, puis un groupe identifié de trois chiffres, puis un tiret, puis un groupe identifié de quatre chiffres. Ce qui est nouveau, c'est qu'elle reconnaît ensuite un autre tiret puis un groupe identifié de un chiffre ou plus, puis la fin de la chaîne.

❷ La méthode `groups()` retourne maintenant un tuple de quatre éléments, puisque l'expression régulière définit quatre groupe à identifier.

❸ Malheureusement cette expression régulière n'est pas la réponse finale non plus, puisqu'elle considère que les différentes parties du numéro de téléphone sont séparées par des tirets. Et si elles étaient séparées par des espaces, des virgules ou des points ? Il nous faut une solution plus générale pour identifier différents types de séparateurs.

❹ Non seulement cette expression régulière ne fait pas tout ce que nous voulions, elle est en fait un pas en arrière puisqu'elle ne peut pas reconnaître de numéros de téléphone *sans* extension. Ce n'est pas du tout ce que nous voulions, si l'extension est présente, nous voulons la connaître, mais si elle ne l'est pas, nous voulons tout de même connaître les différentes parties du numéro.

L'exemple suivant montre l'expression régulière qui reconnaît les séparateurs entre les différentes parties d'un numéro de téléphone.

Exemple 7.12. Reconnaissance des séparateurs

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') ❶
```

```
>>> phonePattern.search('800 555 1212 1234').groups()                    ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups()                    ❸
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234')                                ❹
>>>
>>> phonePattern.search('800-555-1212')                                  ❺
>>>
```

❶ Faites bien attention. Nous reconnaissons le début de la chaîne, puis un groupe de trois chiffres, puis `\D+`. Qu'est-ce que c'est que ça ? Et bien, `\D` reconnaît n'importe quel caractère *sauf* un chiffre et `+` signifie "un ou plus". Donc `\D+` reconnaît un ou plusieurs caractères n'étant pas des chiffres. C'est ce que nous utilisons pour essayer de reconnaitre les séparateurs.

❷ Utiliser `\D+` au lieu de `-` nous permet de reconnaître des numéros de téléphone dont les différentes parties sont séparées par des espaces au lieu de tirets.

❸ Bien sûr, les numéro de téléphone séparés par des tirets sont toujours reconnus.

❹ Malheureusement ce n'est pas encore la réponse définitive car elle suppose qu'il y a bien un séparateur. Et si le numéro est saisi sans espaces ni tirets ?

❹ Le problème de l'extension optionnelle n'a toujours pas été réglé. Maintenant nous avons deux problèmes, mais nous pouvons les régler tous les deux grâce à la même technique.

L'exemple suivant montre l'expression régulière qui reconnaît les numéros de téléphone *sans* séparateurs.

Exemple 7.13. Reconnaissance des numéros de téléphone sans séparateurs

```
>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$')  ❶
>>> phonePattern.search('80055512121234').groups()                       ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups()                   ❸
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                         ❹
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234')                            ❺
>>>
```

❶ La seule modification depuis la dernière étape est de remplacer le `+` par un `*`. Au lieu de `\D+` entre les différentes parties du numéro de téléphone, nous avons maintenant `\D*`. Vous vous rappelez que `+` signifie "un ou plus" et bien `*` signifie "zéro ou plus". Nous devrions donc pouvoir reconnaître des numéros de téléphone qui n'ont pas de séparateur du tout.

❷ Et ça marche. Nous avons reconnu le début de la chaîne, puis un groupe identifié de trois chiffres (`800`), puis zéro caractères non numériques, puis un groupe identifié de trois caractères (`555`), puis zéro caractères non numériques, puis un groupe identifié de quatre caractères (`1212`), puis zéro caractères non numériques, puis un groupe identifié d'un nombre quelconque de caractères (`1234`), puis la fin de la chaîne.

❸ D'autre variantes marchent également maintenant : des points à la place des tirets et un espace et un `x` avant l'extension.

❹ Finalement, nous avons trouvé une solution à notre problème, les extension sont vraiment optionnelles. Si aucune extension n'est trouvée la méthode `groups()` retourne un tuple de quatre éléments, mais le quatrième élément est simplement une chaîne vide.

❺ La mauvaise nouvelle, c'est que nous n'avons pas terminé. Il y a un caractère supplémentaire avant le code régional, mais l'expression régulière suppose que le code régional est la première chose au début de la chaîne. Bien sûr, nous pouvons utiliser la même technique "zéro ou plus caractères non-numériques" pour sauter les caractères situés avant le code régional.

L'exemple suivant montre comment prendre en compte les caractères au début des numéros de téléphone.

Exemple 7.14. Reconnaissance des caractères de début

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('(800)5551212 ext. 1234').groups()                ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups()                          ❸
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234')                    ❹
>>>
```

❶ C'est la même chose que dans l'exemple précédent, sauf que nous reconnaissons `\D*`, c'est à dire zéro ou plus caractères non numériques, avant le premier groupe identifié (le code régional). Notez que nous n'identifions pas ces caractères non numériques (ils ne sont pas entre parenthèses). Si ils sont présents, nous les passons simplement et commençont à identifier le code régional quand nous y arrivons.

❷ Le numéro de téléphone est correctement reconnu, même avec la parenthèse ouvrante devant le code régional (la parenthèse fermante était déjà prise en compte, elle est considérée comme un séparateur non-numérique et reconnue par le `\D*` suivant le premier groupe identifié).

❸ Une simple vérification de cohérence pour nous assurer que nous n'avons rien endommagé de ce qui fonctionnait déjà. Puisque les caractères de début sont entièrement optionnels, le début de la chaîne est reconnu, puis zéro caractères non-numériques, puis un groupe identifié de trois caractères (`800`), puis un caractère non-numérique (le tiret), puis un groupe identifié de trois caractères (`555`), puis un caractère non-numérique (le tiret), puis un groupe identifié de quatre caractères (`1212`), puis zéro caractères non-numériques, puis un groupe identifié de zéro caractères numériques, puis la fin de la chaîne.

❹ Mais il y a encore un problème. Pourquoi ce numéro ne fonctionne-t-il pas ? Parce qu'il y a un `1` avant le code régional et que nous avons considéré que tous les caractères de début sont non-numériques (`\D*`).

Faisons le point. Jusqu'à présent, nos expressions régulières commençaient toujours la reconnaissance en début de chaîne, mais maintenant nous voyons qu'il peut y avoir un nombre indeterminé de caractères à ignorer en début de chaîne. Au lieu d'essayer de les identifier pour les sauter, essayons une approche différente : ne pas reconnaître explicitemment le début de la chaîne. C'est ce que nous verrons dans l'exemple suivant.

Exemple 7.15. Un numéro de téléphone, où qu'il soit

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ❶
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups()        ❷
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                                ❸
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234')                              ❹
('800', '555', '1212', '1234')
```

❶ Notez l'absence de `^` dans cette expression régulière. Nous ne reconnaissons plus le début de la chaîne. Rien ne nous oblige à reconnaître la chaîne entière, le moteur d'expressions régulières se débrouillera pour trouver où commence la reconnaissance.

❷ Maintenant, nous pouvons reconnaître un numéro de téléphone précédé de caractères et de chiffres et segmenté par des séparateurs de tout type et de toute taille.

❸ Contrôle de cohérence, ça fonctionne toujours.

❹ Cela aussi fonctionne toujours.

Vous avez pu voir comment les expressions régulières peuvent rapidemment échapper à tout contrôle. Parcourez les différentes versions de notre expression régulière, pouvez−vous dire quelles différences les séparent ?

Tant que nous comprenons encore la version finale (et c'est bien la version finale, si vous découvrez un cas qu'elle n'est pas capable de traiter, je ne veux pas en entendre parler), écrivons−la sous la forme d'une expression régulière détaillée avant d'oublier les choix que nous avons fait.

Exemple 7.16. Reconnaissance des numéros de téléphone (version finale)

```
>>> phonePattern = re.compile(r'''
                # don't match beginning of string, number can start anywhere
    (\d{3})     # area code is 3 digits (e.g. '800')
    \D*         # optional separator is any number of non−digits
    (\d{3})     # trunk is 3 digits (e.g. '555')
    \D*         # optional separator
    (\d{4})     # rest of number is 4 digits (e.g. '1212')
    \D*         # optional separator
    (\d*)       # extension is optional and can be any number of digits
    $           # end of string
    ''', re.VERBOSE)
>>> phonePattern.search('work 1−(800) 555.1212 #1234').groups()        ❶
('800', '555', '1212', '1234')
>>> phonePattern.search('800−555−1212')                                ❷
('800', '555', '1212', '')
```

❶ Mis à part le fait qu'elle est distribuée sur plusieurs lignes, c'est exactement la même expression régulière qu'à la dernière étape, il n'est donc pas étonnant qu'elle reconnaisse les mêmes entrées de manière identique.

❷ Vérification de cohérence finale. Oui, ça marche toujours, nous avons terminé.

Pour en savoir plus sur les expressions régulières

- La Regular Expression HOWTO (http://py−howto.sourceforge.net/regex/regex.html) explique les expressions régulières et leur usage en Python.
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume le module `re` (http://www.python.org/doc/current/lib/module−re.html).

7.7. Résumé

Nous n'avons vu que la pointe de la partie émergée de l'iceberg des possibilités offertes par les expressions régulières. En d'autres termes, même si vous vous sentez totalement dépassé, vous n'avez encore rien vu.

Vous devez maintenant être familiarisé avec les techniques suivantes :

- `^` reconnaît le début d'une chaîne.
- `$` reconnaît la fin d'une chaîne.
- `\b` reconnaît la limite d'un mot.
- `\d` reconnaît un chiffre.
- `\D` reconnaît un caractère non−numérique.
- `x?` reconnaît un caractère `x` optionnel (autrement dit, il reconnaît un `x` zéro ou une fois).
- `x*` reconnaît zéro ou plus `x`.
- `x+` reconnaît un ou plusieurs `x`.
- `x{n,m}` reconnaît un caractère `x` au moins `n` fois, mais pas plus de `m` fois.
- `(a|b|c)` reconnaît soit `a` soit `b` soit `c`.

- `(x)` en général est un *groupe identifié*. Vous pouvez obtenir la valeur de ce qui a été reconnu à l'aide de la méthode `groups()` de l'objet retourné par `re.search`.

Les expressions régulières sont extrêmement puissantes, mais elles ne sont pas la solution correcte pour chaque problème. Vous devriez en apprendre assez sur elles pour savoir quand elles sont appropriées, quand elle peuvent résoudre votre problème et quand elles causent plus de problèmes qu'elles n'en résolvent.

> Certaines personnes, face à un problème, se disent "je sais, je vais utiliser une expression régulière." Maintenant elles ont deux problèmes.
>
> −−Jamie Zawinski, dans comp.emacs.xemacs (http://groups.google.com/groups?selm=33F0C496.370D7C45%40netscape.com)

---

[2] Note du traducteur : Je suppose que cet exemple suffit pour les lecteurs francophones.

Chapitre 8. Traitement du HTML

8.1. Plonger

Je vois souvent sur comp.lang.python (http://groups.google.com/groups?group=comp.lang.python) des questions comme "Comment faire une liste de tous les [en−têtes|images|liens] de mon document HTML ?" "Comment faire pour [*parser*|traduire|transformer] le texte d un document HTML sans toucher aux balises ?" "Comment faire pour [ajouter|enlever|mettre entre guillemets] des attributs de mes balises HTML d un coup ?" Ce chapitre répondra à toutes ces questions.

Voici un programme Python complet et fonctionnel en deux parties. La première partie, `BaseHTMLProcessor.py`, est un outil générique destiné à vous aider à traiter des fichiers HTML en parcourant les balises et les blocs de texte. La deuxième partie, `dialect.py`, est un exemple montrant comment utiliser `BaseHTMLProcessor.py` pour traduire le texte d un document HTML sans toucher aux balises. Lisez les `doc strings` et les commentaires pour avoir une vue d ensemble de ce qui se passe. Une grande partie va avoir l air magique parce qu il n est pas évident de voir comment ces méthodes de classes sont appelées. Ne vous inquiétez pas, tout vous sera bientôt expliqué.

Exemple 8.1. `BaseHTMLProcessor.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Ideally we would like to reconstruct original tag and attributes, but
        # we may end up quoting attribute values that weren't quoted in the source
        # document, or we may change the type of quotes around the attribute value
        # (single to double quotes).
        # Note that improperly embedded non−HTML code (like client−side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime script errors.
        # All non−HTML code must be enclosed in HTML comment tags (<!−− code −−>)
        # to ensure that it will pass through this parser unaltered (in handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        # called for each character reference, e.g. for " ", ref will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())
```

```
    def handle_entityref(self, ref):
        # called for each entity reference, e.g. for "&copy;", ref will be "copy"
        # Reconstruct the original entity reference.
        self.pieces.append("&%(ref)s" % locals())
        # standard HTML entities are closed with a semicolon; other entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):
        # called for each block of plain text, i.e. outside of any tag and
        # not containing any character or entity references
        # Store the original text verbatim.
        self.pieces.append(text)

    def handle_comment(self, text):
        # called for each HTML comment, e.g. <!-- insert Javascript code here -->
        # Reconstruct the original comment.
        # It is especially important that the source document enclose client-side
        # code (like Javascript) within comments so it can pass through this
        # processor undisturbed; see comments in unknown_starttag for details.
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):
        # called for each processing instruction, e.g. <?instruction>
        # Reconstruct original processing instruction.
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        # called for the DOCTYPE, if present, e.g.
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Reconstruct original DOCTYPE
        self.pieces.append("<!%(text)s>" % locals())

    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

Exemple 8.2. `dialect.py`

```
import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # called for every </pre> tag in HTML source
```

```
        # Decrement verbatim mode count
        self.unknown_endtag("pre")
        self.verbatim -= 1

    def handle_data(self, text):
        # override
        # called for every block of text in HTML source
        # If in verbatim mode, save text unaltered;
        # otherwise process the text with a series of substitutions
        self.pieces.append(self.verbatim and text or self.process(text))

    def process(self, text):
        # called from handle_data
        # Process text block by performing series of regular expression
        # substitutions (actual substitions are defined in descendant)
        for fromPattern, toPattern in self.subs:
            text = re.sub(fromPattern, toPattern, text)
        return text

class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak

    based on the classic chef.x, copyright (c) 1992, 1993 John Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))

class FuddDialectizer(Dialectizer):
    """convert HTML to Elmer Fudd-speak"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))

class OldeDialectizer(Dialectizer):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
```

```
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))

def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect

    dialect in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()

def test(url):
    """test all dialects against URL"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
```

```
        fsock.write(translate(url, dialect))
        fsock.close()
        import webbrowser
        webbrowser.open_new(outfile)

if __name__ == "__main__":
    test("http://diveintopython.org/odbchelper_list.html")
```

Exemple 8.3. Sortie de `dialect.py`

Ce script effectue la traduction de la Section 3.2, «Présentation des listes» en pseudo−Chef Suédois (http://diveintopython.org/chef.html) (des Muppets), pseudo−Elmer Fudd (http://diveintopython.org/fudd.html) (de Bugs Bunny) et en pseudo−ancien Anglais (http://diveintopython.org/olde.html) (librement adapté de *The Canterbury Tales* de Chaucer). Si vous regardez le source HTML de la sortie, vous verrez que toutes les balises HTML et les attributs sont intacts mais que le texte entre les balises a été "traduit" dans le pseudo−langage. Si vous regardez plus attentivement, vous verrez qu en fait, seuls les titres et les paragraphes ont été traduits, les listing de code et les exemples d écrans ont été laissé intacts.

```
<div class="abstract">
<p>Lists awe <span class="application">Pydon</span>'s wowkhowse datatype.
If youw onwy expewience wif wists is awways in
<span class="application">Visuaw Basic</span> ow (God fowbid) de datastowe
in <span class="application">Powewbuiwdew</span>, bwace youwsewf fow
<span class="application">Pydon</span> wists.</p>
</div>
```

8.2. Présentation de `sgmllib.py`

Le traitement du HTML est divisé en trois étapes : diviser le HTML en éléments, modifier les éléments et reconstruire le HTML à partir des éléments. La première étape est réalisée par `sgmllib.py`, qui fait partie de la bibliothèque standard de Python.

La clé de la compréhension de ce chapitre est de réaliser que le HTML n est pas seulement du texte, c est du texte structuré. La structure est dérivée de la séquence plus ou moins hiérarchique de balises de début et de fin. Habituellement, on ne travaille pas de cette manière sur du HTML, on travaille *textuellement* dans un éditeur de texte ou *visuellement* dans un navigateur ou un éditeur de pages web. `sgmllib.py` présente le HTML de manière *structurelle*.

`sgmllib.py` contient une classe principale : `SGMLParser`. `SGMLParser` analyse le HTML et le décompose en éléments utiles, comme des balises de début et de fin. Dès qu il parvient à extraire des données un élément utile, il appelle une de ses propres méthodes en fonction de l élément trouvé. Pour utiliser l analyseur, on dérive une classe de `SGMLParser` et on redéfinit ces méthodes. C est ce que j entendais par présentation du HTML de manière *structurelle* : la structure du code HTML détermine la séquence d appels de méthodes et les arguments passés à chaque méthode.

`SGMLParser` décompose le HTML en 8 sortes de données et appelle une méthode différente pour chacune d entre elles :

Balise de début
: Une balise HTML qui ouvre un bloc, comme `<html>`, `<head>`, `<body>` ou `<pre>`, ou une balise autonome comme `<br>` ou `<img>`. Lorsqu il trouve une balise de début *`tagname`*, `SGMLParser` cherche une méthode nommée `start_`*`tagname`* ou `do_`*`tagname`*. Par exemple, lorsqu il trouve une balise `<pre>`, il cherche une méthode nommée `start_pre` ou `do_pre`. S il la trouve, `SGMLParser` l appelle

avec une liste des attributs de la balise en paramètre, sinon il appelle `unknown_starttag` avec le nom de la balise et la liste de ses attributs en paramètre.

Balise de fin
: Une balise HTML qui ferme un bloc, comme `</html>`, `</head>`, `</body>` ou `</pre>`. Lorsqu il trouve une balise de fin, `SGMLParser` cherche une méthode nommée `end_`*`tagname`*. S il la trouve, `SGMLParser` appelle cette méthode, sinon il appelle `unknown_endtag` avec le nom de la balise.

Référence de caractère
: Un caractère référencé par son équivalent décimal ou hexadécimal, comme ` `. Lorsqu il en trouve une, `SGMLParser` appelle `handle_charref` avec le texte de l équivalent décimal ou hexadécimal.

Référence d entité
: Une entité HTML, comme `&copy;`. Lorsqu il en trouve une, `SGMLParser` appelle `handle_entityref` avec le nom de l entité HTML.

Commentaire
: Un commentaire HTML, encadré par `<!-- ... -->`. Lorsqu il en trouve un, `SGMLParser` appelle `handle_comment` avec le corps du commentaire.

Instruction de traitement
: Une instruction de traitement HTML, encadrée par `<? ... >`. Lorsqu il en trouve une, `SGMLParser` appelle `handle_pi` avec le corps de l instruction.

Déclaration
: Une déclaration HTML, comme un `DOCTYPE`, encadrée par `<! ... >`. Lorsqu il en trouve une, `SGMLParser` appelle `handle_decl` avec le corps de la déclaration

Données texte
: Un bloc de texte. Tout ce qui n entre dans aucune des 7 catégories précédentes. Lorsqu il en trouve un, `SGMLParser` appelle `handle_data` avec le texte.

Python 2.0 avait un bogue qui empêchait `SGMLParser` de reconnaître les déclarations (`handle_decl` n était jamais appelé), ce qui veut dire que les `DOCTYPE`s étaient ignorés silencieusement. Ce bogue est corrigé dans Python 2.1.

`sgmllib.py` est accompagné d'une suite de tests pour illustrer cela. Si on exécute `sgmllib.py` en lui passant le nom d un fichier HTML en argument de ligne de commande, il affichera les balises et les autres éléments au fur et à mesure qu il analyse le fichier. Il fait cela en dérivant une classe de `SGMLParser` et en définissant des méthodes comme `unknown_starttag`, `unknown_endtag`, `handle_data` et autres qui ne font qu afficher leur argument.

Dans l IDE ActivePython sous Windows, vous pouvez spécifier des arguments de ligne de commande dans la boîte de dialogue "Run script". Séparez les différents arguments par des espaces.

Exemple 8.4. Exemple de test de `sgmllib.py`

Voici un extrait de la table des matières de la version HTML de ce livre. Bien sûr, les chemins peuvent être différents. Si vous n'avez pas téléchargé la version HTML du livre, vous pouvez le faire à l'adresse suivante : http://diveintopython.org/.

```
c:\python23\lib> type "c:\downloads\diveintopython\html\toc\index.html"
 <!DOCTYPE html
  PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

      <title>Dive Into Python</title>
```

```
      <link rel="stylesheet" href="diveintopython.css" type="text/css">

... nous coupons la suite pour rester bref ...
```

En l utilisant avec la suite de tests de `sgmllib.py`, on obtient la sortie suivante :

```
c:\python23\lib> python sgmllib.py "c:\downloads\diveintopython\html\toc\index.html"
data: '\n\n'
start tag: <html lang="en" >
data: '\n   '
start tag: <head>
data: '\n      '
start tag: <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" >
data: '\n   \n      '
start tag: <title>
data: 'Dive Into Python'
end tag: </title>
data: '\n      '
start tag: <link rel="stylesheet" href="diveintopython.css" type="text/css" >
data: '\n      '

... nous coupons la suite pour rester bref ...
```

Voici le plan du reste de ce chapitre :

- Dérivation de `SGMLParser` pour créer des classes qui extraient des données intéressantes de documents HTML.
- Dérivation de `SGMLParser` pour créer `BaseHTMLProcessor`, qui redéfinit les 8 méthodes de gestion et les utilise pour reconstruire le code HTML à partir de ses éléments.
- Dérivation de `BaseHTMLProcessor` pour créer `Dialectizer`, qui ajoute des méthodes traitant des balises HTML spécifiques et redéfinit la méthode `handle_data` pour fournir une structure de traitement de blocs de tests entre les balises HTML.
- Dérivation de `Dialectizer` pour créer des classes qui définissent des règles de traitement du texte utilisées par `Dialectizer.handle_data`.
- Ecriture d une suite de tests qui récupère une véritable page web de `http://diveintopython.org/` et en traite le contenu.

En cours de route, vous apprendrez également `locals`, `globals` et le formatage de chaînes à l'aide de dictionnaire.

8.3. Extraction de données de documents HTML

Pour extraire des données de documents HTML, on dérive une classe de `SGMLParser` et on définit des méthodes pour chaque balise ou entité que l on souhaite traiter.

La première étape pour extraire des données d un document HTML est d obtenir le HTML. Si vous avez un fichier HTML, vous pouvez le lire à l aide des fonctions de fichier, mais le plus intéressant est d obtenir le HTML depuis des sites web.

Exemple 8.5. Présentation de `urllib`

```
>>> import urllib                                          ❶
>>> sock = urllib.urlopen("http://diveintopython.org/")    ❷
>>> htmlSource = sock.read()                               ❸
>>> sock.close()                                           ❹
>>> print htmlSource                                       ❺
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
      <meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'>
   <title>Dive Into Python</title>
<link rel='stylesheet' href='diveintopython.css' type='text/css'>
<link rev='made' href='mailto:mark@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial, object-oriented, programming, docu
<meta name='description' content='a free Python tutorial for experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084' alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline' colspan='2'>Python for experienced programmers</td></tr>

[...coupé...]
```

❶ Le module `urllib` fait partie de la bibliothèque standard de Python. Il comprend des fonctions permettant d obtenir des information et des données à partir d URLs (principalement des pages web).

❷ L usage le plus simple de `urllib` est de lire le texte complet d une page web à l aide de la fonction `urlopen`. L ouverture d une URL est semblable à l ouverture d un fichier. La valeur de retour de `urlopen` est un objet semblable à un objet-fichier et dont certaines méthodes sont les mêmes que celles d un objet-fichier.

❸ La chose la plus simple à faire avec l objet retourné par `urlopen` est d appeler `read`, qui lit l ensemble du code HTML de la page web en une chaîne unique. L objet permet également l emploi de `readlines`, qui lit le code ligne par ligne et le stocke dans une liste.

❹ Quand vous n en avez plus besoin, assurez vous de fermer l objet par un `close`, comme pour un objet fichier.

❺ Nous avons maintenant l ensemble du code HTML de la page d accueil de `http://diveintopython.org/` dans une chaîne et nous sommes prêts à l analyser.

Exemple 8.6. Présentation de `urllister.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython-examples-5.4.zip) du livre.

```
from sgmllib import SGMLParser

class URLLister(SGMLParser):
    def reset(self):                              ❶
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):                     ❷
        href = [v for k, v in attrs if k=='href'] ❸ ❹
        if href:
            self.urls.extend(href)
```

❶ `reset` est appelé par la méthode `__init__` de `SGMLParser` et peut également être appelé manuellement quand une instance de l analyseur a été créée. Donc, si vous avez une quelconque initialisation à faire, faites la dans `reset` et pas dans `__init__`, de manière à ce que la réinitialisation se fasse correctement lorsque quelqu un réutilise une instance de l analyseur.

❷ `start_a` est appelé par `SGMLParser` à chaque fois qu il trouve une balise `<a>`. La balise peut contenir un attribut `href` et/ou d autres attributs comme `name` ou `title`. Le paramètre `attrs` est une liste de tuples,

`[(attribut, valeur), (attribut, valeur), ...]`. La balise peut aussi être un simple `<a>`, ce qui est une balise HTML valide (bien qu inutile), dans ce cas `attrs` sera une liste vide.

❸ Nous pouvons savoir si la balise `<a>` a un attribut `href` à l aide d une simple mutation de liste multi−variable.

❹ Les comparaisons de chaînes comme `k=='href'` sont toujours sensibles à la casse, mais ça ne pose pas de problème ici car `SGMLParser` convertit les noms d attributs en minuscules lors de la création de `attrs`.

Exemple 8.7. Utilisation de `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())          ❶
>>> usock.close()                      ❷
>>> parser.close()                     ❸
>>> for url in parser.urls: print url  ❹
toc/index.html
#download
#languages
toc/index.html
appendix/history.html
download/diveintopython−html−5.0.zip
download/diveintopython−pdf−5.0.zip
download/diveintopython−word−5.0.zip
download/diveintopython−text−5.0.zip
download/diveintopython−html−flat−5.0.zip
download/diveintopython−xml−5.0.zip
download/diveintopython−common−5.0.zip


... nous coupons la suite pour rester bref ...
```

❶ Appelez la méthode `feed`, définie dans `SGMLParser`, pour charger le code HTML dans l analyseur. [3] La méthode prend une chaîne en argument, ce qui est ce que `usock.read()` retourne.

❷ Comme pour les fichiers, vous devez fermer vos objets URL par `close` dès que vous n en avez plus besoin.

❸ Vous devez également fermer l objet analyseur par `close`, mais pour une raison différente. La méthode `feed` ne garantit pas qu elle traite tout le code HTML que vous lui passez, elle peut la garder dans un tampon en attendant que vous lui en passiez plus. Quand il n y en a plus, appelez `close` pour vider le tampon et forcer l analyse de tout le code.

❹ Une fois le parser fermé par `close`, l analyse est complète et `parser.urls` contient une liste de toutes les URLs pour lesquelles il y a un lien dans le document HTML.

8.4. Présentation de `BaseHTMLProcessor.py`

`SGMLParser` ne produit rien de lui même. Il ne fait qu analyser et appeler une méthode pour chaque élément intéressant qu il trouve, mais les méthodes ne font rien. `SGMLParser` est un *consommateur* de HTML : il prend du code HTML et le décompose en petits éléments structurés. Comme nous l avons vu dans la section précédente, on peut dériver `SGMLParser` pour définir une classe qui trouve des balises spécifiques et produit quelque chose d utile, comme une liste de tous les liens d une page web. Nous allons maintenant aller un peu plus loin en définissant une classe qui prends tout ce que `SGMLParser` lui envoi et reconstruit entièrement le document HTML. En termes techniques, cette classe sera un *producteur* de HTML.

`BaseHTMLProcessor` est dérivé de `SGMLParser` et fournit les 8 méthodes de prise en charge essentielles : `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl` et `handle_data`.

Exemple 8.8. Présentation de `BaseHTMLProcessor`

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):                              ❶
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):      ❷
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):               ❸
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):               ❹
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):             ❺
        self.pieces.append("&%(ref)s" % locals())
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):                 ❻
        self.pieces.append(text)

    def handle_comment(self, text):              ❼
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):                   ❽
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        self.pieces.append("<!%(text)s>" % locals())
```

❶ `reset` est appelé par `SGMLParser.__init__` initialise `self.pieces` en une liste vide avant d appeler la méthode ancêtre. `self.pieces` est une donnée attribut qui contient les éléments du document HTML que nous assemblons. Chaque méthode de prise en charge va reconstruire le code HTML que `SGMLParser` a analysé et chaque méthode ajoutera la chaîne résultante à `self.pieces`. Notez que `self.pieces` est une liste. Vous pouvez être tenté de la définir comme une chaîne et de lui ajouter simplement chaque élément. Cela fonctionnerait mais Python gère les listes de manière bien plus efficiente.[4]

❷ Comme `BaseHTMLProcessor` ne définit aucune méthode pour des balises spécifiques (comme la méthode `start_a` de `urllister.py`), `SGMLParser` appelera `unknown_starttag` pour chaque balise de début. Cette méthode prend en paramètre la balise (`tag`) et la liste des paires nom/valeurs de ses attributs (`attrs`), reconstruit le code HTML originel et l ajoute à `self.pieces`. Le formatage de chaîne ici est un peu étrange, nous l expliquerons (ainsi que la fonction `locals` à l'air étrange) dans la prochaine section.

❸ Reconstruire les balises de fin est beaucoup plus simple, il suffit de prendre le nom de la balise est de l encadrer de `</...>`.

❹ Lorsque `SGMLParser` trouve une référence de caractère, il appelle `handle_charref` avec la référence. Si le document HTML contient la référence ` `, `ref` vaudra `160`. La reconstruction de la référence de caractère originelle ne demande que d encadrer `ref` par `&#...;`.

❺ Les références d entité sont semblables aux références de caractères, mais sans le signe dièse. La

reconstruction de la référence d entité originelle demande d encadrer `ref` par `&...;`. (En fait, comme un lecteur savant me l a fait remarquer, c est un peu plus compliqué que ça. Seulement certaines entités standard du HTML finissent par un point−virgule. Heureusement pour nous, l ensemble des entités standards est défini dans un dictionnaire dans un module Python appelé `htmlentitydefs`. C est l explication de l instruction `if` supplémentaire.)

❻ Les blocs de texte sont simplement ajouté à `self.pieces` sans modification.

❼ Les commentaires HTML sont encadrés par `<!--...-->`.

❽ Les instructions de traitement sont encadrés par `<?...>`.

La spécification HTML exige que tous les éléments non−HTML (comme le JavaScript côté client) soient compris dans des commentaires HTML, mais toutes les pages web ne le font pas (et les navigateurs web récents ne l exigent pas). `BaseHTMLProcessor`, lui, l exige, si le script n est correctement encadré dans un commentaire, il sera analysé comme s il était du code HTML. Par exemple, si le script contient des signes inférieurs à ou égal, `SGMLParser` peut considérer à tort qu il a trouvé des balises et des attributs. `SGMLParser` convertit toujours les noms de balises et d attributs en minuscules, ce qui peut empêcher la bonne exécution du script et `BaseHTMLProcessor` entoure toujours les valeurs d attributs entre guillemets (même si le document HTML n en utilisait pas ou utilisait des guillemets simples), ce qui empêchera certainement l exécution du script. Protégez toujours vos script côté client par des commentaires HTML.

Exemple 8.9. `Sortie de BaseHTMLProcessor`

```
    def output(self):                    ❶
        """Return processed HTML as a single string"""
        return "".join(self.pieces) ❷
```

❶ Voici l unique méthode de `BaseHTMLProcessor` qui n est jamais appelée par son ancêtre, `SGMLParser`. Comme les méthodes de prise en charge stockent le HTML reconstitué dans `self.pieces`, cette fonction est nécessaire pour assembler toutes ces pièces en une chaîne unique. Comme noté précédemment, Python est bon pour gérer les listes et moyens pour les chaînes, nous ne créons donc la chaîne seulement quand un utilisateur la réclame explicitement.

❷ Si vous préférez, vous pouvez plutôt utiliser la méthode `join` du module `string` : `string.join(self.pieces, "")`

Pour en savoir plus

- Le W3C (http://www.w3.org/) traite des références de caractères et d entités (http://www.w3.org/TR/REC−html40/charset.html#entities).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) confirme vos soupçons selon lesquels le module (http://www.python.org/doc/current/lib/module−htmlentitydefs.html) est exactement ce que son nom laisse deviner.

8.5. `locals` et `globals`

Laissons de coté le traitement du HTML une minute pour parler de la manière dont Python gère les variables. Python a deux fonctions prédéfinies permettant d accéder aux variables locales et globales sous forme de dictionnaire : `locals` et `globals`.

Vous vous rappelez de `locals` ? Vous l'avez vu pour la première fois ici :

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())
```

Mais vous ne pouvez rien apprendre sur `locals` pour le moment. Vous devez d'abord apprendre les espaces de noms. C est un concept un peu aride mais important, lisez donc attentivement.

Python utilise ce que l on appelle des espaces de noms pour suivre les variables. Un espace de noms est semblable a un dictionnaire dans lequel les clés sont les noms des variables et les valeurs du dictionnaire sont les valeurs des variables. En fait, on accède à un espace de noms comme à un dictionnaire Python, comme nous le verrons un peu plus loin.

A n importe quel point dans un programme Python, il y a plusieurs espaces de noms disponibles. Chaque fonction a son propre espace de noms, appelé espace de noms local, qui suit les variables de la fonction, y compris ses arguments et les variables définies localement. Chaque module a son propre espace de noms, appelé l espace de noms global, qui suit les variables du module, y compris les fonctions, les classes, les modules importés et les variables et constantes du module. Il y a également un espace de noms prédéfini, accessible de n importe quel module et qui contient les fonctions et exceptions du langage.

Lorsqu une ligne de code demande la valeur d une variable `x`, Python recherche cette variable dans tous les espaces de noms disponibles dans l ordre suivant :

1. Espace de noms local − spécifique à la fonction ou méthode de classe en cours. Si la fonction a défini une variable locale `x`, ou si elle a un argument `x`, Python l utilise et arrête sa recherche.
2. Espace de noms global − spécifique au module en cours. Si le module a défini une variable, une fonction ou une classe nommée `x`, Python l utilise et arrête sa recherche.
3. Espace de noms prédéfini − global à tous les modules. En dernière instance, Python considère que `x` est le nom d une fonction ou variable du langage.

Si Python ne trouve `x` dans aucun de ces espaces de noms, il abandonne et déclenche une exception `NameError` avec le message `There is no variable named 'x'`, que vous avez vu tout au début au chapitre 1, mais à ce moment là vous ne pouviez pas savoir tout le travail que Python fait avant de vous renvoyer cette erreur.

Python 2.2 a introduit une modification légère mais importante qui affecte l ordre de recherche dans les espaces de noms : les portées imbriquées. Dans les versions précédentes de Python, lorsque vous référenciez une variable dans une fonction imbriquée ou une fonction `lambda`, Python recherchait la variable dans l espace de noms de la fonction (imbriquée ou `lambda`) en cours, puis dans l espace de noms du module. Python 2.2 recherche la variable dans l espace de noms de la fonction (imbriquée ou `lambda`) en cours, *puis dans l espace de noms de la fonction parente*, puis dans l espace de noms du module. Python 2.1 peut adopter l'un ou l'autre de ces comportements, par défaut il fonctionne comme Python 2.0, mais vous pouvez ajouter la ligne de code suivante au début de vos modules pour les faire fonctionner comme avec Python 2.2 :

```
from __future__ import nested_scopes
```

Vous êtes perdu ? Ne vous inquiétez pas, je vous promet que c'est très utile. Comme beaucoup de chose en Python, les espaces de noms sont *directement accessibles durant l exécution*. L espace de noms local est accessible par la fonction prédéfinie `locals` et l espace de noms global (du module) est accessible par la fonction prédéfinie `globals`.

Exemple 8.10. Présentation de `locals`

```
>>> def foo(arg): ❶
...     x = 1
...     print locals()
...
>>> foo(7)       ❷
```

```
{'arg': 7, 'x': 1}
>>> foo('bar')    ❸
{'arg': 'bar', 'x': 1}
```

❶ La fonction `foo` a deux variables dans son espace de noms local : `arg`, dont la valeur est passée à la fonction et `x`, qui est définie dans la fonction.

❷ `locals` retourne un dictionnaire de paires nom/valeur. Les clés du dictionnaire sont les noms des variables sous forme de chaînes, les valeurs du dictionnaire sont les valeurs des variables. Donc, appeler `foo` avec `7` affiche un dictionnaire contenant les deux variables locales de la fonction : `arg` (`7`) et `x` (`1`).

❸ Rappelez−vous que Python est à typage dynamique, vous pouvez donc passer une chaîne pour `arg`, la fonction (et l appel à `locals`) fonctionne tout aussi bien. `locals` fonctionne avec toutes les variables de tous les types.

Ce que fait `locals` pour l espace de noms local (de la fonction), `globals` le fait pour l espace de noms global (du module). `globals` est cependant plus intéressant, parce que l espace de noms d un module est plus intéressant. L espace de noms d un module ne comprend pas seulement les variables et constantes du module mais aussi l ensemble des fonctions et classes définies dans le module. De plus, il contient tout ce qui a été importé dans le module.

Vous rappelez−vous de la différence entre `from` *`module`* `import` et `import` *`module`* ? Avec `import` *`module`*, le module lui−meme est importé mais il garde son propre espace de noms, c est pourquoi vous devez utiliser le nom du module pour accéder à ses fonctions ou attributs : *`module.fonction`*. Mais avec `from` *`module`* `import`, vous importez des fonctions et des attributs spécifiques d un autre module dans votre propre espace de noms, c est pourquoi vous y accédez directement sans référence au module dont ils viennent. Avec la fonction `globals`, vous pouvez voir ce qui se passe.

Exemple 8.11. Présentation de `globals`

Regardez me bloc de code suivant, à la fin de `BaseHTMLProcessor.py`:

```
if __name__ == "__main__":
    for k, v in globals().items():           ❶
        print k, "=", v
```

❶ Ne soyez pas intimidé, vous avez déjà vu tout cela. La fonction `globals` retourne un dictionnaire et nous parcourons le dictionnaire à l aide de la méthode `items` et de l assignement multiple. Le seul élément nouveau ici est la fonction `globals`.

Maintenant, exécuter le programme de la ligne de commande nous donne la sortie suivante (notez qu'elle peut être légèrement différente en fonction de votre plate−forme et de l'endroit où vous avez installé Python) :

```
c:\docbook\dip\py> python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                ❶
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python23\lib\htmlentitydefs.py'> ❷
BaseHTMLProcessor = __main__.BaseHTMLProcessor ❸
__name__ = __main__                            ❹
... rest of output omitted for brevity...
```

❶ `SGMLParser` a été importé de `sgmllib`, en utilisant `from` *`module`* `import`. Cela veut dire qu il a été importé directement dans l espace de noms du module, et nous le voyons donc s afficher.

❷ Comparez avec `htmlentitydefs`, qui a été importé avec `import`. Le module `htmlentitydefs` lui−même est dans notre espace de noms, mais la variable `entitydefs` définie dans `htmlentitydefs` ne l est pas.

❸ Ce module ne définit qu une classe, `BaseHTMLProcessor` et la voici. Notez que la valeur est ici la classe elle−même et non une instance quelconque de cette classe.

❹ Vous rappelez−vous de l astuce `if __name__` ? Lorsque vous exécutez un module (plutôt que de l importer d un autre module), l attribut prédéfini `__name__` a une valeur spéciale, `__main__`. Comme nous avons exécuté ce module comme un programme de la ligne de commande, `__name__` vaut `__main__`, c est pourquoi notre petit code de test qui affiche `globals` est exécuté.

A l aide des fonctions `locals` et `globals`, vous pouvez obtenir la valeur d une variable quelconque dynamiquement, en fournissant le nom de la variable sous forme de chaîne. C est une fonctionnalité semblable à celle de la fonction `getattr`, qui vous permet d accéder à une fonction quelconque dynamiquement en fournissant le nom de la fonction sous la forme d une chaîne.

Il y a une différence importante entre `locals` et `globals` que vous devez apprendre maintenant pour ne pas qu elle vous joue des tours plus tard. Elle vous jouera des tours de toute manière mais au moins vous vous souviendrez que vous l avez appris.

Exemple 8.12. `locals` est en lecture seule, `globals` ne l'est pas

```
def foo(arg):
    x = 1
    print locals()    ❶
    locals()["x"] = 2 ❷
    print "x=",x      ❸

z = 7
print "z=",z
foo(3)
globals()["z"] = 8    ❹
print "z=",z          ❺
```

❶ Puisque `foo` est appelé avec `3` en paramètre, cela affichera `{'arg': 3, 'x': 1}`. Cela ne devrait pas surprendre.

❷ `locals` est une fonction qui retourne un dictionnaire et ici vous changez une valeur dans ce dictionnaire. Vous pourriez penser que cela change la valeur de la variable locale `x` à `2`, mais ce n est pas le cas. `locals` ne retourne pas vraiment l espace de noms local mais une copie, modifier le dictionnaire retourné ne change pas les variables de l espace de noms local.

❸ Ceci affiche `x= 1`, pas `x= 2`.

❹ Après avoir été déçu par `locals`, vous pourriez penser que cela *ne va pas* changer la valeur de `z`, mais ce serait une erreur. Pour des raisons ayant trait à l implémentation de Python (dans le détail desquelles je ne rentrerais pas, ne les comprenant pas totalement), `globals` retourne l espace de noms global lui−même et non une copie, le comportement inverse de `locals`. Donc, toute modification du dictionnaire retourné par `globals` affecte les variables globales.

❺ Ceci affiche `z= 8`, pas `z= 7`.

8.6. Formatage de chaînes à l aide d un dictionnaire

Pourquoi avoir appris `locals` et `globals` ? Pour apprendre le formatage de chaînes à l'aide d'un dictionnaire. Comme vous vous le rappelez, le formatage de chaînes permet d insérer facilement des valeurs dans des chaînes. Les valeurs sont énumérées dans un tuple et insérées dans l ordre dans la chaîne à la place de chaque marqueur de formatage. Bien que ce soit efficace, cela ne donne pas le code le plus simple à lire, surtout quand de multiples valeurs sont insérées. Vous ne pouvez pas simplement lire la chaîne en une fois pour comprendre ce que le résultat va être, vous devez constamment passer de la chaîne au tuple.

Il existe une technique de formatage de chaînes alternative utilisant un dictionnaire au lieu de valeurs stockées dans un tuple.

Exemple 8.13. Présentation du formatage de chaînes à l aide d un dictionnaire

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> "%(pwd)s" % params                                          ❶
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params       ❷
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params       ❸
'master of mind, master of body'
```

❶ Au lieu d un tuple de valeurs explicites, ce type de formatage de chaînes utilise un dictionnaire, `params`. Et au lieu d un simple marqueur `%s` dans la chaîne, le marqueur contient un nom entre parenthèses. Ce nom est utilisé comme clé dans le dictionnaire `params` et la valeur correspondante, `secret`, est substituée au marqueur `%(pwd)s`.

❷ Le formatage à l aide d un dictionnaire fonctionne avec n importe quel nombre de clés nommées. Chaque clé doit exister dans le dictionnaire, sinon le formatage échouera avec une erreur `KeyError`.

❸ Vous pouvez même insérer la même clé plusieurs fois, chaque occurrence sera remplacée avec la même valeur.

Quand utiliser le formatage à l aide d un dictionnaire ? Il est un peu exagéré de mettre en place un dictionnaire de clés et de valeurs simplement pour formater une ligne, c est en fait plus approprié lorsque vous avez déjà un dictionnaire. Comme par exemple `locals`.

Exemple 8.14. Formatage à l aide d un dictionnaire dans `BaseHTMLProcessor.py`

```
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals())  ❶
```

❶ L utilisation de la fonction prédéfinie `locals` est le cas le plus commun d emploi du formatage à l aide d un dictionnaire. Cela vous permet d utiliser les noms des variables locales dans votre chaîne (dans ce cas, `text`, qui a été passé en argument à la méthode de classe) et chaque variable sera remplacée par sa valeur. Si `text` est `'Begin page footer'`, le formatage de chaîne `"<!--%(text)s-->" % locals()` se traduira par la chaîne `'<!--Begin page footer-->'`.

Exemple 8.15. Autres exemples de formatage à l aide d un dictionnaire

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])  ❶
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())                   ❷
```

❶ Lorsque cette méthode est appelée, `attrs` est une liste de tuples clé/valeur, comme les `items` (éléments) d un dictionnaire, ce qui signifie que nous pouvons utiliser l assignement multiple pour la parcourir. Cela devrait être un motif familier maintenant, mais il se passe beaucoup de choses ici, détaillons−les :

a. Supposez que `attrs` vaut `[('href', 'index.html'), ('title', 'Go to home page')]`.
b. Durant la première étape de la *list comprehension*, la clé (`key`) sera `'href'` et la valeur (`value`) `'index.html'`.
c. Le formatage de chaîne `' %s="%s"' % (key, value)` donnera

`' href="index.html"'`. Cette chaîne devient le premier élément de la valeur de retour de la *list comprehension*.

d. Durant la seconde étape, `key` sera `'title'` et `value` `'Go to home page'`.
e. Le formatage de chaîne donnera `' title="Go to home page"'`.
f. La *list comprehension* retourne une liste de ces deux chaînes et `strattrs` joindra les deux éléments de cette liste pour former `' href="index.html" title="Go to home page"'`.

❷ Maintenant, en utilisant le formatage à l aide d un dictionnaire, nous insérons la valeur de `tag` et de `strattrs` dans une chaîne. Donc si `tag` vaut `'a'`, le résultat final sera `'<a href="index.html" title="Go to home page">'` et c est ce qui sera ajouté à `self.pieces`.

L utilisation du formatage de chaîne à l aide d un dictionnaire avec `locals` est une manière pratique de rendre des expressions de formatage complexes plus lisibles, mais elle a un prix. Il y a une petite baisse de performance due à l appel de `locals`, puisque `locals` effectue une copie de l espace de noms local.

8.7. Mettre les valeurs d attributs entre guillemets

Une question courante sur comp.lang.python (http://groups.google.com/groups?group=comp.lang.python) est la suivante : "J ai plein de documents HTML avec des valeurs d attributs sans guillemets et je veux les mettre entre guillemets. Comment faire ?"[5] (C est en général du à un chef de projet qui pratique la religion du HTML−est−un−standard et proclame que toutes les pages doivent passer les tests d un validateur HTML. Les valeurs d attributs sans guillemets sont une violation courante du standard HTML). Quelle que soit la raison, les valeurs d attributs peuvent se voir dotées de guillemets en soumettant le HTML à `BaseHTMLProcessor`.

`BaseHTMLProcessor` prend du HTML en entrée (puisqu il est dérivé de `SGMLParser`) et produit du HTML, mais le HTML en sortie n est pas identique à l entrée. Les balises et les noms d attributs sont mis en minuscules et les valeurs d attributs sont mises entre guillemets, quel qu ait été les format en entrée. C est de cet effet de bord que nous pouvons profiter.

Exemple 8.16. Mettre les valeurs d attributs entre guillemets

```
>>> htmlSource = """                    ❶
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Home</a></li>
...     <li><a href=toc.html>Table of contents</a></li>
...     <li><a href=history.html>Revision history</a></li>
...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource)             ❷
>>> print parser.output()               ❸
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
```

```
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>
```

❶ Notez que la valeur de l attribut `href` de la balise `<a>` n est pas entre guillemets. (Notez aussi que nous utilisons des triples guillemets pour quelque chose d autre qu une `doc string` et directement dans l IDE. Elles sont très utiles.)

❷ On passe la chaîne au *parser*.

❸ En utilisant la fonction `output` définie dans `BaseHTMLProcessor`, nous obtenons la sortie sous forme d une chaîne unique, avec les valeurs d attributs entre guillemets. Cela peut sembler évident, mais réfléchissez à tout ce qui s est passé ici : `SGMLParser` a analysé le document HTML en entier, le décomposant en balises, références, données etc. ; `BaseHTMLProcessor` a utilisé tous ces éléments pour reconstruire des pièces de HTML (qui sont encore stockées dans `parser.pieces`, si vous voulez les voir) ; finalement, nous avons appelé `parser.output`, qui a assemblé l ensemble des pièces de HTML en une chaîne.

8.8. Présentation de `dialect.py`

`Dialectizer` est un descendant simple (et humoristique) de `BaseHTMLProcessor`. Il procède à une série de substitutions dans un bloc de text, mais il s assure que tout ce qui est contenu dans un bloc `<pre>...</pre>` passe sans altération.

Pour traiter les blocs `<pre>`, nous définissons deux méthodes dans `Dialectizer`: `start_pre` et `end_pre`.

Exemple 8.17. Traitement de balises spécifiques

```
    def start_pre(self, attrs):                ❶
        self.verbatim += 1                     ❷
        self.unknown_starttag("pre", attrs)    ❸

    def end_pre(self):                         ❹
        self.unknown_endtag("pre")             ❺
        self.verbatim -= 1                     ❻
```

❶ `start_pre` est appelé à chaque fois que `SGMLParser` trouve une balise `<pre>` dans le source HTML. (Nous verrons bientôt comment cela se fait.) La méthode prend un paramètre unique, `attrs`, qui contient les attributs de la balise (s il y en a). `attrs` est une liste de tuples clé/valeur, exactement le paramètre que prend `unknown_starttag`.

❷ Dans la méthode `reset`, nous initialisons un attribut qui sert de compteur de balises `<pre>`. Chaque fois que nous rencontrons une balise `<pre>`, nous incrémentons le compteur ; chaque fois que nous rencontrons une balise fermante `</pre>`, nous décrémentons le compteur. (Nous pourrions l utiliser simplement comme un drapeau en le mettant à `1` puis à `0`, mais c est aussi facile de cette manière et permet de traiter le cas improbable (mais possible) de balises `<pre>` imbriquées.) Dans une minute, nous verrons comment ce compteur est mis à profit.

❸ C est tout, c est le seul traitement particulier que nous faisons pour la balise `<pre>`. Maintenant, nous passons la liste des attributs à `unknown_starttag` pour qu il puisse faire le traitement par défaut.

❹ `end_pre` est appelé chaque fois que `SGMLParser` trouve une balise fermante `</pre>`. Comme les balises fermantes ne peuvent pas contenir d attributs, la méthode ne prend pas de paramètre.

❺ D abord nous voulons effectuer le traitement par défaut, comme pour toute balise fermante.

❻ Ensuite, nous décrémentons notre compteur pour signaler que ce bloc `<pre>` a été fermé.

Arrivé à ce point, il est temps d examiner plus en détail `SGMLParser`. J ai prétendu jusqu à maintenant (et vous avez dû me croire sur parole) que `SGMLParser` cherche et appelle des méthodes spécifiques pour chaque balise, si elles existent. Par exemple, nous venons juste de voir la définition de `start_pre` et `end_pre` pour traiter `<pre>` et `</pre>`. Mais comment est−ce que cela se produit ? Et bien, ce n est pas de la magie, simplement de la bonne programmation Python.

Exemple 8.18. `SGMLParser`

```
    def finish_starttag(self, tag, attrs):                    ❶
        try:
            method = getattr(self, 'start_' + tag)            ❷
        except AttributeError:                                 ❸
            try:
                method = getattr(self, 'do_' + tag)           ❹
            except AttributeError:
                self.unknown_starttag(tag, attrs)             ❺
                return −1
            else:
                self.handle_starttag(tag, method, attrs)      ❻
                return 0
        else:
            self.stack.append(tag)
            self.handle_starttag(tag, method, attrs)
            return 1                                           ❼

    def handle_starttag(self, tag, method, attrs):
        method(attrs)                                          ❽
```

❶ A ce niveau, `SGMLParser` a déjà trouvé une balise ouvrante et lu la liste d attributs. La seule chose restant à faire est de trouver s il existe un méthode spécifique pour cette balise ou si il faut la traiter avec la méthode par défaut (`unknown_starttag`).

❷ La "magie" de `SGMLParser` n est rien de plus que notre vieille connaissance `getattr`. Ce que vous n aviez peut−être pas réalisé auparavant, c est que `getattr` peut trouver des méthodes définies dans les descendants d un objet aussi bien que dans l objet lui−même. Ici, l objet est `self`, l instance de la méthode qui l appelle. Donc si `tag` est `'pre'`, cet appel à `getattr` cherchera une méthode `start_pre` dans l instance, qui est une instance de la classe `Dialectizer`.

❸ `getattr` déclenche une exception `AttributeError` si la méthode qu il cherche n existe pas dans l objet (ni dans aucun de ses descendants), mais cela n est pas un problème puisque nous avons encadré l appel à `getattr` dans un bloc `try...except` et explicitement intercepté l exception `AttributeError`.

❹ Comme nous n avons pas trouvé de méthode `start_xxx`, nous cherchons également une méthode `do_xxx` avant d abandonner. Cette désignation alternative est généralement utilisée pour les balises isolées, telles que `<br>`, qui n ont pas de balise fermante correspondante. Vous pouvez utiliser l une comme l autre, comme vous le voyez `SGMLParser` essaie les deux pour chaque balise (vous ne devez pas définir à la fois une méthode `start_xxx` et une méthode `do_xxx` pour la même balise, seule la méthode `start_xxx` serait appelée).

❺ Encore une exception `AttributeError`, ce qui veut dire que l appel à `getattr` a échoué pour `do_xxx`. Comme nous n avons trouvé ni un méthode `start_xxx` ni une méthode `do_xxx` pour cette balise, nous interceptons l exception et nous rabattons sur la méthode par défaut, `unknown_starttag`.

❻ Rappelez−vous que les blocs `try...except` peuvent avoir une clause `else`, qui est appelée si aucune exception n est déclenchée au cours du bloc `try...except`.

Logiquement, cela signifie que nous *avons* trouvé une méthode `do_xxx` pour la balise, donc nous allons l appeler.

❼ Au fait, ne vous inquiétez pas pour ces valeurs de retour différentes. En théorie elles signifient quelque chose mais elles ne sont jamais réellement utilisées. Ne vous inquiétez pas non plus de `self.stack.append(tag)`, `SGMLParser` vérifie trace en interne si vos balises ouvrantes correspondent à des balises fermantes, mais il ne fait rien non plus de cette information. En théorie, vous pourriez utiliser ce module pour vérifier que vos balises sont équilibrée, mais cela n en vaut sans doute pas la peine et cela dépasse le cadre de ce chapitre. Nous avons des choses bien plus importantes auxquelles penser maintenant.

❽ Les méthodes `start_xxx` et `do_xxx` ne sont pas appelées directement. La balise, la méthode et les attributs sont passés à cette fonction, `handle_starttag`, de manière à ce que des classes dérivées puissent la redéfinir pour changer la manière dont *toutes* les balises ouvrantes sont traitées. Nous n avons pas besoin d un tel niveau de contrôle, donc nous laissons simplement cette méthode faire ce qu elle doit faire, c est à dire appeler la méthode (`start_xxx` ou `do_xxx`) avec la liste des attributs. Rappelez−vous que `method` est une fonction, retournée par `getattr` et que les fonctions sont des objets (je sais que vous en avez assez de l entendre et je promets d arrêter de le dire dès que nous aurons cessé de trouver de nouvelles manières de l utiliser à notre avantage). Ici, l objet fonction est passé à cette méthode d appel en argument et cette méthode appelle la fonction. Arrivés là, nous n avons pas à savoir quelle est la fonction, quel est son nom ni l endroit où elle a été définie. La seule chose que nous devons savoir est qu elle est appelée avec un argument, `attrs`.

Revenons à nos moutons : `Dialectizer`. Nous l avons laissé au moment de définir des méthodes spéciales pour le traitement des balises `<pre>` et `</pre>`. Il n y a plus qu une chose à faire et c est de traiter les blocs de texte avec nos substitutions prédéfinies. Pour cela nous devons redéfinir la méthode `handle_data`.

Exemple 8.19. Redéfinition de la méthode `handle_data`

```
    def handle_data(self, text):                                            ❶
        self.pieces.append(self.verbatim and text or self.process(text)) ❷
```

❶ `handle_data` est appelée avec un seul argument, le texte à traiter.

❷ Dans la classe parente `BaseHTMLProcessor`, la méthode `handle_data` ne fait qu ajouter le texte au tampon de sortie, `self.pieces`. Ici, la logique n est qu un petit peu plus compliquée. Si nous sommes au milieu d un bloc `<pre>...</pre>`, `self.verbatim` sera une valeur quelconque supérieure à `0`, nous voulons alors ajouter le texte au tampon de sortie sans modification. Sinon, nous appellerons une méthode spécifique pour appliquer les substitutions, puis ajouterons le résultat de ce traitement dans le tampon de sortie. En Python, cela se fait en une ligne, en utilisant l astuce `and−or`.

Nous sommes près de comprendre complètement `Dialectizer`. Le seul chaînon manquant concerne la nature des substitutions de texte elle−mêmes. Si vous connaissez un peu de Perl, vous savez que lorsque des substitutions de texte complexes sont nécessaires, la seule vrai solution est d utiliser les expressions régulières. Les classes suivantes de `dialect.py` définissent une série d'expressions régulières qui opèrent sur le texte entre les balises HTML. Mais nous venons d'avoir un chapitre entier sur les expressions régulières. Je pense que nous en avons assez appris pour un chapitre.

8.9. Assembler les pièces

Il est temps d utiliser tout ce que nous avons appris. J espère que vous avez été attentif.

Exemple 8.20. La fonction `translate`, première partie

```
def translate(url, dialectName="chef"): ❶
    import urllib                       ❷
    sock = urllib.urlopen(url)          ❸
    htmlSource = sock.read()
    sock.close()
```

❶ La fonction `translate` a un argument optionnel `dialectName`, qui est une chaîne spécifiant le dialecte que nous allons utiliser. Nous allons voir comment il est employé dans une minute.

❷ Attendez une seconde, il y a une instruction `import` dans cette fonction ! C est parfaitement légal en Python. Vous êtes habitué à voir des instructions `import` au début d un programme, ce qui signifie que le module importé est disponible n importe où dans le programme. Mais vous pouvez également importer des modules dans une fonction, ce qui signifie que le module importé n est disponible qu à l intérieur de cette fonction. Si un module n est utilisé que dans une seule fonction, c est un bon moyen de rendre votre code plus modulaire (quand vous vous rendrez compte que votre bidouille du week−end est devenue une oeuvre respectable de 800 lignes et que vous déciderez de la segmenter en une dizaine de modules réutilisables, vous apprécierez cette possibilité).

❸ Ici, nous obtenons le code source de l URL passée en paramètre.

Exemple 8.21. La fonction `translate`, deuxième partie : de bizarre en étrange

```
    parserName = "%sDialectizer" % dialectName.capitalize() ❶
    parserClass = globals()[parserName]                     ❷
    parser = parserClass()                                  ❸
```

❶ `capitalize` est une méthode de chaîne que nous n avons pas encore vue. Elle met simplement en majuscule la première lettre d une chaîne et met le reste en minuscules. En la combinant à un formatage de chaîne, nous avons pris le nom d un dialecte et l avons transformé en un nom de classe Dialectizer lui correspondant. Si `dialectName` est la chaîne `'chef'`, `parserName` sera la chaîne `'ChefDialectizer'`.

❷ Nous avons le nom d une classe sous forme de chaîne (`parserName`) et l espace de noms sous forme de dictionnaire (`globals`()). En les combinant, nous pouvons obtenir une référence à la classe désignée par la chaîne (rappelez−vous que les classes sont des objets et peuvent être assignés à des variables comme n importe quel autre objet). Si `parserName` est la chaîne `'ChefDialectizer'`, `parserClass` sera la classe `ChefDialectizer`.

❸ Maintenant nous avons un objet de classe (`parserClass`) et nous voulons une instance de la classe. Nous savons déjà comment on fait ça, en appelant la classe comme une fonction. Le fait que la classe soit référencée par une variable locale ne fait absolument aucune différence, nous appelons simplement la variable locale comme une fonction et obtenons une instance de la classe. Si `parserClass` est la classe `ChefDialectizer`, `parser` sera une instance de la classe `ChefDialectizer`.

Pourquoi un tel effort ? Après tout, il n y a que 3 classes `Dialectizer`, pourquoi ne pas utiliser simplement une instruction `case` (il n y a pas de `case` en Python, mais nous pourrions utiliser une série d instructions `if`) ? Pour une seule raison, l extensibilité. La fonction `translate` n a absolument aucune idée du nombre de classes Dialectizer que nous avons défini. Imaginez que nous définissions une nouvelle classe `FooDialectizer` demain, `translate` continuerait de fonctionner en recevant `'foo'` en paramètre `dialectName`.

Encore mieux, imaginez que nous mettions `FooDialectizer` dans un module séparé et que nous l importions par `from` *`module`* `import`. Nous avons déjà vu que cela l ajoute à `globals`(), donc `translate` fonctionnerait toujours sans modification, même si `FooDialectizer` était dans un autre fichier.

Maintenant, imaginez que le nom du dialecte provienne de l extérieur du programme, par exemple d une base de données ou d une valeur entrée par un utilisateur dans un formulaire. Vous pouvez utiliser n importe quelle architecture Python côté serveur pour générer dynamiquement des pages Web, cette fonction pourrait prendre une URL et un nom de dialecte (les deux sous la forme de chaîne) dans la chaîne d une requête de page Web et renvoyer

la page Web "traduite".

Finalement, imaginez un *framework* `Dialectizer` avec une architecture de *plug−ins*. Vous pourriez mettre chaque classe `Dialectizer` dans un fichier séparé, laissant uniquement la fonction `translate` dans `dialect.py`. Avec un modèle de nommage uniforme, la fonction `translate` pourrait importer dynamiquement la classe appropriée du fichier approprié, uniquement à partir du nom de dialecte (vous n avez pas encore vu d'importation dynamique, mais je promet de la traiter dans un prochain chapitre). Pour ajouter un nouveau dialecte, vous ajouteriez simplement un nouveau fichier correctement nommé dans le répertoire des *plug−ins* (par exemple `foodialect.py` contenant la classe `FooDialectizer`). Appeler la fonction `translate` avec le nom du dialecte `'foo'` ferait charger le module `foodialect.py`, importer la classe `FooDialectizer` et lancer la traduction.

Exemple 8.22. La fonction `translate`, troisième partie

```
    parser.feed(htmlSource) ❶
    parser.close()          ❷
    return parser.output()  ❸
```

❶ Après tout ce que je vous ai demandé d imaginer, cela va sembler plutôt ennuyeux, mais la fonction `feed` est responsable de toute la transformation. Nous avons l ensemble du source HTML rassemblé en une seule chaîne, donc nous n avons à appeler `feed` qu une seule fois. Cependant, vous pouvez appeler `feed` autant de fois que vous le voulez et le *parser* continuera son travail. Si vous vous inquiétez de l utilisation mémoire (ou si vous savez que vous aurez à traiter de très grandes pages HTML), vous pouvez écrire une boucle dans laquelle vous lisez quelques lignes de HTML et les passez au *parser*. Le résultat serait le même.

❷ Comme `feed` gère un tampon interne, vous devez toujours appelez la méthode `close` du *parser* lorsque vous avez terminé (même si vous lui avez passé la totalité en une seule fois comme nous venons de le faire). Dans le cas contraire vous risquez de vous apercevoir que votre sortie est tronquée.

❸ Rappelez−vous que `output` est la fonction que nous avons définie dans `BaseHTMLProcessor` qui assemble toutes les pièces de sortie que nous avons stockées en tampon et les retourne sous forme d une chaîne unique.

Et rien qu en faisant ça, nous avons "traduit" une page Web, rien qu à partir d une URL et d un nom de dialecte.

Pour en savoir plus

- Vous croyiez que je plaisantais quand je parlais de traitement côté serveur. C est ce que je pensais aussi, jusqu à ce que je trouve ce "traducteur" en ligne (http://rinkworks.com/dialect/). Malheureusement, le code source n a pas l air d être disponible.

8.10. Résumé

Python vous fournit un outil puissant, `sgmllib.py`, pour manipuler du code HTML en transformant sa structure en modèle objet. Vous pouvez utiliser cet outil de nombreuses manières.

- analyser le code HTML en cherchant quelque chose de précis
- assembler les résultats, comme le fait URL lister
- modifier la structure à la volée, comme le fait attribute quoter
- transformer le code HTML en quelque chose d autre en manipulant le texte sans toucher aux balises, comme le fait `Dialectizer`

En plus de ces exemples, vous devriez vous sentir à l aise pour :

- Utiliser `locals`() et `globals`() pour accéder aux espaces de noms

- Formater des chaînes à l aide d un dictionnaire

---

[3] Le terme technique pour un analyseur comme `SGMLParser` est *consommateur*: il consomme du HTML est le décompose. On peut penser que le nom de `feed` (nourrir) a été choisi pour cadrer avec ce modèle du "consommateur". Personnellement, ça me fait penser à une cage sombre dans un zoo, sans arbres ni plantes ni trace de vie d aucune sorte, mais où vous pouvez deviner, si vous vous tenez tout à fait immobile, deux yeux en vrilles qui vous regardent en retour dans le coin du fond à gauche, mais vous arrivez à vous convaincre que c est votre esprit qui vous joue des tours et la seule chose qui vous permet de dire que ce n est pas une cage vide est une petite pancarte sur la rambarde sur laquelle est écrit "Ne pas nourrir l analyseur." Mais peut−être que j ai trop d imagination. De toute manière, c est une image mentale intéressante.

[4] La raison pour laquelle Python gère mieux les listes que les chaînes est que les listes sont modifiables et que les chaînes sont non−modifiables. Cela signifie qu ajouter à une liste ne fait qu ajouter l élément et mettre à jour l index. Mais comme les chaînes ne peuvent pas être changées après avoir été créées, du code tel que `s = s + newpiece` créera une nouvelle chaîne à partir de la concaténation de l original et du nouvel élément, puis jettera la chaîne originelle. Cela implique une gestion mémoire coûteuse et le coût augmente avec la taille de la chaîne, donc faire `s = s + newpiece` à l intérieur d une boucle est fatal. En termes techniques, ajouter `n` éléments à une liste est `O(n)`, alors qu ajouter `n` éléments à une chaîne items est $O(n^2)$.

[5] Bon, en fait ce n est pas une question si courante. Elle n est pas aussi courante que "Quel éditeur faut−il utiliser pour écrire du code Python ?" (réponse : Emacs) ou "Python est−il meilleur ou moins bon que Perl?" (réponse : "Perl est moins bon que Python parce que les gens voulaient qu il soit moins bon." Larry Wall, 14/10/1998) Mais des questions sur le traitement du HTML apparaissent sous une forme ou l autre à peu près une fois par mois et parmi ces questions celle−ci est fréquente.

Chapitre 9. Traitement de données XML

9.1. Plonger

Les deux prochains chapitres concernent le traitement des données XML en Python. Il serait préférable que vous sachiez préalablement à quoi ressemble un document XML, qu'il est constitué de balises structurées pour former une hiérarchie d'éléments etc. Si cela vous est étranger, lisez d'abord ce tutoriel XML (http://directory.google.com/Top/Computers/Data_Formats/Markup_Languages/XML/Resources/FAQs,_Help,_and_Tutorials/) puis reprenez votre lecture.

Si vous n'êtes pas particulièrement intéressés par XML, vous devriez prendre néanmoins connaissance de ces chapitres qui couvrent des sujets importants comme les paquetages Python, l'Unicode, les arguments de la ligne de commande et la façon d'utiliser `getattr` pour la sélection de méthode.

Nul besoin d'être un grand connaisseur en philosophie, bien que si vous avez eu la malchance d'être confronté aux écrits d'Emmanuel Kant, vous apprécierez davantage le programme qui sert d'exemple que si votre expertise touche à une matière beaucoup plus pragmatique, telle que la programmation.

Il y a fondamentalement deux manières de travailler avec XML. L'une est appelée SAX ("Simple API for XML") et fonctionne en lisant les données XML au fur et à mesure et en appelant une méthode chaque fois qu'un élément est rencontré. (Si vous lisez Chapitre 8, *Traitement du HTML*, cela devrait vous paraître familier, parce que c'est la façon dont le module `sgmllib` fonctionne.) L'autre est appelée DOM ("Document Object Model"), et fonctionne en lisant le document XML dans son entier et en en créant une représentation interne au moyen de classes Python natives reliées dans une structure arborescente. Python dispose de modules standards pour chacun de ces deux traitements, mais ce chapitre ne concernera que l'utilisation du DOM.

Ce qui suit est un programme complet en Python qui génère en sortie un résultat pseudo−aléatoire basé sur une grammaire hors contexte (*context−freei*) définie dans un format XML. Ne vous inquiétez pas pour l'instant de n'y rien comprendre; vous examinerez plus en détail à la fois les données en entrée et en sortie du programme tout au long de ce chapitre et du chapitre à venir.

Exemple 9.1. `kgp.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Kant Generator for Python

Generates mock philosophy based on a context−free grammar

Usage: python kgp.py [options] [source]

Options:
  −g ..., −−grammar=...   use specified grammar file or URL
  −h, −−help              show this help
  −d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py −g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate
"""
```

```
from xml.dom import minidom
import random
import toolbox
import sys
import getopt

_debug = 0

class NoSourceError(Exception): pass

class KantGenerator:
    """generates mock philosophy based on a context-free grammar"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        """load XML input source, return parsed XML document

        - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
        - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        """guess default source of the current grammar

        The default source will be one of the <ref>s that is not
        cross-referenced.  This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
```

```
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

    def refresh(self):
        """reset output buffer, re-parse entire source file, and return output

        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar file
        each time.
        """
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        """output generated text"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
        """choose a random child element of a node

        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        """parse a single XML node

        A parsed XML document (from minidom.parse) is a tree of nodes
        of various types.  Each node is represented by an instance of the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The following
        statement constructs the name of a class method based on the type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        """parse the document node

        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)

    def parse_Text(self, node):
        """parse a text node

        The text of a text node is usually added to the output buffer
        verbatim.  The one exception is that <p class='sentence'> sets
        a flag to capitalize the first letter of the next word.  If
        that flag is set, we capitalize the text and reset the flag.
```

```
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Element(self, node):
        """parse an element

        An XML element corresponds to an actual tag in the source:
        <xref id='...'>, <p chance='...'>, <choice>, etc.
        Each element type is handled in its own method.  Like we did in
        parse(), we construct a method name based on the name of the
        element ("do_xref" for an <xref> tag, etc.) and
        call the method.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)

    def parse_Comment(self, node):
        """parse a comment

        The grammar can contain XML comments, but we ignore them
        """
        pass

    def do_xref(self, node):
        """handle <xref id='...'> tag

        An <xref id='...'> tag is a cross-reference to a <ref id='...'>
        tag.  <xref id='sentence'/> evaluates to a randomly chosen child of
        <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))

    def do_p(self, node):
        """handle <p> tag

        The <p> tag is the core of the grammar.  It can contain almost
        anything: freeform text, <choice> tags, <xref> tags, even other
        <p> tags.  If a "class='sentence'" attribute is found, a flag
        is set and the next word will be capitalized.  If a "chance='X'"
        attribute is found, there is an X% chance that the tag will be
        evaluated (and therefore a (100-X)% chance that it will be
        completely ignored)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
            doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
```

```
        """handle <choice> tag

        A <choice> tag contains one or more <p> tags.  One <p> tag
        is chosen at random and evaluated; the rest are ignored.
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)

    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

Exemple 9.2. `toolbox.py`

```
"""Miscellaneous utility functions"""

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
```

```
        return source

    if source == '-':
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))
```

Lancez le programme `kgp.py`, qui va alors analyser la grammaire au format XML fournie par défaut dans le fichier `kant.xml`, puis afficher une prose philosophique dans le style d'Emmanuel Kant.

Exemple 9.3. Exemple de sortie de `kgp.py`

```
[you@localhost kgp]$ python kgp.py
     As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never-ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.
```

```
[...snip...]
```

Il s'agit, bien entendu, d'un complet charabia. Et bien, pas tout à fait. Le propos est syntaxiquement et grammaticalement correct (bien que très verbeux −− Kant n'est pas connu pour son style laconique). Quelques propositions peuvent se trouver "vraies" (ou du moins correspondre à l'esprit de la pensée kantienne), d'autres sont d'une fausseté flagrante et la majeure partie du propos demeure incohérente. Mais l'ensemble reste dans le style d'Emmanuel Kant.

Inutile de préciser à nouveau que mieux vaut avoir un solide passé de philosophe pour apprécier cette forme d'humour.

L'intérêt de ce programme n'a aucun rapport avec une quelconque connaissance de la philosophie kantienne. Tout le contenu retourné par l'exemple précédent provient du fichier de grammaire, `kant.xml`. Si un autre fichier de grammaire est spécifié à la ligne de commande, le résultat sera complètement différent.

Exemple 9.4. Résultat plus simple à la sortie de `kgp.py`

```
[you@localhost kgp]$ python kgp.py −g binary.xml
00101001
[you@localhost kgp]$ python kgp.py −g binary.xml
10110100
```

Vous approfondirez la structure du fichier de grammaire plus loin dans ce chapitre. Pour le moment, il vous suffit de savoir que ce fichier définit la structure du résultat en sortie et que le programme `kgp.py` parcourt cette grammaire et choisit aléatoirement des mots et leur emplacement.

9.2. Les paquetages

Analyser un document XML est pour l'heure chose très simple : cela tient sur une ligne de code. Cependant, avant que vous n'abordiez cette ligne de code, une digression s'impose pour parler des paquetages.

Exemple 9.5. Charger un document XML (bref aperçu)

```
>>> from xml.dom import minidom ❶
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

❶ Vous n'aviez pas encore vu cette syntaxe. Elle ressemble presque à votre cher `from` *`module`* `import`, mais le `"."` montre que c'est quelque chose de plus qu'un simple import. En fait, `xml` est bien le paquetage, `dom` un paquetage imbriqué dans `xml` et `minidom` un module de `xml.dom`.

Cela paraît compliqué, mais il n'en est rien. Un examen de l'implémentation réelle peut aider. Les paquetages sont un peu plus que des répertoires de modules; les paquetages imbriqués sont les sous−répertoires. Les modules au sein d'un paquetage (ou d'un paquetage imbriqué) sont juste des fichiers `.py`, comme toujours, sauf qu'ils sont rangés dans un sous−répertoire plutôt que dans le répertoire principal `lib/` de votre installation Python.

Exemple 9.6. La disposition des fichiers dans un paquetage

```
Python21/           root Python installation (home of the executable)
|
+−−lib/             library directory (home of the standard library modules)
   |
   +−− xml/         xml package (really just a directory with other stuff in it)
       |
```

```
+--sax/      xml.sax package (again, just a directory)
|
+--dom/      xml.dom package (contains minidom.py)
|
+--parsers/  xml.parsers package (used internally)
```

Ainsi, lorsque vous écrivez `from xml.dom import minidom`, Python comprend la chose suivante : "rechercher dans le répertoire `xml` un répertoire `dom`, rechercher *dans ce répertoire* le module `minidom` et l'importer en tant que `minidom`". Mais Python est beaucoup plus astucieux que cela; non seulement il vous est possible d'importer la totalité des modules d'un paquetage, mais il est encore possible de sélectionner des classes ou des fonctions spécifiques à l'intérieur d'un module. Il vous est également possible d'importer le paquetage lui−même comme un module. La syntaxe est toujours la même; Python comprend ce à quoi vous faites référence à partir de la disposition des fichiers dans le paquetage, et s'exécute automatiquement.

Exemple 9.7. Les paquetages sont aussi des modules

```
>>> from xml.dom import minidom          ❶
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element  ❷
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom                  ❸
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml                           ❹
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

❶ Ici vous importez un module (`minidom`) à partir d'un paquetage imbriqué (`xml.dom`). Il en résulte que `minidom` est importé dans votre espace de nom et que pour référencer les classes du module `minidom` (comme `Element`), vous devez les faire précéder du nom du module.

❷ Ici vous importez la classe (`Element`) d'un module (`minidom`) à partir d'un paquetage imbriqué (`xml.dom`). Il en résulte que `Element` est importé directement dans votre espace de noms. Remarquez que cela n'interfère aucunement avec la précédente importation, la classe `Element` peut désormais être référencée de deux façons (mais il s'agit toujours de la même classe).

❸ Ici vous importez le paquetage `dom` (un paquetage imbriqué de `xml`) en tant que module. Chaque niveau d'un paquetage peut être considéré comme un module, comme vous le verrez plus loin. Il peut même posséder ses propres attributs et méthodes, comme les modules que vous avez vus précédemment.

❹ Ici vous importez le paquetage racine `xml` comme un module.

Mais alors, comment un paquetage (c'est−à−dire un répertoire) peut−il être importé et traité comme un module (c'est−à−dire un fichier) ? Par la magie du fichier `__init__.py`. Les paquetages ne sont pas de simples répertoires; ce sont des répertoires qui contiennent un fichier spécifique, `__init__.py`. Ce fichier définit les attributs et les méthodes du paquetage. Par exemple, `xml.dom` contient une classe `Node`, laquelle est définie dans `xml/dom/__init__.py`. Lorsque vous importez un paquetage en tant que module (comme `dom` à partir de `xml`), vous importez aussi son fichier `__init__.py`.

Un paquetage est un répertoire pourvu du fichier `__init__.py`. Le fichier `__init__.py` définit les attributs et

les méthodes du paquetage. Il n'est cependant pas tenu de définir quoi que ce soit; ce peut simplement être un fichier vide, mais il se doit d'être présent. Et si `__init__.py` n'existe pas, le répertoire reste un répertoire, pas un paquetage et ne peut ni être importé ni contenir de modules ou de paquetages imbriqués.

Pourquoi s'embêter avec des paquetages ? C'est qu'ils permettent de regrouper logiquement des modules associés. Plutôt que d'avoir un paquetage `xml` contenant les paquetages `sax` et `dom`, les auteurs auraient pu décider de ranger toutes les fonctionnalités de `sax` dans le fichier `xmlsax.py` et de la même façon les fonctionnalités de `dom` dans `xmldom.py`, voire même de n'en faire qu'un seul module. Mais cela aurait été peu maniable (à ce jour, le paquetage XML contient plus de 3000 lignes de code) et difficile à gérer (des fichiers sources séparés permettent à plusieurs personnes de travailler simultanément à différents endroits du code).

Si vous vous retrouvez à écrire un long sous−système en Python (ou, plus probablement, quand vous vous rendrez compte que votre petit sous−système s'est énormément développé), prenez le temps de construire une solide architecture de paquetage. C'est là encore l'un des nombreux avantages de Python, alors profitez−en.

9.3. Analyser un document XML

Comme je le disais, analyser un document XML est chose très simple qui tient en une ligne de code. A vous de décider de la suite.

Exemple 9.8. Charger un document XML (version longue)

```
>>> from xml.dom import minidom                                          ❶
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml') ❷
>>> xmldoc                                                               ❸
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()                                                 ❹
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ Comme vous l'avez vu dans la section précédente, cela importe le module `minidom` à partir du paquetage `xml.dom`.

❷ Voici la ligne de code qui fait tout le travail : `minidom.parse` prend un argument et retourne une représentation analysée du document XML. Un argument peut recouvrir beaucoup de choses; dans ce cas, il s'agit du nom de fichier d'un document XML sur le disque local. (afin de poursuivre, il vous faudra modifier le chemin pour pointer vers votre répertoire d'exemples.) Mais vous pouvez aussi lui passer un objet−fichier, ou même un pseudo objet−fichier. Vous tirerez avantage de cette souplesse plus loin dans ce chapitre.

❸ L'objet retourné par `minidom.parse` est un objet `Document`, un descendant de la classe `Node`. Cet objet `Document` est la racine d'une structure pseudo−arborescente d'objets Python en relation qui représente complètement le document XML passé à `minidom.parse`.

❹ `toxml` est une méthode de la classe `Node` (et donc disponible à partir de l'objet `Document` que vous obtenez de `minidom.parse`). `toxml` affiche les données XML représentées par ce `Node`. Pour le noeud `Document`, il affiche le document XML en entier.

Maintenant que vous avez un document XML en mémoire, vous pouvez commencer à le parcourir.

Exemple 9.9. Obtenir les noeuds enfants

```
>>> xmldoc.childNodes     ❶
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0] ❷
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild    ❸
<DOM Element: grammar at 17538908>
```

❶ Chaque `Node` possède un attribut `childNodes`, qui est une liste d'objets `Node`. Un `Document` n'a jamais qu'un noeud enfant : l'élément racine du document XML (dans ce cas, l'élément `grammar`).

❷ Pour obtenir le premier (et, dans ce cas, le seul) noeud enfant, utilisez simplement la syntaxe normale des listes. Rappelez−vous qu'il n'y a rien de particulier à ce sujet; il ne s'agit que d'une liste d'objets Python tout à fait normaux.

❸ Puisqu'obtenir le premier noeud enfant d'un noeud parent est une action utile et courante, la classe `Node` possède un attribut `firstChild`, lequel est synonyme de `childNodes[0]`. (Il existe également un attribut `lastChild`, qui correspond à `childNodes[−1]`.)

Exemple 9.10. `toxml` fonctionne pour tout noeud

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml() ❶
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ Puisque la méthode `toxml` est définie dans la classe `Node`, elle vaut pour tout noeud XML et pas seulement pour l'élément `Document`.

Exemple 9.11. Les noeuds enfants peuvent être de type texte

```
>>> grammarNode.childNodes                    ❶
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml()      ❷


>>> print grammarNode.childNodes[1].toxml() ❸
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() ❹
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()      ❺
```

❶ A l'examen des éléments XML de `binary.xml`, vous pourriez penser que `grammar` a seulement deux noeuds enfants, les deux éléments `ref`. Mais vous oubliez quelque chose : les retours chariot ! Après `'<grammar>'` et avant le premier `'<ref>'` se trouve un retour chariot et ce texte compte comme un noeud enfant de l'élément `grammar`. De la même façon, il y a un retour charriot après chaque `'</ref>'`; il faut également les compter comme des noeuds enfants. Aussi `grammar.childNodes` correspond en réalité à une liste de 5 objets: 3 objets `Text` et 2 objets `Element`.

❷ Le premier enfant est un objet `Text` qui représente le retour chariot après la balise `'<grammar>'` et avant la première balise `'<ref>'`.

❸ Le deuxième enfant est un objet `Element` représentant le premier élément `ref`.

❹ Le quatrième enfant est un objet `Element` représentant le second élément `ref`.

❺ Le dernier enfant est un objet `Text` représentant le retour chariot après la dernière balise `'</ref>'` et avant la dernière balise `'</grammar>'`.

Exemple 9.12. Tracer une route jusqu'au texte

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] ❶
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes                  ❷
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()                 ❸
<p>0</p>
>>> pNode.firstChild                    ❹
<DOM Text node "0">
>>> pNode.firstChild.data               ❺
u'0'
```

❶ Comme vous l'aviez vu dans l'exemple précédent, le premier élément `ref` correspond à `grammarNode.childNodes[1]`, puisque childNodes[0] est un noeud `Text` qui a pour valeur le retour chariot.

❷ L'élément `ref` possède son propre ensemble de noeuds enfants : un pour le retour chariot, un autre pour les espaces, un troisième pour l'élément `p` et ainsi de suite.

❸ Vous pouvez même utiliser ici la méthode `toxml`, profondément imbriqué dans le document.

❹ L'élément `p` possède un seul noeud enfant (vous ne pouvez pas dire cela de cet exemple, mais regardez le `pNode.childNodes` si vous en me croyez pas) et il s'agit du noeud `Text` représentant le caractère `'0'`.

❺ L'attribut `.data` du noeud `Text` vous donne la chaîne de caractères représentée par le noeud texte. Mais que signifie le `'u'` qui précède cette chaîne ? La réponse mérite d'être développée dans une section à part entière.

9.4. Le standard Unicode

Le standard Unicode est un mécanisme pour représenter les caractères de tous les différents langages à travers le monde. Quand Python analyse un document XML, toutes les données sont stockées en mémoire au format Unicode.

Vous y reviendrez dans une minute après un bref rappel historique.

Remarque historique. Avant l'Unicode, il y avait des systèmes d'encodage de caractères propres à chaque langage, chacun utilisant les mêmes positions (0−255) pour représenter son jeu de caractères. Certains langages (comme le Russe) disposaient de multiples standards divergents sur le manière de représenter les mêmes caractères; d'autres langages (comme le japonais) avaient tant de caractères qu'ils nécessitaient de recourir à de multiples jeu de caractères. L'échange de documents entre systèmes était difficile parce qu'il n'y avait aucun moyen pour une machine de dire avec certitude quel schéma d'encodage avait utilisé l'auteur d'un document; la machine ne voyait que des nombres et ces nombres pouvaient avoir plusieurs significations. Imaginez alors d'enregistrer ces documents au même endroit (comme dans la même table d'une base de données); vous auriez besoin d'enregistrer le jeu d'encodage avec chaque partie du texte et de vous assurer de le transmettre en même temps que le texte. Imaginez alors à quoi ressembleraient des documents multilingues rassemblant les caractères issus de différents langages. (Typiquement, ils utiliseraient des codes d'échappement pour passer d'un mode à l'autre; et hop, vous utilisez le mode russe koi8−r, et le caractère 241 signifie ceci; et hop, vous êtes maintenant dans un mode grec, et le caractère 241 signifie cela. Et ainsi de suite.) C'est pour résoudre ce problème qu'Unicode a été conçu.

Pour résoudre ces problèmes, Unicode représente chaque caractère comme un nombre codé sur 2 octets, de 0 à 65535.[6] Chaque nombre représente un unique caractère utilisé au moins dans l'un des langages existant. (Les caractères présents dans de multiples langages ont le même code numérique.) Il y a exactement un nombre par caractère et un caractère par nombre. Un caractère Unicode n'est jamais ambigu.

Bien sûr, il n'en demeure pas moins le problème de tous ces systèmes d'encodage antiques. L'ASCII 7−bit, par exemple, qui enregistre les caractères anglais dans une fourchette de 0 à 127. (65 représente le "A" majuscule, 97 le "a" minuscule et ainsi de suite.) L'anglais dispose d'un alphabet très simple et il peut être complètement exprimé en ASCII 7 bits. Les langages d'Europe de l'Ouest comme le français, l'espagnol, et l'allemand utilisent tous un système d'encodage appelé ISO−8859−1 (appelé encore "latin−1"), qui reprend les caractères de l'ASCII 7 bits pour les positions de 0 à 127, puis étend son espace de positions de 128 et 255 pour les caractères comme le 'n coiffé d'un tilde' (241), et le 'u surmonté de deux points' (252). Unicode utilise les mêmes caractères que l'ASCII 7 bits de 0 à 127, les mêmes caractères que l'ISO−8859−1 de 128 à 255, puis étend son espace de positions de 256 à 65535 pour accueillir les caractères des autres langues.

Quand vous vous occupez de données Unicode, vous pouvez avoir ponctuellement besoin de rétroconvertir ces données dans un système d'encodage archaïque. Par exemple, pour les intégrer dans un autre système informatique qui s'attend à recevoir des données dans un modèle d'encodage spécifique sur un octet, ou pour les afficher sur une console ou une imprimante qui ne gère pas l'Unicode. Ou encore, pour les stocker dans un document XML qui précise explicitement un modèle d'encodage différent.

Et sur cette remarque, retournons à Python.

Le langage Python supporte Unicode depuis la version 2.0. Le paquetage XML utilise Unicode pour mémoriser toutes les données XML analysées, mais vous pouvez utiliser Unicode partout.

Exemple 9.13. Introduction à Unicode

```
>>> s = u'Dive in'          ❶
>>> s
u'Dive in'
>>> print s                 ❷
Dive in
```

❶ Pour créer une chaîne de caractères Unicode plutôt qu'une chaîne ASCII, ajoutez la lettre "u" devant la chaîne. Notez que cette chaîne particulière ne possède aucun caractère non−ASCII. Parfait; Unicode est

un sur−ensemble d'ASCII (un immense sur−ensemble en vérité), ainsi toute chaîne de caractères ASCII peut être aussi stockée en Unicode.

❷ Lorsque l'on affiche une chaîne de caractères, Python essaie de la convertir dans l'encodage par défaut, généralement l'ASCII. (Nous y venons dans une minute.) Puisque cette chaîne Unicode est constituée de caractères qui sont également des caractères ASCII, leur affichage produit dans les deux cas le même résultat; la conversion est uniforme et si vous ne saviez pas que `s` était une chaîne de caractères Unicode, vous ne remarquez pas la différence.

Exemple 9.14. Mémoriser des caractères non−ASCII

```
>>> s = u'La Pe\xf1a'          ❶
>>> print s                    ❷
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin-1') ❸
La Peña
```

❶ Le véritable avantage d'Unicode est, bien entendu, sa capacité à mémoriser des caractères non−ASCII, comme "ñ" espagnol (`n` surmonté d'un tilde). Le caractère Unicode pour n tilde est `U+0xf1` en hexadécimal (241 en décimal), que vous pouvez écrire de cette façon : `\xf1`.

❷ Vous rappelez−vous que je disais que la fonction `print` essaie de convertir une chaîne de caractères Unicode en ASCII afin de pouvoir l'afficher ? Et bien, cela ne marche pas ici parce que notre chaîne Unicode contient des caractères non−ASCII et Python déclenche une erreur `UnicodeError`.

❸ Voici comment s'effectue la conversion d'Unicode vers d'autres modèles d'encodage. `s` est une chaîne de caractères Unicode, mais `print` peut seulement afficher une chaîne de caractères normale. Pour résoudre ce problème, vous appelez la méthode `encode`, disponible pour toute chaîne Unicode, afin d'effectuer la conversion de l'Unicode vers le modèle d'encodage passé en paramètre. Dans ce cas, vous utilisez `latin−1` (aussi connu sous le nom de `iso−8859−1`), qui inclut le n tilde (tandis que l'encodage ASCII par défaut ne le prend pas en charge, étant donné qu'il inclut seulement les caractères positionnés de 0 à 127).

Vous rappelez−vous que je disais que Python convertit d'habitude l'Unicode en ASCII toutes les fois qu'une chaîne Unicode a besoin d'être transformée en une chaîne normale ? Et bien, ce modèle d'encodage par défaut est une option qui peut être personnalisée.

Exemple 9.15. `sitecustomize.py`

```
# sitecustomize.py                    ❶
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site-packages/
import sys
sys.setdefaultencoding('iso-8859-1') ❷
```

❶ `sitecustomize.py` est un script particulier; Python essaie de le charger au démarrage, si bien qu'il est lancé automatiquement. Comme il est indiqué dans le commentaire, il peut se trouver n'importe où (à condition que `import` puisse le retrouver), mais il est placé généralement dans le répertoire `site−packages` au sein du répertoire Python `lib`.

❷ La fonction `setdefaultencoding` déclare un encodage par défaut. Python tâchera d'utiliser ce modèle d'encodage quand il aura besoin de transformer automatiquement une chaîne Unicode en une chaîne normale.

Exemple 9.16. Les effets du paramétrage de l'encodage par défaut

```
>>> import sys
>>> sys.getdefaultencoding() ❶
'iso-8859-1'
>>> s = u'La Pe\xf1a'
>>> print s                  ❷
La Peña
```

❶ Cette exemple suppose que vous ayez effectué les changements indiqués dans l'exemple précédent concernant le fichier `sitecustomize.py` et que vous ayez redémarré Python. si l'encodage par défaut signale encore `'ascii'`, vous n'avez pas paramétré correctement `sitecustomize.py`, ou bien Python n'a pas été redémarré. L'encodage par défaut peut seulement être changé au démarrage de Python; vous ne pouvez pas le modifier ultérieurement. (En raison de quelques excentricités de programmation que je ne détaillerai pas maintenant, vous ne pouvez plus appeler `sys.setdefaultencoding` après que Python ait démarré. Pour creuser la question, voyez `site.py` et recherchez "`setdefaultencoding`".)

❷ Maintenant que le modèle d'encodage par défaut inclut tous les caractères que vous utilisez dans votre chaîne, Python n'a aucun problème pour la convertir automatiquement et l'afficher.

Exemple 9.17. Spécifier l'encodage des fichiers `.py`

Si vous stockez des chaînes non−ASCII dans votre code Python, vous aurez besoin de spécifier l'encodage pour chaque fichier `.py` en déclarant l'encodage en tête de chaque fichier. Cette déclaration définit le fichier `.py` au format UTF−8 :

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

Qu'en est−il maintenant des documents XML ? Et bien, chaque document XML possède un encodage spécifique. De nouveau, ISO−8859−1 est un encodage courant pour traiter les données des langages de l'Europe de l'Ouest. C'est également le cas de KOI8−R pour les documents en langue russe. Lorsque l'encodage est spécifié, il apparaît dans l'en−tête du document XML.

Exemple 9.18. `russiansample.xml`

```
<?xml version="1.0" encoding="koi8-r"?>      ❶
<preface>
<title> @548A;>285</title>                   ❷
</preface>
```

❶ Cet exemple est extrait d'un véritable document XML rédigé en russe; c'est une partie de la traduction de ce livre−ci. Remarquez l'encodage, `koi8-r`, spécifié dans l'en−tête.

❷ Ces caractères cyrilliques composent, à ma connaissance, le mot russe qui signifie "Preface". Si vous ouvrez ce fichier dans un éditeur de texte classique, vous obtiendrez vraisemblablement une chaîne de caractères incompréhensible, parce que ces caractères sont encodés en `koi8-r`, mais sont affichés en `iso-8859-1`.

Exemple 9.19. Analyser `russiansample.xml`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml') ❶
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title                                       ❷
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435'
>>> print title                                 ❸
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
```

```
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r')     ❹
>>> convertedtitle
'\xf0\xd2\xc5\xc4\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle                        ❺
 @548A;>285
```

❶ Je suppose que vous avez sauvegardé l'exemple précédent en tant que `russiansample.xml` dans le répertoire courant. Je suppose également, pour ne rien oublier, que vous êtes revenu à un encodage en `'ascii'` par défaut, en retirant le fichier `sitecustomize.py`, ou en commentant la ligne `setdefaultencoding`.

❷ Notez que les données texte de la balise `title` (maintenant dans la variable `title`, grâce à cette longue concaténation de fonctions Python dont je ne dirai mot jusqu'à la prochaine section, au risque de vous agacer) −− les données texte de l'élément `title` du document XML sont mémorisées en Unicode.

❸ Il n'est pas possible d'afficher le titre parce que cette chaîne Unicode contient des caractères non−ASCII et Python ne peut le convertir en ASCII car cela n'a aucun sens.

❹ Vous pouvez cependant les convertir explicitement en `koi8-r` et dans ce cas vous obtenez une chaîne (normale, et non Unicode) de caractères codés sur un octet (`f0`, `d2`, `c5`, et ainsi de suite) qui sont les versions `koi8-r` des caractères de la chaîne Unicode d'origine.

❺ L'affichage de la chaîne encodée en `koi8-r` ne produira probablement qu'un charabia sur votre écran, parce que votre IDE Python interprète ces caractères en `iso-8859-1`, non en `koi8-r`. Mais ils sont au moins affichés. (Et , si vous regardez attentivement, il s'agit du même charabia que lorsque vous aviez ouvert le document XML original dans un éditeur de texte non−Unicode. Python le convertit de `koi8-r` en Unicode lorsqu'il analyse le document XML et vous l'avez juste rétroconvertis.)

Pour résumer, Unicode peut paraître intimidant si vous n'y avez jamais eu à faire auparavant, mais les données en Unicode sont très faciles à manipuler avec Python. Si vos documents XML sont tous codés en ASCII 7−bit (comme les exemples de ce chapitre), vous ne vous soucierez jamais d'Unicode. Python convertira les données ASCII des documents XML en Unicode au moment du traitement et les restituera automatiquement en ASCII au besoin, sans que vous ne le remarquiez jamais. Mais s'il devient nécessaire de les gérer dans d'autres langages, Python répond présent à l'appel.

Lectures complémentaires

- Unicode.org (http://www.unicode.org/) est le site officiel du standard Unicode et propose une brève introduction technique (http://www.unicode.org/standard/principles.html).
- Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) contient beaucoup plus d'exemples sur la façon d'utiliser les fonctions Unicode de Python, y compris la manière de forcer Python à contraindre l'Unicode en ASCII, même s'il regimbe.
- PEP 263 (http://www.python.org/peps/pep−0263.html) approfondit la manière de définir un jeu d'encodage dans vos fichiers `.py`.

9.5. Rechercher des éléments

Parcourir des documents XML en s'arrêtant à chaque noeud peut être fastidieux. Si vous recherchez un noeud particulier, enfoui au plus profond de votre document XML, il existe un raccourci pour le retrouver rapidement : `getElementsByTagName`.

Pour cette section, vous utiliserez le fichier de grammaire `binary.xml` qui se présente comme suit :

Exemple 9.20. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

Il a deux `ref`s, `'bit'` et `'byte'`. Un `bit` contient soit un `'0'` soit un `'1'` et un `byte` vaut 8 `bit`s.

Exemple 9.21. Introduction à `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') ❶
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

❶ `getElementsByTagName` prend pour argument le nom de l'élément à rechercher. Il retourne une liste d'objets `Element`, correspondant aux éléments XML qui portent ce nom. Dans ce cas, vous trouvez deux éléments `ref`.

Exemple 9.22. Tout élément peut être recherché

```
>>> firstref = reflist[0]                          ❶
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p") ❷
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()                         ❸
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

❶ Toujours avec le même exemple, le premier objet de votre `reflist` est l'élément `ref 'bit'`.

❷ Vous pouvez utiliser la même méthode `getElementsByTagName` sur cet `Element` pour

trouver tous les éléments <p> à l'intérieur de l'élément `ref 'bit'`.

❸ Comme précédemment, la méthode `getElementsByTagName` retourne une liste de tous les éléments trouvés. Dans ce cas, il y en a deux, un pour chaque bit.

Exemple 9.23. La recherche est en réalité récursive

```
>>> plist = xmldoc.getElementsByTagName("p") ❶
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM Element: p at 136146124>]
>>> plist[0].toxml()                          ❷
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()                          ❸
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

❶ Soyez attentif à la différence entre cet exemple et le précédent. Précédemment, vous aviez cherché les éléments `p` contenus dans `firstref`, mais ici vous recherchez les éléments `p` de `xmldoc`, l'objet racine qui représente le document XML complet. Cette instruction *retrouve* les éléments `p` contenus dans les éléments `ref` de l'élément racine `grammar`.

❷ Les deux premiers éléments `p` sont contenus dans le premier élément `ref` (`ref 'bit'`).

❸ Le dernier élément `p` se trouve dans le second élément `ref` (`ref 'byte'`).

9.6. Accéder aux attributs d'un élément

Les éléments XML peuvent avoir un ou plusieurs attributs et il est très facile d'y accéder une fois le document XML analysé.

Pour cette section, vous utiliserez le même fichier de grammaire `binary.xml` que dans la section précédente.

Cette section peut paraître un peu confuse dans le mesure où des terminologies se recouvrent. Les éléments d'un document XML ont des attributs, mais les objets Python ont aussi des attributs. Lorsque vous analysez un document XML, vous obtenez un paquet d'objets Python qui représentent l'ensemble des parties du document XML et certains de ces objets Python représentent les attributs des éléments XML. Mais les objets (Python) qui représentent les attributs (XML) possèdent également des attributs (Python), qui sont utilisés pour accéder à diverses parties de l'attribut (XML) que l'objet représente. Je vous avais bien dit que tout cela prêtait à confusion. Je suis ouvert à toute suggestion qui permettrait de les distinguer plus clairement.

Exemple 9.24. Accéder aux attributs d'un élément

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes                ❶
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()         ❷ ❸
[u'id']
>>> bitref.attributes.values()       ❹
```

```
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]    ❺
<xml.dom.minidom.Attr instance at 0x81d5044>
```

❶ Chaque objet `Element` a un attribut appelé `attributes`, qui est un objet `NamedNodeMap`. Pas de panique : un objet `NamedNodeMap` joue le même rôle qu'un dictionnaire, objet dont vous connaissez déjà l'usage.

❷ En Considérant `NamedNodeMap` comme un dictionnaire, vous pouvez obtenir une liste des noms des attributs de cet élément au moyen de `attributes.keys()`. Cet élément a seulement un attribut, `'id'`.

❸ Les noms d'attributs, comme tout autre texte d'un document XML, sont emmagasinés en Unicode.

❹ En considérant de nouveau `NamedNodeMap` comme un dictionnaire, vous pouvez obtenir une liste des valeurs des attributs au moyen de `attributes.values()`. Les valeurs sont elles−même des objets de type `Attr`. Vous verrez comment obtenir de précieuses informations à partir de cet objet dans l'exemple suivant.

❺ En considérant toujours `NamedNodeMap` comme un dictionnaire, vous pouvez accéder à un attribut individuel par son nom, en ayant recours à la syntaxe courante d'un dictionnaire. (Les lecteurs extrêmement attentifs savent déjà comment la classe `NamedNodeMap` accomplit ce remarquable tour : en définissant une méthode spéciale `__getitem__`. Que les autres se rassurent, ils n'ont pas besoin d'en connaître le fonctionnemment pour l'utiliser efficacement.)

Exemple 9.25. Accéder aux attributs individuels

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name    ❶
u'id'
>>> a.value    ❷
u'bit'
```

❶ L'objet `Attr` représente complétement l'attribut XML unique d'un élément XML unique. Le nom de l'attribut (le nom déjà utilisé pour trouver cet objet dans le pseudo−dictionnaire `NamedNodeMap` `bitref.attributes`) est rangé dans `a.name`.

❷ La valeur courante du texte de cet attribut XML est stockée dans `a.value`.

A l'instar d'un dictionnaire, les attributs d'un élément XML ne sont pas ordonnés. Il se peut que les attributs *apparaissent* dans un certain ordre dans le document XML original et qu'ils *apparaissent* dans un certain ordre quand le document XML est analysé en objets Python, mais ces ordonnancements sont arbitraires et n'ont aucune signification particulière. Vous devriez toujours accéder aux attributs individuels par leur nom, comme les clés d'un dictionnaire.

9.7. Transition

Voilà, c'est tout pour ce qui concerne XML en tant que tel. Le chapitre suivant reprend les mêmes programmes donnés en exemple, mais en mettant l'accent sur certains aspects qui rendent plus souple leur maniement : l'utilisation des flots de données (*streams*) pour le traitement des données en entrée, l'utilisation de `getattr` pour la sélection de méthode et l'utilisation des drapeaux de ligne de commande pour permettre aux utilisateurs de paramétrer le programme sans intervenir dans le code.

Avant de passer au chapitre suivant, vous devez être à l'aise avec ces différents points :

- Analyser les documents XML au moyen de `minidom`, faire une recherche dans le document analysé et accéder arbitrairement aux attributs et aux enfants d'un élément
- Organiser des bibliothèques complexes en paquetages
- Convertir des chaînes Unicode dans différents jeux d'encodage de caractères

---

[6] Il s'agit *là encore* d'une extrême simplification. Unicode a maintenant été étendu pour tenir compte des textes en chinois ancien, en coréen et en japonais, lesquels sont composés de tant de caractères que le système Unicode sur 2 octets n'aurait pu suffire à tous les représenter. Mais Python ne le supporte pas actuellement et je ne sais pas s'il y a un projet en cours pour l'y ajouter. Vous avez atteint les limites de mon expertise, désolé.

Chapitre 10. Des scripts et des flots de données (streams)

10.1. Extraire les sources de données en entrée

L'une des grandes forces de Python repose sur son principe de liaison dynamique, dont un puissant usage est le *pseudo objet−fichier* (*file−like objecti*).

De nombreuses fonctions qui nécessitent une source de données en entrée pourraient simplement prendre un nom de fichier, ouvrir le fichier en lecture, le lire et le fermer après lecture. Mais elles ne le font pas. A la place, elles utilisent un *pseudo objet−fichier*.

Dans les cas les plus simples, un *pseudo objet−fichier* est tout objet pourvu d'une méthode `read` accompagnée d'un paramètre `size` optionnel et qui retourne une chaîne. Quand elle est appelée sans le paramètre `size`, elle lit l'ensemble du contenu de la source en entrée et retourne l'ensemble des données comme une seule chaîne. Lorsqu'elle est appelée avec le paramètre `size`, elle ne parcourt que la longueur indiquée et retourne les données correspondantes; Lorsqu'elle est de nouveau appelée, elle poursuit sa lecture là où elle s'était interrompue, et renvoie le paquet de données suivant.

Vous aviez vu comment la lecture de véritables fichiers fonctionne; La différence tient à ce que vous n'êtes pas restreints à utiliser de réels fichiers. La source en entrée peut être n'importe quoi : un fichier sur le disque, une page web, voire une chaîne codée en dur. Tant que vous passez un pseudo objet−fichier à la fonction et qu'elle appelle simplement la méthode `read` de cet objet, la fonction peut manipuler toute sorte de source en entrée sans avoir besoin de recourir à un code spécifique pour chacune.

Au cas où vous vous demanderiez en quoi cela concerne le traitement des données XML, `minidom.parse` est justement une fonction qui peut recevoir ce type d'objet.

Exemple 10.1. Analyser un document XML à partir d'un fichier

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')     ❶
>>> xmldoc = minidom.parse(fsock) ❷
>>> fsock.close()                  ❸
>>> print xmldoc.toxml()           ❹
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

❶ D'abord, vous ouvrez le fichier sur le disque. Vous obtenez alors un objet−fichier.

❷ Vous passez l'objet−fichier à `minidom.parse`, qui appelle la méthode `read` de `fsock` et lit le document XML à partir du fichier sur le disque.

❸ Assurez−vous d'appeler la méthode `close` de l'objet−fichier, une fois la lecture terminée.

`minidom.parse` ne s'en charge pas.

❹ Appeler la méthode `toxml()` du document XML retourné affiche la totalité de son contenu.

Et bien, tout cela ressemble à une colossale perte de temps. Après tout, vous aviez déjà vu que la fonction `minidom.parse` peut simplement prendre en argument le nom du fichier et effectuer automatiquement les opérations d'ouverture et de fermeture. Et il est vrai que si vous savez que vous devez analyser un fichier local, vous pouvez lui passer le nom du fichier et la fonction `minidom.parse` est suffisamment intelligente pour avoir le bon réflexe (Do The Right Thing(tm)). Mais remarquez maintenant combien l'analyse d'un document XML en provenance d'Internet est semblable −− et tout aussi aisée.

Exemple 10.2. Analyser XML à partir d'un URL

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') ❶
>>> xmldoc = minidom.parse(usock)                              ❷
>>> usock.close()                                              ❸
>>> print xmldoc.toxml()                                       ❹
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...snip...]
```

❶ Comme vous l'avez vu au chapitre précédent, `urlopen` prend l'URL d'une page web et retourne un pseudo objet−fichier. De plus, cet objet dispose d'une méthode `read` qui retourne la source HTML d'une page web.

❷ Maintenant vous passez le pseudo objet−fichier à la fonction `minidom.parse`, qui, très obéissante, appelle la méthode `read` de l'objet et analyse les données XML retournées par cette méthode. Le fait que ces données XML proviennent directement d'une page web n'a aucune pertinence. La fonction `minidom.parse` ne sait pas ce qu'est une page web et ne s'en soucie guère; elle ne connaît que les pseudo objet−fichiers.

❸ Dès que vous en avez terminé, assurez−vous de fermer le pseudo objet−fichier fourni par `urlopen`.

❹ Signalons au passage qu'il s'agit là d'un URL qui propose un véritable contenu XML. C'est la version XML des titres à la une du site Slashdot (http://slashdot.org/), un site consacré aux nouveautés et aux potins de l'actualité technologique .

Exemple 10.3. Analyser XML à partir d'une chaîne (voie facile mais rigide)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) ❶
>>> print xmldoc.toxml()
```

```
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ `minidom` possède une méthode, `parseString`, qui récupère un document XML entier sous forme de chaîne et l'analyse. Vous pouvez l'utiliser à la place de `minidom.parse` si vous savez que votre document XML est sous la forme d'une chaîne.

D'accord, vous pouvez ainsi utiliser la fonction `minidom.parse` pour analyser à la fois des fichiers locaux et des URLs distantes, mais pour analyser des chaînes, vous utilisez... une fonction différente. Cela signifie que si vous voulez être capable de recevoir en entrée un fichier, un URL, ou une chaîne, vous avez besoin de mettre en place une logique particulière pour contrôler s'il s'agit d'une chaîne et appeler le cas échéant la fonction `parseString`. Quelle déception !

S'il y avait un moyen de transformer une chaîne en un pseudo objet−fichier, vous pourriez alors simplement passer cet objet à `minidom.parse`. En fait, un module spécifiquement conçu à cet effet existe : il s'agit de `StringIO`.

Exemple 10.4. Introduction à `StringIO`

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)   ❶
>>> ssock.read()                          ❷
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()                          ❸
''
>>> ssock.seek(0)                         ❹
>>> ssock.read(15)                        ❺
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()                         ❻
```

❶ Le module `StringIO` ne contient qu'une seule classe qui a pour nom `StringIO`, laquelle vous permet de transformer une chaîne en un pseudo objet−fichier. La classe `StringIO` prend la chaîne en paramètre au moment de créer une instance.

❷ Vous avez désormais un pseudo objet−fichier et vous pouvez le manipuler comme s'il s'agissait d'un fichier. En utilisant, par exemple, la méthode `read`, qui retourne la chaîne originale.

❸ Appeler `read` une seconde fois retourne une chaîne vide. Les véritables objets−fichier fonctionnent également de cette façon; une fois que la totalité du fichier est lue, vous ne pouvez lire rien de plus à moins de revenir explicitement au début du fichier. L'objet `StringIO` fonctionne pareillement.

❹ Vous pouvez revenir explicitement au début de la chaîne de la même façon que pour un fichier, en utilisant la méthode `seek` de l'objet `StringIO`.

❺ Vous pouvez aussi lire la chaîne par morceaux, en passant un paramètre `size` à la méthode `read`.

❻ A tout moment, `read` retournera le reste de la chaîne qui n'a pas encore été lu. Le fonctionnement est exactement le même que pour les objets−fichier; d'où le terme *pseudo objet−fichier*.

Exemple 10.5. Analyser XML à partir d'une chaîne (la voie du pseudo objet−fichier)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) ❶
>>> ssock.close()
>>> print xmldoc.toxml()
```

```
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ Désormais vous pouvez passer le pseudo objet−fichier (une instance de `StringIO`) à `minidom.parse`, qui appelle la méthode `read` de l'objet et l'analyse en retour sans se soucier du fait qu'il s'agit en entrée d'une chaîne codée en dur.

Ainsi, vous savez comment utiliser une fonction unique, `minidom.parse`, pour analyser un document XML stocké sur une page web, dans un fichier local, ou dans une chaîne codée en dur. Pour une page web, vous utilisez `urlopen` pour obtenir un pseudo objet−fichier; pour un fichier local, vous utilisez `open`; et pour une chaîne, vous utilisez `StringIO`. Passez maintenant à l'étape suivante et généralisez toutes *ces* différences.

Exemple 10.6. `openAnything`

```
def openAnything(source):                          ❶
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)              ❷
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)                        ❸
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))          ❹
```

❶ La fonction `openAnything` prend un seul argument, `source` et retourne un pseudo objet−fichier. `source` est une chaîne quelconque; ce peut être ou bien un URL (comme `'http://slashdot.org/slashdot.rdf'`), ou bien un chemin d'accès absolu ou relatif à un fichier local (comme `'binary.xml'`), ou encore une chaîne qui contient les données XML à analyser.

❷ Premièrement, vous testez si `source` est un URL. La méthode est brutale : vous essayez de l'ouvrir comme un URL et vous ignorez les erreurs survenues s'il ne s'agit pas d'un URL. Le procédé n'est cependant pas sans élégance dans la mesure où, si `urllib` supporte à l'avenir de nouveaux types d'URLs, ils seront pris en compte sans avoir besoin de reprogrammer.

❸ Si `urllib` se plaint que `source` n'est pas un URL valide, vous supposez alors que c'est le chemin d'un fichier sur le disque et vous essayez de l'ouvrir. De nouveau, rien de très sophistiqué pour tester si `source` est ou non un nom de fichier valide (les règles de validation d'un nom de fichier variant grandement d'un système à l'autre, vous vous égareriez certainement en procédant différemment). A la place, vous ouvrez à l'aveugle le fichier et interceptez silencieusement les erreurs éventuelles.

❹ A ce stade, vous devez supposer que `source` est une chaîne codée en dur (puisque rien d'autre n'a fonctionné), aussi utilisez−vous `StringIO` pour la convertir en un pseudo objet−fichier et le retourner. (En fait, puisque vous utilisez la fonction `str`, `source` n'a pas besoin d'être une chaîne; ce pourrait être un objet quelconque et vous utiliseriez sa représentation sous forme de chaîne, telle qu'elle est définie par la méthode spéciale `__str__`.)

Vous pouvez à présent utiliser la fonction `openAnything` en conjonction avec `minidom.parse` pour écrire une fonction qui prend un argument `source` en référence à un document XML quelconque (un URL, un fichier local, ou encore un document XML sous la forme d'une chaîne codée en dur) et l'analyse.

Exemple 10.7. Utiliser `openAnything` dans le fichier `kgp.py`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

10.2. Entrée, sortie et erreur standard

Les utilisateurs d'UNIX sont déjà familiers avec les concepts d'entrée standard, de sortie standard et d'erreur standard. Cette section s'adresse aux autres.

La sortie standard et l'erreur standard (communément abrégé en `stdout` et en `stderr`) sont des canaux de communication (pipes) intégrés à chaque système UNIX. Lorsque vous affichez (fonction `print`) quelque chose, il est dirigé vers le canal de communication `stdout`; quand votre programme plante et affiche des informations de débogage (comme un traceback en Python), elles sont envoyées vers le canal de communication `stderr`. Chacun de ces deux canaux sont d'ordinaire simplement connectés à la fenêtre du terminal avec laquelle vous travaillez et de cette façon vous voyez s'afficher la sortie du programme ou l'information de débogage s'il plante. (Si vous travaillez sur un système pourvu d'un IDE Python fenêtré, `stdout` et `stderr` sont redirigés par défaut vers la "Fenêtre Interactive".)

Exemple 10.8. Introduction à `stdout` et à `stderr`

```
>>> for i in range(3):
...     print 'Dive in'              ❶
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Dive in') ❷
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in') ❸
Dive inDive inDive in
```

❶ Comme vous l'avez vu dans l'Exemple 6.9, «Compteurs simples», vous pouvez utiliser la fonction prédéfinie de Python `range` pour construire un simple compteur de boucles qui répète une instruction un nombre déterminé de fois.

❷ `stdout` est un pseudo objet−fichier; appeler sa fonction `write` affichera toutes les chaînes que vous lui donnez. En fait, c'est bien ce que fait la fonction `print`; elle ajoute un retour chariot à la fin de la chaîne que vous affichez et appelle `sys.stdout.write`.

❸ Dans le cas le plus simple, `stdout` et `stderr` envoient leur sortie au même endroit : l'IDE Python (si vous en utilisez un), ou la console (si vous avez lancé Python à partir de la ligne de commande). Comme `stdout`, `stderr` n'ajoute pas de retour chariot pour vous; si vous en avez besoin, ajoutez−les vous−même.

`stdout` et `stderr` sont toutes les deux des pseudo objet−fichiers, comme ceux dont il a été question dans la Section 10.1, «Extraire les sources de données en entrée», mais ils sont tous les deux en écriture seule. Ils n'ont pas de méthode `read`, seulement une méthode `write`. Ils n'en restent pas moins des pseudo objet−fichiers auxquels vous pouvez assigner n'importe quel autre fichier − ou pseudo objet−fichier afin d'en rediriger la sortie.

Exemple 10.9. Rediriger la sortie standard

```
[you@localhost kgp]$ python stdout.py
Dive in
[you@localhost kgp]$ cat out.log
This message will be logged instead of displayed
```

(Avec Windows, il faut utiliser `type` au lieu de `cat` pour afficher le contenu d'un fichier.)

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
#stdout.py
import sys

print 'Dive in'                                                ❶
saveout = sys.stdout                                         ❷
fsock = open('out.log', 'w')                                 ❸
sys.stdout = fsock                                           ❹
print 'This message will be logged instead of displayed'     ❺
sys.stdout = saveout                                         ❻
fsock.close()                                                ❼
```

❶ Ce message s'affichera dans la "Fenêtre Interactive" de l'IDE (ou sur la console, si le script est lancé à la ligne de commande).

❷ Sauvegarder toujours `stdout` avant de le rediriger, ainsi vous pourrez revenir à la normale plus tard.

❸ Ouvre un nouveau fichier en écriture. Si le fichier n'existe pas, il sera créé. Si le fichier existe, il sera écrasé

❹ Redirige toutes les sorties supplémentaires dans le fichier que vous venez d'ouvrir.

❺ Ce message "s'affichera" seulement dans le fichier journal; il ne sera pas visible ni dans la fenêtre de l'IDE ni à l'écran.

❻ Restaure `stdout` dans l'état où il était avant que vous n'y mettiez la pagaïe.

❼ Ferme fichier journal.

Rediriger `stderr` fonctionne exactement de la même manière, en utilisant `sys.stderr` au lieu de `sys.stdout`.

Exemple 10.10. Rediriger un message d'erreur

```
[you@localhost kgp]$ python stderr.py
[you@localhost kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
#stderr.py
import sys

fsock = open('error.log', 'w')                ❶
sys.stderr = fsock                            ❷
raise Exception, 'this error will be logged'  ❸ ❹
```

❶ Ouvre le fichier journal où vous voulez enregistrer l'information de débogage.

❷ Redirige l'erreur standard en affectant à `stderr` l'objet−fichier correspondant au fichier journal nouvellement créé.

❸ Déclenche une exception. Notez que sur l'écran de sortie *rien* ne s'affiche. Toute l'information de traceback a été écrite dans `error.log`.

❹ Remarquez également que vous ne fermez pas explicitement votre fichier journal, ni ne restaurez `stderr` dans son état d'origine. Il n'y a pas d'erreur, puisqu'une fois le programme planté (à cause de l'exception), Python nettoiera et fermera le fichier pour nous et cela n'a pas d'importance que `stderr` soit restauré, puisque, comme je l'ai signalé, le programme plante et Python se termine. Un retour à l'état antérieur est plus important pour `stdout`, si vous souhaiter continuer à travailler avec le même script ultérieurement.

Puisqu'il est si trivial d'écrire des messages d'erreurs sur le canal d'erreur standard, il existe une syntaxe abrégée qui peut être utilisée plutôt que de s'embêter à effectuer une redirection complète.

Exemple 10.11. Afficher un message sur `stderr`

```
>>> print 'entering function'
entering function
>>> import sys
>>> print >> sys.stderr, 'entering function' ❶
entering function
```

❶ Cette syntaxe abrégée de l'expression `print` peut être utilisée pour écrire dans tout fichier ou pseudo objet−fichier. Dans cet exemple, vous pouvez rediriger une seule expression `print` vers `stderr` sans affecter les expressions `print` ultérieures.

L'entrée standard, de l'autre côté, est un objet−fichier en lecture seule et représente les données circulant entre un programme et un programme exécuté antérieurement. Cela n'a probablement pas grand sens pour les utilisateurs chevronnés de Mac OS, ou même pour les utilisateurs de Windows à moins que vous ne soyez coutumier de la ligne de commande MS−DOS. Son principe de fonctionnement vous permet de construire une chaîne de commandes sur une seule ligne, de telle sorte que la sortie d'un premier programme devienne l'entrée du programme suivant dans la chaîne. Le premier programme retourne simplement un résultat vers la sortie standard (sans effectuer lui−même une redirection spéciale, sinon le renvoi en sortie de quelques instructions `print` ou que sais−je encore), le programme suivant lit l'entrée standard, et le système d'exploitation se charge de connecter la sortie d'un programme à l'entrée du programme suivant.

Exemple 10.12. Chaîner les commandes

```
[you@localhost kgp]$ python kgp.py −g binary.xml          ❶
01100111
[you@localhost kgp]$ cat binary.xml                       ❷
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "−//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[you@localhost kgp]$ cat binary.xml | python kgp.py −g − ❸ ❹
```

```
10110001
```

❶ Comme vous l'aviez vu dans la Section 9.1, «Plonger», cette commande affiche une chaîne de huit bits aléatoires, `0` ou `1`.

❷ Cette commande affiche simplement la totalité du contenu de `binary.xml`. (les utilisateurs de Windows doivent utiliser `type` au lieu de `cat`.)

❸ Cette commande affiche le contenu de `binary.xml`, mais le caractère "`|`", appelé "pipe", signifie que le contenu ne sera pas affiché à l'écran. A la place, il deviendra l'entrée standard de la prochaine commande qui dans ce cas appelle votre script Python.

❹ Plutôt que de spécifier un module (comme `binary.xml`), vous spécifiez "`−`", ce qui oblige votre script à charger la grammaire à partir de l'entrée standard au lieu d'un fichier particulier sur le disque. (Vous en saurez plus à ce propos dans le prochain exemple.) Ainsi le résultat est le même qu'avec la syntaxe précédente, où vous spécifiiez directement le nom du fichier de grammaire, mais pensez en plus aux nombreuses possibilités qui s'offrent à vous. Plutôt que de simplement exécuter `cat binary.xml`, vous pourriez lancer un premier script qui générerait dynamiquement une grammaire que vous redirigeriez vers votre script. Les données pourraient provenir de n'importe où : une base de données, un méta−script générateur de grammaire, ou que sais−je encore. L'important est que vous n'avez pas besoin de modifier votre script `kgp.py` pour tenir compte de cette fonctionnalité. Tout ce dont vous avez besoin, c'est de pouvoir récupérer le fichier de grammaire à partir de l'entrée standard et alors vous pouvez confier toute la logique restante à un autre programme.

Comment donc notre script "sait"−il qu'il doit lire à partir de l'entrée standard quand le fichier de grammaire correspond à "`−`" ? Cela n'a rien de magique; juste logique.

Exemple 10.13. Lire à partir de l'entrée standard dans `kgp.py`

```
def openAnything(source):
    if source == "−":    ❶
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:

[... snip ...]
```

❶ Il s'agit de la fonction `openAnything` de `toolbox.py`, que vous aviez précédemment examinée dans la Section 10.1, «Extraire les sources de données en entrée». Tout ce que vous avez fait est d'ajouter trois lignes de code au début de cette fonction pour tester si la source correspond à "`−`"; si c'est le cas, vous retournez `sys.stdin`. Rien de plus ! Souvenez−vous que `stdin` est un pseudo objet−fichier pourvu d'une méthode `read`, si bien que le reste du code (dans `kgp.py` où vous appelez `openAnything`) ne change pas d'un pouce.

10.3. Mettre en cache la consultation de noeuds

`kgp.py` recourt à différentes astuces qui peuvent ou non se révéler également utiles dans votre traitement XML. La première tire avantage de la structure logique des documents en entrée pour construire un cache de noeuds.

Un fichier de grammaire définit une série d'éléments `ref`. Chaque élément `ref` contient un ou plusieurs éléments `p`, qui peuvent contenir à leur tour un bon nombre de choses, y compris des éléments `xref`. Chaque fois que vous rencontrez un élément `xref`, vous cherchez un élément `ref` correspondant avec le même attribut `id` et choisissez l'un des enfants de l'élément `ref` pour l'analyser. (Vous verrez comment est effectué ce choix aléatoire dans la prochaine section.)

Voici comment construire la grammaire : définissez des éléments `ref` pour les plus petites parties, puis définissez des éléments `ref` qui "incluent" les premiers éléments `ref` au moyen de `xref` et ainsi de suite. Ensuite vous analysez la référence "la plus large" et suivez chaque `xref`, et au besoin vous récupérez le texte brut associé. Le texte que vous produisez dépend des décisions (aléatoires) que vous faites chaque fois que vous renseignez un élément `xref`, ainsi le résultat est à chaque fois différent.

Tout cela fonctionne d'une manière très souple, mais il y a un revers : la performance. Lorsque vous trouvez un élément `xref` et avez besoin de retrouver l'élément `ref` correspondant, un problème se pose. L'élément `xref` a un attribut `id` et vous désirez trouver l'élément `ref` qui a le même attribut `id`, mais il n'y a pas de moyen simple de le faire. La façon lente de procéder serait de récupérer à chaque fois la liste complète des éléments `ref`, et de les parcourir en recherchant chaque attribut `id`. La manière rapide est de ne faire qu'une fois ce travail en construisant un cache sous la forme d'un dictionnaire.

Exemple 10.14. `loadGrammar`

```
    def loadGrammar(self, grammar):
        self.grammar = self._load(grammar)
        self.refs = {}                                          ❶
        for ref in self.grammar.getElementsByTagName("ref"): ❷
            self.refs[ref.attributes["id"].value] = ref      ❸ ❹
```

❶ Commencez par créer un dictionnaire vide, `self.refs`.

❷ Comme vous l'avez vu dans la Section 9.5, «Rechercher des éléments», `getElementsByTagName` retourne une liste de tous les éléments portant le même nom. Vous pouvez obtenir facilement une liste de tous les éléments `ref`, puis simplement la parcourir.

❸ Comme vous l'avez vu dans la Section 9.6, «Accéder aux attributs d'un élément», vous pouvez accéder aux attributs individuels d'un élément par leur nom, en utilisant la syntaxe d'un dictionnaire standard. Ainsi les clés du dictionnaire `self.refs` seront les valeurs de l'attribut `id` de chaque élément `ref`.

❹ Les valeurs du dictionnaire `self.refs` seront les éléments `ref` eux−mêmes. Comme vous l'avez vu dans la Section 9.3, «Analyser un document XML», chaque élément, chaque noeud, chaque commentaire, chaque fragment de texte d'un document XML après analyse devient un objet.

Une fois le cache construit, il vous suffit simplement de consulter `self.refs` lorsque vous tombez sur un élément `xref` et qu'il vous faut retrouver l'élément `ref` avec l'attribut `id` correspondant.

Exemple 10.15. Utiliser le cache de noeuds `ref`

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Vous explorerez la fonction `randomChildElement` dans la prochaine section.

10.4. Trouver les descendants directs d'un noeud

Une autre technique bien utile lorsqu'il s'agit d'analyser un document XML consiste à retrouver tous les descendants directs d'un élément particulier. Par exemple, dans les fichiers de grammaire, un élément `ref` peut contenir plusieurs éléments `p`, qui à leur tour peuvent contenir un certain nombre de choses, y compris d'autres éléments `p`. Mais vous ne voulez retrouver que les éléments `p` qui sont les enfants d'un élément `ref` et non les éléments `p` qui sont les enfants

d'un autre élément p.

Vous pourriez penser qu'il suffit pour cela d'utiliser simplement la fonction getElementsByTagName, mais il n'en est rien. La fonction getElementsByTagName cherche récursivement et retourne une liste unique de tous les éléments qu'elle trouve. Et puisqu'un élément p peut contenir d'autres éléments p, vous ne pouvez pas utiliser getElementsByTagName, parce qu'elle retournerait alors des éléments p imbriqués dont vous n'avez pas besoin. Retrouver uniquement les descendants directs est une tâche qui vous incombe.

Exemple 10.16. Trouver les descendants directs de type élément

```
    def randomChildElement(self, node):
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE] ❶ ❷ ❸
        chosen = random.choice(choices)             ❹
        return chosen
```

❶ Comme vous l'avez vu dans l'Exemple 9.9, «Obtenir les noeuds enfants», l'attribut childNodes retourne une liste de tous les noeuds enfants d'un élément.

❷ Cependant, comme vous l'avez vu dans l'Exemple 9.11, «Les noeuds enfants peuvent être de type texte», la liste retournée par childNodes contient tous les différents types de noeuds, y compris les noeuds texte. Mais ce n'est pas ce que vous recherchez ici. Vous voulez seulement les enfants qui sont des éléments.

❸ Chaque noeud possède un attribut nodeType, dont la valeur peut être ELEMENT_NODE, TEXT_NODE, COMMENT_NODE, ou une quelque autre valeur. La liste complète des valeurs possibles se trouve dans le fichier __init__.py du paquetage xml.dom. (Voir la Section 9.2, «Les paquetages» pour en savoir plus sur les paquetages.) Mais comme vous n'êtes intéressés que par les noeuds de type élément, vous pouvez filtrer la liste pour ne tenir compte que des noeuds dont le nodeType est ELEMENT_NODE.

❹ Une fois établie la liste des éléments disponibles, il est aisé d'en choisir un au hasard. Python dispose d'un module appelé random qui inclut plusieurs fonctions bien utiles. La fonction random.choice retourne un élément au hasard pris dans une liste d'éléments quelconques. Par exemple, si les éléments ref contiennent plusieurs éléments p, alors choices sera une liste d'éléments p et chosen se verra assigner au final l'un d'entre eux, pris au hasard.

10.5. Créer des gestionnaires distincts pour chaque type de noeud

Une troisième astuce bien utile au traitement XML implique la séparation de votre code en fonctions logiques, sur la base des types de noeud et des noms d'élément. Les documents XML analysés sont constitués de divers types de noeud, chacun représenté par un objet Python. La racine d'un document est elle−même représentée par un objet Document. L'objet Document contient alors un ou plusieurs objets Element (pour les balises XML courantes), dont chacun peut contenir d'autres objets Element, Text (pour les fragments de texte), ou Comment (pour les commentaires imbriqués). Python facilite l'écriture d'un sélecteur pour séparer la logique de chaque type de noeud.

Exemple 10.17. Les noms de classe des objets XML analysés

```
>>> from xml.dom import minidom

>>> xmldoc = minidom.parse('kant.xml') ❶
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__                   ❷
<class xml.dom.minidom.Document at 0x01105D40>
```

```
>>> xmldoc.__class__.__name__                ❸
'Document'
```

❶ Supposez pour le moment que `kant.xml` se trouve dans le répertoire courant.

❷ Comme vous l'avez vu dans la Section 9.2, «Les paquetages», l'objet retourné par l'analyse d'un document XML est un objet `Document`, tel que défini dans le module `minidom.py` du paquetage `xml.dom`. Comme vous l'avez vu dans la Section 5.4, «Instantiation de classes», `__class__` est un attribut prédéfini de chaque objet Python.

❸ De plus, `__name__` est un attribut prédéfini de chaque classe Python et c'est une chaîne. Cette chaîne n'a rien de mystérieux; elle correspond au nom de la classe que vous inscrivez lorsque vous définissez vous−même une classe. (Voir la Section 5.3, «Définition de classes».)

Parfait, vous pouvez désormais obtenir le nom de la classe de n'importe quel noeud XML particulier (puisque chaque noeud XML est représenté par un objet Python). Comment pouvez−vous mettre cet avantage à profit pour séparer la logique de traitement de chaque type de noeud ? La réponse est `getattr`, ce que vous aviez vu précédemment dans la Section 4.4, «Obtenir des références objet avec getattr».

Exemple 10.18. La fonction `parse`, un sélecteur de noeuds XML générique

```
    def parse(self, node):
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) ❶ ❷
        parseMethod(node) ❸
```

❶ Tout d'abord, remarquez que vous construisez une longue chaîne basée sur le nom de la classe du noeud que vous passez à la fonction (dans l'argument `node`). Ainsi, si vous passez un noeud `Document`, vous constituez la chaîne `'parse_Document'` et ainsi de suite.

❷ Maintenant vous pouvez traiter cette chaîne comme un nom de fonction et obtenir une référence de la fonction elle−même en utilisant `getattr`

❸ Enfin, vous pouvez appeler cette fonction et lui passer le noeud comme argument. L'exemple suivant présente les définitions de chacune de ces fonctions.

Exemple 10.19. Les fonctions appelées par le sélecteur de méthodes `parse`

```
    def parse_Document(self, node): ❶
        self.parse(node.documentElement)

    def parse_Text(self, node):     ❷
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Comment(self, node): ❸
        pass

    def parse_Element(self, node): ❹
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

❶ La fonction `parse_Document` n'est jamais appelée qu'une fois, puisqu'il n'y a qu'un unique noeud `Document` dans un document XML et un unique objet `Document` dans la représentation du document XML analysé. Elle ne tient compte que de l'élément racine du fichier de grammaire et l'analyse.

❷ La fonction `parse_Text` est appelée pour les noeuds de type texte. Elle commence par mettre automatiquement en majuscule le premier mot de chaque phrase et ajoute ensuite le texte considéré à une liste.

❸ La fonction `parse_Comment` ne contient que l'instruction `pass`, puisque vous n'avez que faire des commentaires imbriqués dans les fichiers de grammaire. Notez, cependant, que vous avez tout de même besoin de définir cette fonction et, explicitement, de ne lui faire jouer aucun rôle. Si cette fonction n'existait pas, la fonction générique `parse` échouerait sitôt qu'elle rencontrerait un commentaire, faute de pouvoir trouver la fonction `parse_Comment`. Définir une fonction distincte pour chaque type de noeud, même si elle n'est d'aucun usage, permet à la fonction générique `parse` de rester simple et silencieuse.

❹ La méthode `parse_Element` est en réalité elle−même un sélecteur, basé sur le nom de la balise de l'élément. L'idée de départ est la même : retenir le caractère discriminant de chacun de ces éléments (le nom de leur balise) et l'affecter à une fonction distincte. Vous construisez une chaîne comme `'do_xref'` (pour une balise `<xref>`), trouvez la fonction qui porte ce nom et l'appelez. Et ainsi de suite pour chacun des autres noms de balise que vous pourriez trouver au cours de l'analyse d'un fichier de grammaire (les balises `<p>`, les balises `<choice>`).

Dans cet exemple, les fonctions de sélection `parse` et `parse_Element` trouvent simplement les autres méthodes dans la même classe. Si votre traitement est très complexe (ou si vous avez de nombreux noms de balise), vous pourriez diviser votre code en modules séparés et utiliser l'importation dynamique pour importer chaque module et appeler les fonctions dont vous avez besoin. L'importation dynamique sera discuté dans le Chapitre 16, *Programmation fonctionnelle*.

10.6. Manipuler les arguments de la ligne de commande

Python supporte complètement la création de programmes qui peuvent être lancés en ligne de commande, à l'aide d'arguments et de drapeaux longs ou cours pour spécifier diverses options. Cela n'est nullement spécifique à XML, mais comme ce script fait grand usage du traitement en ligne de commande, il est très à propos d'y faire ici mention.

Il est difficile de parler du traitement en ligne de commande sans aborder la façon dont les arguments sont passés au programme Python, commencez donc par un petit programme en guise d'introduction.

Exemple 10.20. Introduction à `sys.argv`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
#argecho.py
import sys

for arg in sys.argv: ❶
    print arg
```

❶ Chaque argument de ligne de commande passé au programme est ajouté à `sys.argv`, qui est un objet liste. Ici le script affiche chaque argument sur une ligne séparée.

Exemple 10.21. Les caractéristiques de `sys.argv`

```
[you@localhost py]$ python argecho.py            ❶
argecho.py
[you@localhost py]$ python argecho.py abc def    ❷
argecho.py
abc
def
[you@localhost py]$ python argecho.py --help     ❸
```

```
argecho.py
--help
[you@localhost py]$ python argecho.py -m kant.xml ❹
argecho.py
-m
kant.xml
```

❶ Ce qu'il faut d'abord retenir de l'objet `sys.argv` est qu'il contient le nom du script que vous appelez. Vous en tirerez avantage plus tard, dans le Chapitre 16, *Programmation fonctionnelle*. Ne vous en souciez pas pour le moment.

❷ Les arguments de la ligne de commande sont séparés par des espaces et chacun se présente comme un élément distinct dans la liste `sys.argv`.

❸ Les drapeaux de la ligne de commande, comme `--help`, se présentent également comme des éléments propres dans la liste `sys.argv`.

❹ Pour corser le tout, certains drapeaux de la ligne de commande prennent eux-mêmes des arguments. Par exemple, vous avez ici un drapeau (`-m`) qui prend un argument (`kant.xml`). Aussi bien le drapeau que son argument sont présentés comme des éléments distincts dans la liste `sys.argv`. Rien n'est fait pour les associer; vous n'obtenez rien de plus qu'une liste.

Comme vous pouvez le voir maintenant, vous disposez indiscutablement de toutes les informations passées à la ligne de commande, mais, de nouveau, il apparaît que tout ne sera pas forcément facile à utiliser. Pour des programme simples qui ne nécessite qu'un seul argument et pas de drapeau, vous pouvez simplement utiliser `sys.argv[1]` pour accéder à l'argument. Aucune honte à avoir; je fais ça tout le temps. Pour des programmes plus complexes, il vous faut recourir au module `getopt`.

Exemple 10.22. Introduction à `getopt`

```
def main(argv):
    grammar = "kant.xml"                    ❶
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="]) ❷
    except getopt.GetoptError:              ❸
        usage()                             ❹
        sys.exit(2)

...

if __name__ == "__main__":
    main(sys.argv[1:])
```

❶ Tout d'abord, regardez au bas de l'exemple et remarquez que vous appelez la fonction `main` avec `sys.argv[1:]`. Rappelez-vous, `sys.argv[0]` est le nom du script en cours; vous n'avez pas à vous en soucier pour le traitement en ligne de commande, aussi le supprimez-vous et envoyez-vous le reste de la liste.

❷ C'est ici que se trouve la partie intéressante du traitement. La fonction `getopt` du module `getopt` prend trois paramètres : la liste des arguments (que vous obtenez à partir de `sys.argv[1:]`), une chaîne contenant tous les drapeaux courts possibles acceptés par le programme et une liste des drapeaux plus longs qui correspondent aux versions courtes. C'est bien confus à première vue, mais l'explication détaillée vient plus bas.

❸ Si un dysfonctionnement survient au moment d'analyser les drapeaux de ligne de commande, `getopt` déclenche une exception, que vous récupérez ensuite. Comme vous avez indiqué à `getopt` tous les drapeaux que vous connaissiez, il y a fort à parier que l'utilisateur final a passé des drapeaux de ligne de commande qui vous sont inconnus.

❹ Comme il est de coutume dans le monde UNIX, quand le script reçoit des drapeaux qu'il ne connaît pas, vous mettez fin au programme de la manière la plus élégante qui soit, en fournissant un résumé des règles de bon usage. Remarquez que je n'ai pas présenté ici la fonction `usage`. Vous aurez encore besoin d'ajouter dans un coin quelques lignes de code pour afficher le résumé approprié; ce n'est pas automatique.

Quels sont donc tous ces paramètres que vous passez à la fonction `getopt` ? Et bien, le premier est simplement la liste brute des arguments et des drapeaux de ligne de commande (à l'exception du premier élément, le nom du script, que vous avez éliminé avant d'appeler la fonction `main`). Le deuxième est la liste des drapeaux courts acceptés par le script.

```
"hg:d"
```

`-h`
: affiche les règles d'usage

`-g ...`
: utilise le fichier de grammaire ou l'URL spécifié

`-d`
: montre l'information de débogage au cours du traitement

Le premier et le troisième drapeaux fonctionnent de manière autonome; vous les spécifiez ou non et le cas échéant ils exécutent une action (affichage l'aide) ou changent un état (actionnement du débogage). Au contraire, le deuxième drapeau (`-g`) *doit* être suivi par un argument, qui est le nom du fichier de grammaire à analyser. En fait, ce peut être un nom de fichier ou une adresse web, et vous ne savez pas encore lequel (vous l'apprendrez plus tard); mais vous êtes certains qu'il doit bien y avoir *quelque chose*. Aussi le signalez-vous à `getopt` en ajoutant deux points après `g`, le second paramètre de la fonction `getopt`.

Pour compliquer davantage les choses, le script accepte soit des drapeaux courts (comme `-h`) soit des drapeaux longs (comme `--help`) et vous voulez que ces derniers effectuent la même chose. D'où l'utilité du troisième paramètre de `getopt` : spécifier une liste de drapeaux longs qui correspondent aux drapeaux courts du second paramètre.

```
["help", "grammar="]
```

`--help`
: affiche les règles d'usage

`--grammar ...`
: utilise le fichier de grammaire ou l'URL spécifié

Trois choses à remarquer ici :

1. Tous les drapeaux longs sont précédés par deux tirets sur la ligne de commande, mais vous n'avez pas besoin d'inclure ces tirets quand vous appelez `getopt`. Ils sont implicites.
2. Le drapeau `--grammar` doit toujours être suivi par un argument additionnel, comme pour le drapeau `-g`. C'est indiqué par un signe égal, `"grammar="`.
3. La liste des drapeaux longs est plus courte que la liste des drapeaux courts, parce que le drapeau `-d` n'a pas de version longue correspondante. Très bien; seul `-d` activera le débogage. Mais l'ordre des drapeaux courts et des drapeaux longs a besoin d'être le même, aussi vous devez d'abord spécifier tous les drapeaux courts qui *possèdent* un drapeau long correspondant, puis tout le reste des drapeaux courts.

Encore confus ? Tournez-vous vers le code en question et voyez si, dans ce contexte, cela fait sens.

Exemple 10.23. Manipuler les arguments de la ligne de commande dans `kgp.py`

```
def main(argv):                          ❶
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:                ❷
        if opt in ("-h", "--help"):      ❸
            usage()
            sys.exit()
        elif opt == '-d':                ❹
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"): ❺
            grammar = arg

    source = "".join(args)               ❻

    k = KantGenerator(grammar, source)
    print k.output()
```

❶ La variable `grammar` gardera une trace du fichier de grammaire utilisé. Vous l'initialisez ici au cas où elle ne serait pas spécifiée à la ligne de commande (en utilisant les drapeaux `-g` ou `--grammar`).

❷ La variable `opts` que vous récupérez à partir de `getopt` contient une liste de tuples : `flag` et `argument`. Si le drapeau ne prend pas d'argument, alors `arg` vaudra simplement `None`. Cela facilite le parcours des drapeaux.

❸ La fonction `getopt` valide les drapeaux de ligne de commande qui sont acceptables, mais elle ne fait aucune sorte de conversion entre les drapeaux courts et les drapeaux longs. Si vous spécifiez le drapeau `-h`, `opt` contiendra `"-h"`; si vous spécifiez le drapeau `--help`, `opt` contiendra `"--help"`. Aussi avez-vous besoin de tester les deux.

❹ Rappelez-vous, le drapeau `-d` n'avait pas de drapeau long correspondant; aussi n'avez-vous besoin que de tester la forme courte. Si vous le détectez, vous déclarez une variable globale à laquelle vous vous référerez plus tard pour afficher les informations de débogage. (Je l'ai utilisé au cours du développement de ce script. Vous ne pensiez tout de même pas que ces exemples ont fonctionné du premier coup ?)

❺ Si vous trouvez un fichier de grammaire, indiqué par les drapeaux `-g` ou `--grammar`, vous conservez l'argument qui le suit (stocké dans `arg`) dans la variable `grammar`, écrasant alors la valeur par défaut que vous aviez initialisée au début de la fonction `main`.

❻ C'est tout. Vous avez parcouru et traité les drapeaux de la ligne de commande. Ce qui signifie qu'il ne peut rester alors que des arguments de ligne de commande. Ils sont retournés par la fonction `getopt` et placés dans la variable `args`. Dans ce cas, vous les traitez comme une source de données pour l'analyseur. Si aucun argument de ligne de commande n'est spécifié, `args` sera une liste vide et `source` contiendra au final une chaîne vide.

10.7. Assembler les pièces

Vous avez déjà parcouru un long chemin. Portez votre regard en arrière et voyez comment rassembler toutes ces étapes.

Pour commencer, il s'agit d'un script qui prend ses arguments sur la ligne de commande, en utilisant le module `getopt`.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
```

```
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

Vous créez une nouvelle instance de la classe `KantGenerator` et vous lui passez un fichier de grammaire et une source de données qui peuvent ou non avoir été spécifiés à la ligne de commande.

```
    k = KantGenerator(grammar, source)
```

L'instance `KantGenerator` charge automatiquement la grammaire, qui est un fichier XML. Vous utilisez la fonction taillée sur−mesure, `openAnything`, pour ouvrir le fichier (qui pourrait être stocké dans un fichier local ou sur un serveur web distant), puis vous utilisez les fonctions de traitement intégrées du module `minidom` pour analyser le document XML en un arbre d'objets Python.

```
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
```

Tiens, au passage, vous tirez avantage de votre connaissance de la structure d'un document XML pour mettre en place un petit cache de références, constitué simplement des éléments du document XML.

```
    def loadGrammar(self, grammar):
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

Si vous avez spécifié une source de données à la ligne de commande, vous l'utilisez; autrement vous décortiquez la grammaire en recherchant la référence de plus haut niveau (celle qui n'est référencée par aucune autre) et vous l'utilisez comme point de départ.

```
    def getDefaultSource(self):
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Maintenant vous décortiquez la source de données. La source est aussi du XML et vous analysez ses noeuds un par un. Pour conserver un code séparé et plus maintenable, vous utilisez des gestionnaires distincts pour chaque type de noeud.

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

Vous parcourez la grammaire, en analysant tous les enfants de chaque élément `p`,

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

en remplaçant les éléments `choice` par un enfant choisi aléatoirement,

```
    def do_choice(self, node):
```

```
        self.parse(self.randomChildElement(node))
```

et en remplaçant les éléments `xref` par l'un des enfants, choisi aléatoirement, de l'élément `ref` correspondant, que vous avez préalablement mis en cache.

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Finalement, vous poussez l'analyse jusqu'au texte brut,

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

qu'il ne vous reste plus qu'à afficher.

```
def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

10.8. Résumé

Python est accompagné de puissantes bibliothèques pour analyser et manipuler des documents XML. Le module `minidom` prend un fichier XML et l'analyse en objets Python, fournissant un accès aléatoire à des éléments arbitraires. Ce chapitre montre encore comment Python peut servir à créer un "véritable" script autonome exécutable en ligne de commande, pourvu de drapeaux et d'arguments de ligne de commande, d'une gestion d'erreur et même de la capacité de récupérer en entrée la redirection du résultat d'un programme antérieur.

Avant de passer au prochain chapitre, prenez le temps de vous familiariser avec les points suivants :

- Chaîner des programmes au moyen de l'entrée standard et de la sortie standard
- Définir des sélecteurs dynamiques avec `getattr`.
- Utiliser les drapeaux de ligne de commande et les valider avec `getopt`

Chapitre 11. Services Web HTTP

11.1. Plonger

Nous avons vu le traitement du HTML et le traitement du XML et, au cours de ces chapitres, comment télécharger une page Web et comment analyser du XML à partir d'une URL. Nous allons maintenant plonger dans le sujet plus général des services Web HTTP.

Pour parler simplement, les services Web HTTP sont une manière d'envoyer et de recevoir des données vers et depuis des serveurs distant par la programmation à l'aide des opérations HTTP. Si nous voulons recevoir des données d'un serveur distant, nous utilisons une simple instruction HTTP GET, si nous voulons envoyer des données au serveur, nous utilisons HTTP POST (certaines API de services Web HTTP plus sophistiquées définissent aussi des manières de modifier ou de supprimer des données avec HTTP PUT et HTTP DELETE). En d'autres termes, les "verbes" du protocole HTTP (GET, POST, PUT et DELETE) correspondent directement à des opérations au niveau de l'application pour recevoir, envoyer, modifier et supprimer des données.

L'avantage principale de cette approche est sa simplicité et cette simplicité a rencontré un succès certain sur de nombreux sites. Les données, en général au format XML, peuvent être construites et stockées de manière statique ou générées dynamiquement par un script côté serveur et tous les principaux langages ont une bibliothèque HTTP pour les télécharger. Le débogage est également plus simple car on peut voir les données brutes à l'aide de n'importe quel navigateur Web. Les navigateur récents affichent même le XML formaté et en couleur pour faciliter sa lecture.

Exemples de services Web XML pur par HTTP :

- L'API d'Amazon (http://www.amazon.com/webservices) vous permet d'obtenir des informations sur les produits du magasin en ligne Amazon.com.
- Le National Weather Service (http://www.nws.noaa.gov/alerts/) (Météo nationale des Etats−Unis) et l'observatoire de Hong Kong (http://demo.xml.weather.gov.hk/) offrent des alertes météo sous forme de services Web.
- L'API Atom (http://atomenabled.org/) pour gérer le contenu sur le Web.
- Les fils de "syndication" (http://syndic8.com/) des weblogs et des sites d'actualité vous apportent les dernières nouvelles sur les sujets les plus divers.

Dans des chapitres suivants, nous explorerons des API qui utilisent HTTP comme moyen de transport pour envoyer et recevoir des données, mais qui ne font pas correspondre la sémantique de l'application et celle du protocole HTTP (elles communiquent tout par un HTTP POST). Mais ce chapitre est centré sur l'utilisation de HTTP GET pour obtenir des données d'un serveur distant et nous verrons plusieurs fonctionnalités du protocole HTTP que nous pouvons utiliser pour obtenir le maximum des services Web HTTP.

Voici une version plus perfectionnée du module `openanything` que nous avons vu au chapitre précédent:

Exemple 11.1. `openanything.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
import urllib2, urlparse, gzip
from StringIO import StringIO

USER_AGENT = 'OpenAnything/1.0 +http://diveintopython.org/http_web_services/'
```

```
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

    def http_error_302(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result

class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):
    def http_error_default(self, req, fp, code, msg, headers):
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code
        return result

def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    '''URL, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    If the etag argument is supplied, it will be used as the value of an
    If-None-Match request header.

    If the lastmodified argument is supplied, it must be a formatted
    date/time string in GMT (as returned in the Last-Modified header of
    a previous request).  The formatted date/time will be used
    as the value of an If-Modified-Since request header.

    If the agent argument is supplied, it will be used as the value of a
    User-Agent request header.
    '''

    if hasattr(source, 'read'):
        return source

    if source == '-':
        return sys.stdin

    if urlparse.urlparse(source)[0] == 'http':
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)
        if etag:
            request.add_header('If-None-Match', etag)
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)
        request.add_header('Accept-encoding', 'gzip')
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler())
        return opener.open(request)

    # try to open with native open function (if source is a filename)
    try:
        return open(source)
    except (IOError, OSError):
```

```
        pass

    # treat source as string
    return StringIO(str(source))

def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)
    result['data'] = f.read()
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')
        if f.headers.get('content-encoding', '') == 'gzip':
            # data came back gzip-compressed, decompress it
            result['data'] = gzip.GzipFile(fileobj=StringIO(result['data'])).read()
    if hasattr(f, 'url'):
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):
        result['status'] = f.status
    f.close()
    return result
```

Pour en savoir plus

- Paul Prescod pense que les services Web HTTP sont le futur de l'Internet (http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html).

11.2. Obtenir des données par HTTP : la mauvaise méthode

Imaginons que nous souhaitons télécharger une ressource par HTTP, par exemple un fil Atom. Seulement, nous ne voulons pas le télécharger une seule fois, nous voulons le télécharger toutes les heures, pour obtenir les dernières nouvelles sur le site qui fournit le fil. Nous allons le faire de la manière la plus simple, puis nous verrons comment faire mieux.

Exemple 11.2. Télécharger un fil de la manière la plus simple

```
>>> import urllib
>>> data = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()    ❶
>>> print data
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
```

❶ Télécharger quoi que ce soit par HTTP est incroyablement simple en Python, en fait cela se fait en une ligne. Le module `urllib` a une fonction `urlopen` qui prend l'adresse de la page que nous voulons télécharger et retourne un objet-fichier que vous pouvez simplement lire par `read()` pour obtenir le contenu de la page. Il n'y a pas plus simple.

Alors, qu'est−ce qui ne va pas dans cette méthode ? Et bien, si c'est un petit test en cours de test ou de développement, il n'y a aucun problème. J'utilise cette méthode tout le temps, je voulais le contenu du fil et c'est ce que j'ai obtenu. La même technique marche pour n'importe quelle page Web. Mais si nous pensons en terme de service Web auquel nous voulons accéder régulièrement (rappelez−vous que nous voulons le relever toutes les heures) alors c'est non seulement inefficace, mais en plus impoli.

Nous allons examiner certaines fonctionnalités de base du protocole HTTP.

11.3. Fonctionnalités de HTTP

Il y a cinq fonctionnalités importantes de HTTP que nous devons supporter dans notre programme.

11.3.1. `User−Agent`

La chaîne d'identification `User−Agent` est simplement un moyen pour le client de déclarer au serveur qui il est lorsqu'il demande une page, un fil ou tout autre service Web par HTTP. Quand le client demande une ressource, il doit toujours annoncer qui il est, de la manière la plus spécifique possible. Cela permet à l'administrateur du serveur de contacter le développeur du client si les choses se passent mal.

Par défaut, Python envoi une chaîne `User−Agent` générique : `Python−urllib/1.15`. Dans la section suivante, nous verrons comment la changer pour quelque chose de plus spécifique.

11.3.2. Les redirections

Parfois, les ressources changent d'emplacement. Les sites Web sont réorganisés, les pages sont déplacées à une nouvelle adresse. Les services Web aussi peuvent être réorganisés. Un fil de syndication à l'adresse `http://example.com/index.xml` peut être déplacé vers `http://example.com/xml/atom.xml`. Un domaine complet peut être déplacé, si une organisation s'élargit et se réorganise. Par exemple, `http://www.example.com/index.xml` peut être redirigé vers `http://server−farm−1.example.com/index.xml`.

A chaque fois que nous demandons une ressource quelle qu'elle soit à un serveur HTTP, le serveur inclut un code de statut dans sa réponse. Le code de statut `200` signifie "tout est normal, voici la page demandée". Le code de statut `404` signifie "page non trouvée" (vous avez sans doute déjà rencontré des erreurs 404 en navigant sur le Web).

Le protocole HTTP a deux manières différentes de signaler qu'une ressources a été déplacée. Le code de statut `302` est une *redirection temporaire*, il signifie "attention, cette page a été déplacée ici temporairement" (il est suivit d'une adresse temporaire dans un en−tête `Location:`). Si nous recevons un code de statut `302`, la spécification HTTP dit que nous devons utiliser la nouvelle adresse pour obtenir la ressource, mais que nous devons réessayer l'ancienne adresse la prochaine fois que nous voulons y accéder. Par contre, si nous recevons un code de statut `301` et une nouvelle adresse, nous devons dorénavant utiliser la nouvelle adresse.

`urllib.urlopen` "suit" automatiquement les redirections lorsqu'il reçoit le code de statut approprié du serveur, mais malheureusement, il ne nous le signale pas. Nous obtenons ainsi les données que nous voulions, mais nous ignorons que la bibliothèque a suivit une redirection "pour vous aider". Donc, nous risquons de continuer d'utiliser l'ancienne adresse et d'être à chaque fois redirigé sur la nouvelle. Cela fait deux aller−retours au lieu d'un seul, ce qui n'est pas très efficace. Nous verrons plus loin dans ce chapitre comment contourner cette difficulté pour pouvoir prendre en compte les redirections permanentes correctement.

11.3.3. Last-Modified/If-Modified-Since

Certaines données changent constamment. La page d'accueil de CNN.com est mise à jour au bout de quelques minutes. Par contre, la page d'accueil de Google.com ne change qu'au bout de plusieurs semaines. Les services Web ne sont pas différents, en général le serveur sait quand les données que nous demandons ont été modifiées pour la dernière fois et HTTP fournit un moyen pour le serveur d'inclure cette date de modification avec les données que nous demandons.

Si nous demandons ces mêmes données une deuxième fois (ou une troisième, une quatrième), nous pouvons dire au serveur la date de dernière modification que nous avons obtenu la dernière fois : nous envoyons un en-tête `If-Modified-Since` avec notre requête avec la date que nous avions obtenue. Si les données n'ont pas été modifiées depuis la dernière fois, le serveur renvoi un code de statut `304`, qui signifie "ces données n'ont pas changé depuis la dernière fois". Qu'est-ce que cela apporte ? Quand le serveur renvoi un code `304`, *il ne renvoi pas les données*. Nous ne recevons que le code de statut. Nous n'avons donc pas besoin de télécharger les mêmes données encore et encore si elles n'ont pas changé, le serveur considère que nous les avons en cache.

Tous les navigateurs récents implémentent la vérification de date de dernière modification. Si nous retournons voir une page qui n'a pas été modifiée de puis votre dernière visite, elle se charge très rapidement. Notre navigateur a mis le contenu de la page dans son cache local à la première visite et à la deuxième il a automatiquement envoyé la date de dernière modification qu'il avait obtenu la première fois. Le serveur a répondu simplement `304: Not Modified`, donc notre navigateur sait qu'il doit recharger la page à partir de son cache. Les services Web peuvent suivre la même procédure.

La bibliothèque URL de Python n'implémente pas la vérification de date de dernière modification, mais comme nous pouvons ajouter les en-têtes que nous voulons à chaque requête et lire les en-têtes que nous voulons à chaque réponse, nous pouvons l'implémenter nous même.

11.3.4. ETag/If-None-Match

Les ETags sont une manière alternative d'accomplir la même chose que la vérification de date de dernière modification : ne pas télécharger des données qui n'ont pas été modifiées. Leur fonctionnement est le suivant : le serveur envoi un code de hachage des données (dans un en-tête `ETag`) avec les données que nous avons demandé. La manière dont ce code est généré est entièrement à la discrétion du serveur. La deuxième fois que nous demandons les données, nous incluons le code de hachage ETag dans un en-tête `If-None-Match:` et si les données n'ont pas été modifiées, le serveur nous renvoi un code de statut `304`. Comme pour la vérification de date de dernière modification, le serveur n'envoi *que* le code `304`, il n'envoi pas les données une nouvelle fois. En incluant le code de hachage ETag dans notre seconde requête, nous disons au serveur qu'il n'est pas nécessaire de renvoyer les mêmes données si elles correspondent à ce code de hachage, puisque nous avons encore les données de la dernière fois.

La bibliothèque URL de Python n'implémente pas les Etags, mais nous verrons plus loin comment les ajouter.

11.3.5. La compression

La dernière fonctionnalité important de HTTP est la compression gzip. Quand on parle de services Web HTTP, il s'agit presque tout le temps d'envoyer et de recevoir du XML. Le XML est du texte et c'est un format verbeux qui se compresse bien. Lorsque nous demandons une ressource par HTTP, nous pouvons demander au serveur, si il a des nouvelles données à nous envoyer, de les envoyer en format compressé. Il suffit d'inclure l'en-tête `Accept-encoding: gzip` dans notre requête et le serveur, si il implémente la compression, nous enverra des données compressées par gzip en les signalant par un en-tête `Content-encoding: gzip`.

La bibliothèque URL de Python n'implémente pas la compression gzip en tant que telle, mais nous pouvons ajouter les en-têtes que nous voulons à la requête. Python fournit un module `gzip` qui a des fonctions nous permettant de

décompresser les données nous même.

Notez que notre petit script d'une ligne pour télécharger un fil de syndication n'implémentait aucune de ces fonctionnalités du protocole HTTP. Voyons comment améliorer cela.

11.4. débogage de services Web HTTP

Pour commencer, nous allons activer les fonctionnalités de débogage de la bibliothèque HTTP de Python pour voir tout ce qui est échangé. Cela nous servira tout au long de ce chapitre, au fur et à mesure que nous rajouterons des fonctionnalités.

Exemple 11.3. débogage de HTTP

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1                    ❶
>>> import urllib
>>> feeddata = urllib.urlopen('http://diveintomark.org/xml/atom.xml').read()
connect: (diveintomark.org, 80)                              ❷
send: '
GET /xml/atom.xml HTTP/1.0                                   ❸
Host: diveintomark.org                                       ❹
User-agent: Python-urllib/1.15                               ❺
'
reply: 'HTTP/1.1 200 OK\r\n'                                 ❻
header: Date: Wed, 14 Apr 2004 22:27:30 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content-Type: application/atom+xml
header: Last-Modified: Wed, 14 Apr 2004 22:14:38 GMT         ❼
header: ETag: "e8284-68e0-4de30f80"                          ❽
header: Accept-Ranges: bytes
header: Content-Length: 26848
header: Connection: close
```

❶ `urllib` utilise une autre bibliothèque standard de Python, `httplib`. Normalement, il n'est pas nécessaire de faire directement un `import httplib` (`urllib` le fait automatiquement), mais nous le faisons ici pour activer le drapeau de débogage de la classe `HTTPConnection` qu'`urllib` utilise en interne pour se connecter au serveur HTTP. C'est une technique extrêmement utile. D'autres bibliothèques de Python ont des drapeaux de déboguage similaires, mais il n'y a pas de standard pour les nommer ou les activer, il faut consulter la documentation de chaque bibliothèque pour voir si une telle option est disponible.

❷ Maintenant que le drapeau de débogage est mis, les informations sur la requête et la réponse HTTP sont affichées en temps réel. La première chose que cela nous dit est que nous nous connectons au serveur `diveintomark.org` sur le port 80, qui est le port standard du protocole HTTP.

❸ Lorsque nous demandons le fil Atom, `urllib` envoi trois lignes au serveur. La première ligne spécifie le verbe HTTP que nous utilisons et le chemin de la ressource (sans le nom de domaine). Toutes les requêtes de ce chapitre utiliseront `GET`, mais dans le chapitre suivant sur SOAP, nous verrons qu'il utilise `POST` pour toutes ses requêtes. La syntaxe de base est la même, indépendamment du verbe.

❹ La deuxième ligne est l'en-tête `Host`, qui spécifie le nom de domaine du service auquel nous accédons. C'est important, parce qu'un serveur HTTP unique peut héberger plusieurs domaines différents. Mon serveur héberge actuellement 12 domaines, d'autres serveurs peuvent en héberger des centaines, voir des milliers.

❺ La troisième ligne est l'en−tête `User−Agent`. Ce qui s'affiche ici est la chaîne `User−Agent` standard que la bibliothèque `urllib` ajoute par défaut. Dans la section suivante, nous verrons comment la modifier pour la rendre plus spécifique.

❻ Le serveur répond par un code de statut et une série d'en−têtes (et peut−être des données, qui ont été stockées dans la variable `feeddata`). Le code de statut est `200`, ce qui signifie "tout est normal, voici les données demandées". Le serveur nous donne également la date à laquelle il a répondu à la requête, des informations sur le serveur lui−même et le type de contenu des données qu'il nous envoi. En fonction de l'application, ces informations peuvent être utile ou non. Ici, nous sommes rassurés, nous avions demandé un fil Atom et le serveur nous renvoi un fil Atom (`application/atom+xml`, qui est le type de contenu déclaré pour les fils Atom).

❼ Le serveur annonce la date de dernière modification de ce fil Atom (dans le cas présent il y a environ 13 minutes). Nous pouvons envoyer cette information au serveur la prochaine fois que nous demandons le même fil pour que le serveur vérifie s'il a été modifié entre temps.

❽ Le serveur annonce que ce fil Atom a un code de hachage ETag de `"e8284−68e0−4de30f80"`. Le code de hachage ne signifie rien par lui−même, nous ne pouvons rien en faire, sauf l'envoyer au serveur lors de notre prochaine requête. Le serveur pourra l'utiliser pour dire si les données ont changé ou non.

11.5. Changer la chaîne `User−Agent`

La première chose à faire pour améliorer notre client de services Web HTTP est de faire en sorte qu'il s'identifie correctement avec une chaîne `User−Agent`. Pour cela, nous devons aller plus loin qu'`urllib` et plonger dans `urllib2`.

Exemple 11.4. Présentation de `urllib2`

```
>>> import httplib
>>> httplib.HTTPConnection.debuglevel = 1                                    ❶
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')        ❷
>>> opener = urllib2.build_opener()                                          ❸
>>> feeddata = opener.open(request).read()                                   ❹
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User−agent: Python−urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:23:12 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content−Type: application/atom+xml
header: Last−Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284−68e0−4de30f80"
header: Accept−Ranges: bytes
header: Content−Length: 26848
header: Connection: close
```

❶ Si vous avez gardé votre IDE Python ouverte avec l'exemple de la section précédente, vous pouvez sauter cette étape, cette instruction active le débogage HTTP pour que nous puissions voir ce qui est exactement envoyé et ce qui est reçu.

❷ Obtenir une ressource HTTP avec `urllib2` se fait en trois étape, pour des raisons qui seront bientôt claires. La première étape est la création d'un objet `Request`, qui prend en paramètre l'URL de la ressource que nous voulons obtenir. Notez que cette étape n'effectue encore aucune requête.

❸ La seconde étape est de construire un `opener` d'URL. Celui−ci peut prendre en paramètre un nombre quelconque de gestionnaires (*handlers*) qui contrôleront la gestion des réponses. Nous pouvons aussi construire un `opener` sans gestionnaire particulier, ce que nous faisons ici. Nous verrons comment définir et utiliser des gestionnaires spécialisés plus loin dans ce ce chapitre lorsque nous traiterons des redirections.

❹ La dernière étape est de dire à l'`opener` d'ouvrir l'URL avec l'objet `Request` que nous avons créé. Comme vous pouvez le voir à l'affichage des informations de débogage, cette étape effectue véritablement la requête et stocke les données obtenues dans `feeddata`.

Exemple 11.5. Ajout d'en−têtes avec l'objet `Request`

```
>>> request                                              ❶
<urllib2.Request instance at 0x00250AA8>
>>> request.get_full_url()
http://diveintomark.org/xml/atom.xml
>>> request.add_header('User−Agent',
...     'OpenAnything/1.0 +http://diveintopython.org/')  ❷
>>> feeddata = opener.open(request).read()               ❸
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User−agent: OpenAnything/1.0 +http://diveintopython.org/   ❹
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Wed, 14 Apr 2004 23:45:17 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Content−Type: application/atom+xml
header: Last−Modified: Wed, 14 Apr 2004 22:14:38 GMT
header: ETag: "e8284−68e0−4de30f80"
header: Accept−Ranges: bytes
header: Content−Length: 26848
header: Connection: close
```

❶ C'est la suite de l'exemple précédent, nous avons déjà créé un objet `Request` avec l'URL à laquelle nous voulons accéder.

❷ En utilisant la méthode `add_header` de l'objet `Request`, nous pouvons ajouter les en−têtes HTTP de notre choix à la requête. Le premier paramètre est l'en−tête, le deuxième est la valeur fournie pour cet en−tête. Par convention, une chaîne `User−Agent` a le format suivant : un nom d'application, suivi d'une barre oblique, suivi d'un numéro de version. Le reste est de forme libre, on en voit de nombreuses variations, mais il doit contenir l'URL de l'application. La chaîne `User−Agent` est en général enregistrée par le serveur dans son journal avec d'autres détails de la requête, inclure une URL de l'application permet aux administrateurs de serveurs de contacter l'auteur en cas de problème.

❸ L'objet `opener` que nous avons créé précédemment peut être réutilisé lui aussi, il ouvrira à nouveau le même fil, mais avec l'en−tête `User−Agent` que nous avons défini.

❹ Voici la chaîne `User−Agent` spécialisée que nous avons définie à la place de la chaîne générique envoyée par défaut par Python. Si vous regardez attentivement, vous verrez que nous avons défini un en−tête `User−Agent`, mais que ce qui a été envoyé est un en−tête `User−agent`. Vous voyez la différence ? `urllib2` a modifié la casse de manière à ce que seule la première lettre soit en majuscule. Cela n'a aucune importance, le protocole HTTP spécifie que les noms de champs d'en−têtes sont insensibles à la casse.

11.6. Prise en charge de `Last−Modified` et `ETag`

Maintenant que nous savons comment ajouter des en−têtes HTTP à nos requêtes de services Web, voyons comment prendre en charge les en−têtes `Last−Modified` et `ETag`.

Ces exemples montrent la sortie avec le mode débogage désactivé. Si il est toujours activé, vous pouvez le désactiver en tapant `httplib.HTTPConnection.debuglevel = 0`. Vous pouvez aussi le laisser activé, si cela vous aide.

Exemple 11.6. Test de `Last-Modified`

```
>>> import urllib2
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener()
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.dict                                    ❶
{'date': 'Thu, 15 Apr 2004 20:42:41 GMT',
 'server': 'Apache/2.0.49 (Debian GNU/Linux)',
 'content-type': 'application/atom+xml',
 'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
 'etag': '"e842a-3e53-55d97640"',
 'content-length': '15955',
 'accept-ranges': 'bytes',
 'connection': 'close'}
>>> request.add_header('If-Modified-Since',
...     firstdatastream.headers.get('Last-Modified'))               ❷
>>> seconddatastream = opener.open(request)                         ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\urllib2.py", line 326, in open
    '_open', req)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 901, in http_open
    return self.do_open(httplib.HTTP, req)
  File "c:\python23\lib\urllib2.py", line 895, in do_open
    return self.parent.error('http', req, fp, code, msg, hdrs)
  File "c:\python23\lib\urllib2.py", line 352, in error
    return self._call_chain(*args)
  File "c:\python23\lib\urllib2.py", line 306, in _call_chain
    result = func(*args)
  File "c:\python23\lib\urllib2.py", line 412, in http_error_default
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
urllib2.HTTPError: HTTP Error 304: Not Modified
```

❶ Vous vous rappelez de tous les en-têtes HTTP qui s'affichaient lorsque nous avions activé le débogage ? Voici la manière d'y accéder par la programmation : `firstdatastream.headers` est un objet qui se comporte comme un dictionnaire et permet d'accéder à chacun des en-têtes retournés par le serveur HTTP.

❷ A la seconde requête, nous ajoutons l'en-tête `If-Modified-Since` avec la date de dernière modification de la première requête. Si les données n'ont pas changé, le serveur devrait retourner un code de status `304`.

❸ Les données n'ont pas changé. Nous pouvons voir dans la trace de pile que `urllib2` déclenche une exception spécifique, `HTTPError`, en réponse au code de statut `304`. C'est assez inhabituel et pas forcément pratique. Après tout, ce n'est pas une erreur, nous avons spécifiquement demandé au serveur de ne pas renvoyer les données si elles n'avaient pas changé, ce qu'il a fait, puisqu'elles n'avaient pas changé. Ce n'est pas une erreur, c'est exactement le résultat que nous recherchions.

La bibliothèque `urllib2` déclenche également une exception `HTTPError` pour des situations que nous considérerions sans doute comme des erreurs, comme le code `404` (page non trouvée). En fait, elle déclenche `HTTPError` pour *n'importe quel* code de statut autre que `200` (OK), `301` (redirection permanente) ou `302`

(redirection temporaire). Il serait plus utile pour notre programme qu'elle capture le code de statut et qu'elle le retourne simplement, sans déclencher d'exception. Pour cela, nous devons définir un gestionnaire d'URL spécialisé.

Exemple 11.7. Définition de gestionnaires d'URL

Ce gestionnaire d'URL spécialisé fait partie de `openanything.py`.

```
class DefaultErrorHandler(urllib2.HTTPDefaultErrorHandler):     ❶
    def http_error_default(self, req, fp, code, msg, headers): ❷
        result = urllib2.HTTPError(
            req.get_full_url(), code, msg, headers, fp)
        result.status = code                                    ❸
        return result
```

❶ La conception d'`urllib2` est centrée sur les gestionnaires d'URL. Chaque gestionnaire est simplement une classe qui peut définir un nombre quelconque de méthodes. Lorsque quelque chose se passe, comme une erreur HTTP ou même un code `304`, `urllib2` recherche par introspection dans la liste des gestionnaires définis une méthode qui puisse le prendre en charge. Nous avons utilisé une technique semblable d'introspection au Chapitre 9, *Traitement de données XML* pour définir des gestionnaires pour différents types de noeuds, mais `urllib2` est plus flexible et recherche par introspection dans tous les gestionnaires définis pour la requête en cours.

❷ `urllib2` recherche parmis les gestionnaires définis et appelle la méthode `http_error_default` lorsqu'il reçoit un code de statut `304` du serveur. En définissant un gestionnaire d'erreur spécialisé, nous pouvons empêcher `urllib2` de déclencher une exception. Nous créons plutôt un objet `HTTPError` et le retournons au lieu de le déclencher.

❸ C'est l'étape−clé : avant de retourner de la fonction, nous sauvegardons le code de statut retourné par le serveur HTTP. Cela permettra d'y accéder facilement à partir du programme appelant.

Exemple 11.8. Utilisation de gestionnaires d'URL spécialisés

```
>>> request.headers                                  ❶
{'If-modified-since': 'Thu, 15 Apr 2004 19:45:21 GMT'}
>>> import openanything
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())          ❷
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                          ❸
304
>>> seconddatastream.read()                          ❹
''
```

❶ Nous continuons l'exemple précédent, donc l'objet `Request` est déjà défini et nous avons déjà ajouté l'en−tête `If-Modified-Since`.

❷ C'est l'étape−clé : maintenant que nous avons défini notre gestionnaire d'URL spécialisé, nous devons dire à `urllib2` de l'utiliser. Vous vous rappelez que j'ai dit qu'`urllib2` décompose l'accès à une ressource en trois étapes et qu'il y avait de bonnes raisons à cela ? Voila pourquoi la construction de l'*opener* d'URL est une étape séparée, pour que nous puissions le construire avec notre propre gestionnaire d'URL redéfinissant le comportement par défaut d'`urllib2`.

❸ Maintenant nous pouvons tranquillement ouvrir la ressource et ce que nous obtenons est un objet qui, en plus des en−têtes habituels (accessibles par `seconddatastream.headers.dict`), contient aussi le code de statut HTTP. Dans ce cas, comme on peut s'y attendre, le code est `304`, ce qui signifie que les données n'ont pas changé depuis la dernière requête.

❹ Notez que lorsque le serveur retourne un code de statut `304`, il ne renvoi pas les données. C'est tout là l'intérêt : préserver de la bande passante en ne retéléchargeant pas ce qui n'a pas été modifié. Nous devons donc mettre ces données en cache la première fois que nous les recevons si nous voulons les utiliser.

La gestion de `ETag` fonctionne de la même manière, mais au lieu de vérifier `Last-Modified` et d'envoyer `If-Modified-Since`, on vérifie `ETag` et on envoi`If-None-Match`. Commençons une nouvelle session dans l'IDE

Exemple 11.9. Prise en charge de `ETag`/`If-None-Match`

```
>>> import urllib2, openanything
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
>>> opener = urllib2.build_opener(
...     openanything.DefaultErrorHandler())
>>> firstdatastream = opener.open(request)
>>> firstdatastream.headers.get('ETag')          ❶
'"e842a-3e53-55d97640"'
>>> firstdata = firstdatastream.read()
>>> print firstdata                              ❷
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> request.add_header('If-None-Match',
...     firstdatastream.headers.get('ETag'))     ❸
>>> seconddatastream = opener.open(request)
>>> seconddatastream.status                      ❹
304
>>> seconddatastream.read()                      ❺
''
```

❶ A l'aide du pseudo-dictionnaire `firstdatastream.headers`, nous pouvons obtenir l'`ETag` retourné par le serveur (si le serveur n'a pas retourné d'`ETag` cette ligne retournera `None`).

❷ Voilà, nous avons les données.

❸ Maintenant, nous préparons le deuxième appel en assignant à l'en-tête `If-None-Match` l'`ETag` obtenu à la première requête.

❹ La deuxième requête réussit silencieusement (sans déclencher d'exception) et nous voyons là encore que le serveur a renvoyé un code de statut `304`. En se basant sur le `ETag` que nous avons envoyé la deuxième fois, il sait que les données n'ont pas changé.

❺ Qu'il soit produit par la vérification de date avec `Last-Modified` ou la correspondance de code de hachage avec `ETag`, les données ne sont jamais renvoyé avec le code de statut `304`. C'est tout l'intérêt.

Dans ces exemples, le serveur HTTP supporte à la fois les en-têtes `Last-Modified` et `ETag`, mais ce n'est pas le cas de tous les serveurs. Pour vos clients de services Web, vous devez prévoir de supporter les deux et programmer de manière défensive au cas ou un serveur ne supporterais que l'un des deux, ou aucun.

11.7. Prise en charge des redirections.

La prise en charge des redirections temporaires et permanentes se fait avec un autre type de gestionnaire d'URL spécialisé.

D'abord, voyons pourquoi un gestionnaire de redirection est nécessaire.

Exemple 11.10. Accéder à des services Web sans gestionnaire de redirection

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1                    ❶
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example301.xml')      ❷
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'                    ❸
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml         ❹
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                                     ❺
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:06:25 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.url                                                      ❻
'http://diveintomark.org/xml/atom.xml'
>>> f.headers.dict
{'content-length': '15955',
'accept-ranges': 'bytes',
'server': 'Apache/2.0.49 (Debian GNU/Linux)',
'last-modified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'connection': 'close',
'etag': '"e842a-3e53-55d97640"',
'date': 'Thu, 15 Apr 2004 22:06:25 GMT',
'content-type': 'application/atom+xml'}
>>> f.status
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: addinfourl instance has no attribute 'status'
```

❶ Pour mieux comprendre ce qu'il se passe, nous activons le débogage.

❷ Voici un URL que j'ai mise en place pour rediriger de manière permanent vers mon fil Atom à l'adresse `http://diveintomark.org/xml/atom.xml`.

❸ Evidemment, lorsque nous essayons de télécharger les données à cette adresse, le serveur renvoi un code de statut `301`, signalant que la ressource a été déplacée de manière permanente.

❹ Le serveur renvoi également un en-tête `Location:` avec la nouvelle adresse de ces données.

❺ `urllib2` remarque le code de redirection et tente automatiquement d'obtenir les données à la nouvelle adresse

spécifiée dans l'en−tête `Location:`.

❻ L'objet obtenu de `opener` contient la nouvelle adresse permanente et tous les en−têtes retourné à la seconde requête (faite sur la nouvelle adresse permanente). Mais le code de statut manque, nous n'avons donc aucun moyen de savoir par la programmation si cette redirection est temporaire ou permanente. Or, cette information est très importante. Si c'est une redirection temporaire, nous devons continuer de demander les données à l'ancienne adresse. Si c'est une redirection permanente, nous devons désormais demander les données à la nouvelle adresse.

C'est loin d'être parfait, mais c'est facile à corriger. `urllib2` ne se comporte pas exactement comme nous le souhaitons dans la gestion des codes `301` et `302`, nous allons donc redéfinir ce comportement. Comment ? Avec un gestionnaire d'URL spécialisé, comme nous l'avons fait pour prendre en charge les codes `304`.

Exemple 11.11. Definition du gestionnaire de redirection

Cette classe est définie dans `openanything.py`.

```
class SmartRedirectHandler(urllib2.HTTPRedirectHandler):     ❶
    def http_error_301(self, req, fp, code, msg, headers):
        result = urllib2.HTTPRedirectHandler.http_error_301( ❷
            self, req, fp, code, msg, headers)
        result.status = code                                 ❸
        return result

    def http_error_302(self, req, fp, code, msg, headers):   ❹
        result = urllib2.HTTPRedirectHandler.http_error_302(
            self, req, fp, code, msg, headers)
        result.status = code
        return result
```

❶ La gestion des redirections est définie dans `urllib2` dans une classe appelée `HTTPRedirectHandler`. Nous ne voulons pas redéfinir entièrement son comportement, nous voulons simplement l'étendre un peu, nous dérivons donc `HTTPRedirectHandler` de manière à pouvoir appeler la classe ancêtre pour faire le gros du travail.

❷ Quand il reçoit un code de statut `301` du serveur, `urllib2` recherche parmi ses gestionnaires et appelle la méthode `http_error_301`. La première chose que la notre fait est d'appeler la méthode `http_error_301` de la classe ancêtre, qui s'occupe du travail de base consistant à chercher l'en−tête `Location:` et à suivre la redirection à la nouvelle adresse.

❸ Voici l'étape−clé : avant le retour de fonction, nous stockons le code de statut (`301`) pour que le programme appelant puisse y accéder plus tard.

❹ Les redirections temporaires (codes de statut `302`) fonctionnent de la même manière : réécriture de la méthode `http_error_302`, appel de l'ancêtre et sauvegarde du code de statut avant retour.

A quoi cela nous avance−t−il ? Nous pouvons maintenant construire un *opener* d'URL avec notre gestionnaire de redirection spécialisé et il suivra toujours les redirections automatiquement, mais maintenant il exposera également le code de statut de redirection.

Exemple 11.12. Utilisation du gestionnaire de redirection pour détecter les redirections permanentes

```
>>> request = urllib2.Request('http://diveintomark.org/redir/example301.xml')
>>> import openanything, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> opener = urllib2.build_opener(
...     openanything.SmartRedirectHandler())          ❶
>>> f = opener.open(request)
```

```
connect: (diveintomark.org, 80)
send: 'GET /redir/example301.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 301 Moved Permanently\r\n'                    ❷
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 338
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:13:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml

>>> f.status                                                   ❸
301
>>> f.url
'http://diveintomark.org/xml/atom.xml'
```

❶ D'abord, nous construisons un *opener* d'URL avec le gestionnaire de redirection que nous venons de définir.

❷ Nous avons envoyé une requête et nous avons reçu un code de statut `301` en réponse. A ce moment, la méthode `http_error_301` est appelée. Nous appelons la méthode ancêtre qui suit la redirection et envoi une requête à la nouvelle adresse (`http://diveintomark.org/xml/atom.xml`).

❸ Voici le bénéfice de notre travail : maintenant, nous n'avons pas seulement accès à la nouvelle URL, mais également au code de statut de redirection, nous pouvons donc voir qu'il s'agit d'une redirection permanente. La prochaine fois que nous demanderont ces données, nous devrons les demander à la nouvelle adresse (`http://diveintomark.org/xml/atom.xml`, comme spécifié dans `f.url`). Si nous avons stocké l'adresse dans un fichier de configuration ou une base de données, nous devons la mettre à jour pour ne pas continuer à envoyer des requêtes à l'ancienne adresse. Il faut mettre à jour notre carnet d'adresse.

Le gestionnaire de redirection peut aussi nous apprendre quand nous *ne devons pas* mettre à jour notre carnet d'adresse.

Exemple 11.13. Utilisation du gestionnaire de redirection pour détecter les redirections temporaires.

```
>>> request = urllib2.Request(
...     'http://diveintomark.org/redir/example302.xml')       ❶
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /redir/example302.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 302 Found\r\n'                                ❷
```

```
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Location: http://diveintomark.org/xml/atom.xml
header: Content-Length: 314
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0                                    ❸
Host: diveintomark.org
User-agent: Python-urllib/2.1
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:18:21 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Content-Length: 15955
header: Connection: close
header: Content-Type: application/atom+xml
>>> f.status                                                  ❹
302
>>> f.url
http://diveintomark.org/xml/atom.xml
```

❶ C'est une URL d'exemple que j'ai configurée pour signaler aux clients de se rediriger *temporairement* sur `http://diveintomark.org/xml/atom.xml`.

❷ Le serveur renvoi un code de statut `302`, ce qui indique une redirection temporaire. Le nouvel emplacement temporaire des données est donné dans l'en-tête `Location:`.

❸ `urllib2` appelle notre méthode `http_error_302`, qui appelle la méthode ancêtre du même nom dans `urllib2.HTTPRedirectHandler`, qui suit la redirection vers le nouvel emplacement. Puis notre méthode `http_error_302` stocke le code de statut (`302`) pour que l'application appelante puisse l'utiliser plus tard.

❹ Nous y voilà, après avoir suivi la redirection vers `http://diveintomark.org/xml/atom.xml`. `f.status` nous dit que c'était une redirection temporaire, ce qui signifie que nous devons continuer à chercher les données à l'adresse originelle (`http://diveintomark.org/redir/example302.xml`). Peut-être que nous serons redirigés encore la prochaine fois, mais peut-être que non. Peut-être que nous seront redirigés vers une adresse différente, nous ne pouvons pas le dire. Le serveur nous a indiqué que cette redirection était temporaire, nous devons suivre cette indication et maintenant que nous exposons assez d'information pour le faire, l'application appelante peut suivre cette indication.

11.8. Prise en charge des données compressées.

La dernière fonctionnalité importante du protocole HTTP que nous voulons supporter est la compression. Beaucoup de services Web ont la capacité d'envoyer les données compressées, ce qui qui peut réduire le volume de données envoyées de 60 % ou plus. C'est particulièrement vrai des services Web XML puisque les données XML se compressent très bien.

Les serveurs n'envoient de données compressées que si on déclare les prendre en charge.

Exemple 11.14. Déclarer au serveur que nous voulons des données compressées.

```
>>> import urllib2, httplib
>>> httplib.HTTPConnection.debuglevel = 1
>>> request = urllib2.Request('http://diveintomark.org/xml/atom.xml')
```

```
>>> request.add_header('Accept-encoding', 'gzip')          ❶
>>> opener = urllib2.build_opener()
>>> f = opener.open(request)
connect: (diveintomark.org, 80)
send: '
GET /xml/atom.xml HTTP/1.0
Host: diveintomark.org
User-agent: Python-urllib/2.1
Accept-encoding: gzip                                      ❷
'
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Thu, 15 Apr 2004 22:24:39 GMT
header: Server: Apache/2.0.49 (Debian GNU/Linux)
header: Last-Modified: Thu, 15 Apr 2004 19:45:21 GMT
header: ETag: "e842a-3e53-55d97640"
header: Accept-Ranges: bytes
header: Vary: Accept-Encoding
header: Content-Encoding: gzip                             ❸
header: Content-Length: 6289                               ❹
header: Connection: close
header: Content-Type: application/atom+xml
```

❶ C'est l'étape-clé : une fois que nous avons créé notre objet `Request`, nous ajoutons un en-tête `Accept-encoding` pour déclarer au serveur que nous acceptons les données encodées gzip. `gzip` est le nom de l'algorithme de compression que nous utilisons. En théorie il pourrait y en avoir d'autres, mais `gzip` est l'algorithme de compression utilisé par 99 % des serveurs Web.

❷ Voici l'en-tête envoyé au serveur.

❸ Et voici la réponse envoyée par le serveur : l'en-tête `Content-Encoding: gzip` signale que les données que nous allons recevoir sont compressées par gzip.

❹ L'en-tête `Content-Length` indique la longueur des données compressées, pas leur longueur décompressées. Comme nous allons le voir, la taille réelle des données décompressées est ici 15955, la compression gzip nous a donc permis de réduire la bande passante utilisée de plus de 60 % !

Exemple 11.15. Decompression des données

```
>>> compresseddata = f.read()                              ❶
>>> len(compresseddata)
6289
>>> import StringIO
>>> compressedstream = StringIO.StringIO(compresseddata)   ❷
>>> import gzip
>>> gzipper = gzip.GzipFile(fileobj=compressedstream)      ❸
>>> data = gzipper.read()                                  ❹
>>> print data                                             ❺
<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
  xmlns="http://purl.org/atom/ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:lang="en">
  <title mode="escaped">dive into mark</title>
  <link rel="alternate" type="text/html" href="http://diveintomark.org/"/>
  <-- rest of feed omitted for brevity -->
>>> len(data)
15955
```

❶ Nous continuons l'exemple précédent, `f` est l'objet-fichier retourné par l'*opener* d'URL. Appeler sa méthode `read()` nous permettrait d'habitude d'obtenir les données non compressées, mais ici ce n'est que la première étape puisque les données sont compressées par gzip.

❷ Cette étape est un peu du bidouillage. Python a un module `gzip` qui lit (et peut aussi écrire) des fichiers compressés par gzip sur le disque. Mais ici, nous n'avons pas de fichier sur le disque, nous avons un tampon de données compressées en mémoire et nous n'allons pas l'écrire dans un fichier temporaire uniquement pour le décompresser. Donc nous créons un objet−fichier à partir de ces données en mémoire (`compresseddata`) à l'aide du module `StringIO`. Nous avons vu le module `StringIO` au chapitre précédent, mais nous avons maintenant un autre emploi pour lui.

❸ Maintenant nous pouvons créer une instance de `GzipFile` et lui indiquer que son "fichier" est l'objet−fichier `compressedstream`.

❹ Voici la ligne qui effectue le véritable travail : "lire" `GzipFile` décompresse les données. C'est étrange, mais en fait il y a une logique. `gzipper` est un objet−fichier qui repreésente un fichier compressé par gzip. Mais ce "fichier" n'est pas un vrai fichier sur le disque, `gzipper` ne "lit" que l'objet−fichier que nous avons créé avec `StringIO` pour contenir les données compressées, qui sont elles−mêmes en mémoire dans la variable `compresseddata`. Et d'où viennent les données compressées ? Nous les avons téléchargées d'un serveur HTTP distant en "lisant" l'objet−fichier que nous avions construit avec `urllib2.build_opener`. Et tout cela fonctionne, chaque étape dans la chaîne n'a aucune idée que l'étape précédente ne produit pas un vrai fichier.

❺ Et voilà, de véritables données (15955 octets, plus précisément).

"Mais attendez !", vous exclamez−vous. "Cela pourrait être simplifié !" Je sais ce que vous pensez, vous vous dite que `opener.open` retourne un objet−fichier, alors pourquoi ne pas se débarasser de l'intermédiaire `StringIO` et passer `f` directement à `GzipFile` ? Bon, peut−être que vous ne pensiez pas ça, mais de toute manière ça ne marche pas.

Exemple 11.16. Decompression directe des données du serveur.

```
>>> f = opener.open(request)                    ❶
>>> f.headers.get('Content-Encoding')           ❷
'gzip'
>>> data = gzip.GzipFile(fileobj=f).read()      ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\lib\gzip.py", line 217, in read
    self._read(readsize)
  File "c:\python23\lib\gzip.py", line 252, in _read
    pos = self.fileobj.tell()   # Save current position
AttributeError: addinfourl instance has no attribute 'tell'
```

❶ En poursuivant l'exemple précédent, nous avons déjà un objet `Request` avec un en−tête `Accept-encoding: gzip`.

❷ L'ouverture de la requête nous donne les en−têtes (mais ne télécharge pas encore les données). Comme vous pouvez le voir, les données qui ont été envoyées sont compressées par gzip.

❸ Puisque `opener.open` retourne un objet−fichier et que nous savons par les en−têtes que nous obtiendrons des données compressées par gzip en le lisant, pourquoi ne pas passer cet objet−fichier directement à `GzipFile` ? Comme nous "lisons" l'instance de `GzipFile`, elle "lira" les données compressées du serveur HTTP distant et les décompressera à la volée. C'est une bonne idée, mais malheureusement ça ne marche pas. A cause de la manière dont la compression gzip fonctionne, `GzipFile` doit sauvegarder sa position et se déplacer vers l'avant et l'arrière dans le fichier compressé. Cela ne marche pas lorsque le "file" est un flux d'octets provenant d'un serveur distant, tout ce que nous pouvons faire est de le recevoir un octet après l'autre, il est impossible de se déplacer d'avant en arrière dans le flux de données. Donc la bidouille inélégante consistant à utiliser `StringIO` est la meilleure solution : télécharger les données compressées, en faire un objet−fichier avec `StringIO` et décompresser les données depuis cet objet−fichier.

11.9. Assembler les pièces

Nous avons vu toutes les pièces nécessaires à la construction d'un client de services Web intelligent. Maintenat, voyons comment tout cela s'assemble.

Exemple 11.17. La fonction `openanything`

Cette fonction est définie dans `openanything.py`.

```
def openAnything(source, etag=None, lastmodified=None, agent=USER_AGENT):
    # non-HTTP code omitted for brevity
    if urlparse.urlparse(source)[0] == 'http':                                    ❶
        # open URL with urllib2
        request = urllib2.Request(source)
        request.add_header('User-Agent', agent)                                   ❷
        if etag:
            request.add_header('If-None-Match', etag)                             ❸
        if lastmodified:
            request.add_header('If-Modified-Since', lastmodified)                 ❹
        request.add_header('Accept-encoding', 'gzip')                             ❺
        opener = urllib2.build_opener(SmartRedirectHandler(), DefaultErrorHandler()) ❻
        return opener.open(request)                                               ❼
```

❶ `urlparse` est un module utile pour, vous l'avez devinez, analyser des URL. Sa fonction principale, appelée également `urlparse`, prend une URL en paramètre et la découpe en un tuple composé de (schème, domaine, chemin, paramètres, paramètres de la chaîne de requête et identificateur de fragment). Seul le schème nous intéresse ici, pour nous assurer qu'il s'agit bien d'une URL HTTP (que `urllib2` peut prendre en charge).

❷ Nous nous identifions auprès du serveur HTTP avec la chaîne `User-Agent` passée par la fonction appelante. Si aucune chaîne `User-Agent` n'a été passée en paramètre, nous utilisons la valeur par défaut définie plus haut dans le module `openanything.py`. Nous n'utilisons jamais la valeur par défaut définie par `urllib2`.

❸ Si nous avons une code de hachage `ETag`, nous l'envoyons dans l'en-tête `If-None-Match`.

❹ Si nous avons une date de dernière modification, nous l'envoyons dans l'en-tête `If-Modified-Since`.

❺ Nous indiquons au serveur que nous souhaitons des données compressées.

❻ Nous construisons un *opener* d'URL qui utilise *les deux* gestionnaires d'URL spécialisés : `SmartRedirectHandler` pour gérer les redirections `301` et `302` et `DefaultErrorHandler` pour gérer les codes `304`, `404` et les autres erreurs.

❼ Et voilà ! Nous ouvrons l'URL et retournons un objet-fichier à l'appelant.

Exemple 11.18. La fonction `fetch`

Cette fonction est définie dans `openanything.py`.

```
def fetch(source, etag=None, last_modified=None, agent=USER_AGENT):
    '''Fetch data and metadata from a URL, file, stream, or string'''
    result = {}
    f = openAnything(source, etag, last_modified, agent)            ❶
    result['data'] = f.read()                                       ❷
    if hasattr(f, 'headers'):
        # save ETag, if the server sent one
        result['etag'] = f.headers.get('ETag')                      ❸
        # save Last-Modified header, if the server sent one
        result['lastmodified'] = f.headers.get('Last-Modified')     ❹
        if f.headers.get('content-encoding', '') == 'gzip':         ❺
            # data came back gzip-compressed, decompress it
```

```
        result['data'] = gzip.GzipFile(fileobj=StringIO(result['data'])).read()
    if hasattr(f, 'url'):                                                    ❻
        result['url'] = f.url
        result['status'] = 200
    if hasattr(f, 'status'):                                                 ❼
        result['status'] = f.status
    f.close()
    return result
```

❶ D'abord, nous appelons la fonction `openAnything` avec une URL, un code de hachage `ETag`, une date `Last-Modified` et une chaîne `User-Agent`.

❷ Nous lisons les données retournées par le serveur. Si elles sont compressées, nous les décompresserons plus tard.

❸ Nous sauvegardons le code de hachage `ETag` retourné par le serveur pour que l'application appelante puisse nous la passer à nouveau la prochaine fois et que nous la passions à `openAnything`, qui la mettra dans l'en-tête `If-None-Match` et l'enverra au serveur distant.

❹ Nous sauvegardons aussi la date `Last-Modified`.

❺ Si le serveur indique qu'il a envoyé des données compressées, nous les décompressons.

❻ Si le serveur nous envoi une URL, nous la sauvegardons et considérons que le code de statut est `200` jusqu'à preuve du contraire.

❼ Si un des gestionnaires d'URL spécialisés a obtenu un code de statut, nous le sauvegardons également.

Exemple 11.19. Utilisation de `openanything.py`

```
>>> import openanything
>>> useragent = 'MyHTTPWebServicesApp/1.0'
>>> url = 'http://diveintopython.org/redir/example301.xml'
>>> params = openanything.fetch(url, agent=useragent)                    ❶
>>> params                                                               ❷
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': 'Thu, 15 Apr 2004 19:45:21 GMT',
'etag': '"e842a-3e53-55d97640"',
'status': 301,
'data': '<?xml version="1.0" encoding="iso-8859-1"?>
<feed version="0.3"
<-- rest of data omitted for brevity -->'}
>>> if params['status'] == 301:                                          ❸
...     url = params['url']
>>> newparams = openanything.fetch(
...     url, params['etag'], params['lastmodified'], useragent)          ❹
>>> newparams
{'url': 'http://diveintomark.org/xml/atom.xml',
'lastmodified': None,
'etag': '"e842a-3e53-55d97640"',
'status': 304,
'data': ''}                                                              ❺
```

❶ La toute première fois que nous allons chercher une ressource, nous n'avons pas de code de hachage `ETag` ni de date `Last-Modified`, donc nous laissons ces paramètres vides (ce sont des paramètres optionnels).

❷ Ce qui nous est retourné est un dictionnaire contenant les en-têtes, le code de statut HTTP et les données retournées par le serveur. le module `openanything` s'occupe de la compression gzip en internet, nous n'avons donc pas à nous en occuper à ce niveau.

❸ Si nous recevons un code de statut `301`, c'est une redirection permanente et nous devons mettre à jour notre URL à la nouvelle adresse.

❹ La deuxième fois que nous allons chercher la même ressource, nous avons toutes sortes d'information a passer en argument : une URL (éventuellement mise à jour), le `ETag` et la date `Last-Modified` de la dernière requête et bien sûr la chaîne `User-Agent`.

❺ Ce qui nous est retourné est à nouveau un dictionnaire, mais les données n'ayant pas changé, nous n'obtenons qu'un code de statut `304` et aucune données.

11.10. Résumé

`openanything.py` et ses fonctions devraient être tout à fait clairs maintenant.

Il y a 5 fonctionnalités importantes des services Web HTTP que chaque client devrait supporter :

- Identifier l'application en définissant une chaîne `User-Agent` appropriée.
- Prendre en charge les redirections permanentes correctement.
- Supporter la vérification de date `Last-Modified` pour éviter de télécharger à nouveau des données non modifiées.
- Supporter les codes de hachage `ETag` pour éviter de télécharger à nouveau des données non modifiées.
- Supporter la compression gzip pour réduire la consommation de bande passante lorsque les données *ont* été modifiées.

Chapitre 12. Services Web SOAP

Le Chapitre 11 était centré sur les services Web HTTP orientés documents. Le "paramètre d'entrée" était l'URL et la "valeur de retour" était un document XML qu'il était de notre responsabilité d'analyser.

Le présent chapitre sera centré sur les services Web SOAP, qui ont une approche plus structurée. Au lieu d'être directement confronté aux requêtes HTTP et aux documents XML, SOAP permet de simuler des appels de fonctions qui retourne des types de données natifs. Comme vous le verrez, l'illusion est presque parfaite, on peut "appeler" une fonction à travers une bibliothèque SOAP avec la syntaxe d'appel standard de Python et la fonction semble retourner des objets et des valeurs Python. Mais en coulisse, la bibliothèque SOAP effectue une transaction complexe impliquant de multiples documents XML et un serveur distant.

SOAP est une spécification complexe et il est un peu trompeur de dire qu'il s'agit seulement d'appel de fonctions distants. Certains diraient qu'en plus SOAP permet le passage unidirectionnel asynchrone de messages et des services Web orientés documents et ils auraient raison. SOAP peut être utilisé de cette manière et de bien d'autres encore. Mais ce chapitre s'en tiendra à ce qu'on appelle SOAP "de type RPC", appeler une fonction distante et obtenir un résultat en retour.

12.1. Plonger

Vous utilisez Google, n'est−ce pas ? N'avez−vous jamais souhaité accéder aux résultats de recherches Google par la programmation ? Maintenant, vous pouvez le faire, voici un programme Python qui fait des recherches sur Google.

Exemple 12.1. `search.py`

```
from SOAPpy import WSDL

# you'll need to configure these two values;
# see http://www.google.com/apis/
WSDLFILE = '/path/to/copy/of/GoogleSearch.wsdl'
APIKEY = 'YOUR_GOOGLE_API_KEY'

_server = WSDL.Proxy(WSDLFILE)
def search(q):
    """Search Google and return list of {title, link, description}"""
    results = _server.doGoogleSearch(
        APIKEY, q, 0, 10, False, "", False, "", "utf-8", "utf-8")
    return [{"title": r.title.encode("utf-8"),
             "link": r.URL.encode("utf-8"),
             "description": r.snippet.encode("utf-8")}
            for r in results.resultElements]

if __name__ == '__main__':
    import sys
    for r in search(sys.argv[1])[:5]:
        print r['title']
        print r['link']
        print r['description']
        print
```

Ce programme peut être importé comme module pour être utilisé dans un plus grand programme, ou utilisé depuis la ligne de commande. En ligne de commande, la requête de recherche est donné comme argument de la commande et le programme affiche l'URL, le titre et la description des cinq première réponses données par Google.

Voici un exemple de sortie pour une recherche du mot "python".

Exemple 12.2. Exemple d'usage de `search.py`

```
C:\diveintopython\common\py> python search.py "python"
<b>Python</b> Programming Language
http://www.python.org/
Home page for <b>Python</b>, an interpreted, interactive, object-oriented,
extensible<br> programming language. <b>...</b> <b>Python</b>
is OSI Certified Open Source: OSI Certified.

<b>Python</b> Documentation Index
http://www.python.org/doc/
 <b>...</b> New-style classes (aka descrintro). Regular expressions. Database
API. Email Us.<br> docs@<b>python</b>.org. (c) 2004. <b>Python</b>
Software Foundation. <b>Python</b> Documentation. <b>...</b>

Download <b>Python</b> Software
http://www.python.org/download/
Download Standard <b>Python</b> Software. <b>Python</b> 2.3.3 is the
current production<br> version of <b>Python</b>. <b>...</b>
<b>Python</b> is OSI Certified Open Source:

Pythonline
http://www.pythonline.com/


Dive Into <b>Python</b>
http://diveintopython.org/
Dive Into <b>Python</b>. <b>Python</b> from novice to pro. Find:
<b>...</b> It is also available in multiple<br> languages. Read
Dive Into <b>Python</b>. This book is still being written. <b>...</b>
```

Pour en savoir plus sur SOAP

- http://www.xmethods.net/ est un répertoire de services Web SOAP en accès public.
- La spécification de SOAP (http://www.w3.org/TR/soap/) est étonnamment lisible, si vous aimez ce genre de choses.

12.2. Installation des bibliothèques SOAP

Contrairement au reste de ce livre, ce chapitre utilise des bibliothèques qui ne sont pas distribuées avec Python.

Avant de plonger dans les services Web SOAP vous devez installer trois bibliothèques : PyXML, fpconst et SOAPpy.

12.2.1. Installation PyXML

La première bibliothèque dont nous avons besoin est PyXML, un ensemble de bibliothèques XML avancées qui proposent plus de fonctionnalités que les bibliothèques XML prédéfinies que nous avons étudié au Chapitre 9.

Procédure 12.1.

Voici la procédure pour installer PyXML:

1. Allez à http://pyxml.sourceforge.net/, cliquez sur Downloads et téléchargez la dernière version correspondant

à votre système d'exploitation.
2. Si vous utilisez Windows, il y a plusieurs choix possibles. Assurez−vous de télécharger la version de PyXML qui correspond à la version de Python que vous utilisez.
3. Double−cliquez sur le programme d'installation. Si vous téléchargez PyXML 0.8.3 pour Windows et Python 2.3, le programme d'installation sera nommé `PyXML−0.8.3.win32−py2.3.exe`.
4. Suivez les étapes du programme d'installation.
5. Une fois l'installation terminée, fermer le programme d'installation. Il n'y aura aucune indication visible de succès de l'installation (pas de programmes installés dans le menu Démarrer ni de raccourcis sur le bureau). PyXML est simplement une collection de bibliothèques XML utilisées par d'autres programmes.

Pour vérifier que vous avez installé PyXML correctement, lancez votre IDE Python et vérifiez la version des bibliothèques XML installées comme ci−dessous.

Exemple 12.3. Vérification de l'installation de PyXML

```
>>> import xml
>>> xml.__version__
'0.8.3'
```

Le numéro de version affiché doit correspondre à celui du programme d'installation de PyXML que vous avez exécuté.

12.2.2. Installation de fpconst

La deuxième bibliothèque dont nous avons besoin est fpconst, un ensemble de constantes et de fonctions pour manipuler les valeurs spéciales double précision IEEE754. Elles fournissent le support des valeurs spéciales Not−a−Number (NaN), Infinité positive (Inf) et Infinité négative (−Inf), qui font partie de la spécification des types de données SOAP

Procédure 12.2.

Voici la procédure pour installer fpconst:

1. Téléchargez la dernière version de fpconst à l'adresse http://www.analytics.washington.edu/statcomp/projects/rzope/fpconst/.
2. Il y a deux fichiers disponibles en téléchargement, un au format `.tar.gz` et l'autre au format `.zip`. Si vous utilisez Windows, téléchargez le fichier `.zip`, sinon téléchargez le fichier `.tar.gz`.
3. Décompressez le fichier téléchargez. Sous Windows XP, vous pouvez faire un clic droit sur le fichier et choisir Tout extraire, pour les versions antérieures de Windows vous aurez besoin d'un programme tiers comme WinZip. Sous Mac OS X, vous pouvez double−cliquer sur le fichier compressé pour le décompresser avec Stuffit Expander.
4. Ouvrez une fenêtre de terminal et allez dans le répertoire où vous avez décompressé les fichiers de fpconst.
5. Tapez **`python setup.py install`** pour lancer le programme d'installation.

Pour vérifier que vous avez installé fpconst correctement, lancez votre IDE Python et vérifiez le numéro de version.

Exemple 12.4. Vérifier l'installation de fpconst

```
>>> import fpconst
>>> fpconst.__version__
'0.6.0'
```

Ce numéro de version doit correspondre à celui de l'archive fpconst que vous avez téléchargée et installée.

12.2.3. Installation de SOAPpy

La troisième et dernière bibliothèque nécessaire est la bibliothèque SOAP elle−même : SOAPpy.

Procédure 12.3.

Voici la procédure pour installer SOAPpy :

1. Allez à l'adresse http://pywebsvcs.sourceforge.net/ et sélectionnez la dernière version officielle de la section SOAPpy.
2. Il y a deux téléchargements disponibles. Si vous utilisez Windows, téléchargez le fichier `.zip`, sinon téléchargez le fichier `.tar.gz`.
3. Décompressez le fichier téléchargé, comme vous l'avez fait pour fpconst.
4. Ouvrez une fenêtre de terminal et naviguez jusqu'au répertoire où vous avez décompressé les fichiers de SOAPpy.
5. Tapez **`python setup.py install`** pour lancer le programme d'installation.

Pour vérifier que vous avez installé SOAPpy correctement, lancez votre IDE Python et vérifiez le numéro de version.

Exemple 12.5. Vérification de l'installation de SOAPpy

```
>>> import SOAPpy
>>> SOAPpy.__version__
'0.11.4'
```

Ce numéro de version doit correspondre à celui de l'archive SOAPpy que vous avez téléchargée et installée.

12.3. Premiers pas avec SOAP

Le coeur de SOAP est la l'appel de fonction distant. Il existe un certain nombre de serveurs publics SOAP qui fournissent des fonctions simples à titre de démonstration.

Le serveur public SOAP le plus populaire est http://www.xmethods.net/. L'exemple suivant utilise une fonction de démonstration qui prend un code postal des Etats−Unis et retourne la température actuelle dans cette région.

Exemple 12.6. Obtenir la température actuelle

```
>>> from SOAPpy import SOAPProxy                          ❶
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> namespace = 'urn:xmethods−Temperature'                ❷
>>> server = SOAPProxy(url, namespace)                    ❸
>>> server.getTemp('27502')                               ❹
80.0
```

❶ On accède à un serveur distant SOAP à travers une classe de délégation (*proxy*), `SOAPProxy`. Cette classe se charge de tout le fonctionnement interne de SOAP pour nous, y compris la création des documents XML de requête à partir du nom de fonction et de la liste d'arguments, l'envoi des requêtes par HTTP au serveur distant SOAP, l'analyse du document XML de réponse et la création de valeurs de retour native Python. Nous verrons à quoi ressemblent ces documents XML dans la section suivante.

❷ Chaque service SOAP a une URL qui gère toutes les requêtes. La même URL est utilisée pour tous les appels de fonction. Ici, le service n'a qu'une seule fonction, mais plus loin nous verrons des exemples de l'API Google qui a plusieurs fonctions. L'URL du service est partagée par toutes les fonctions. Chaque service SOAP a aussi un espace de noms, qui est défini par le serveur et est complètement arbitraire. Il fait simplement partie de la configuration nécessaire pour appeler les méthodes SOAP. Il permet au serveur de partager une URL de service unique et d'aiguiller les requêtes entre plusieurs services indépendants les uns des autres. C'est un peu comme la séparation de modules Python en paquetages.

❸ Nous créons `SOAPProxy` avec l'URL du service et l'espace de noms du service. Cela ne provoque aucune connexion au serveur SOAP, mais crée simplement un objet Python local.

❹ Maintenant que tout est bien configuré, nous pouvons réellement appeler les méthodes SOAP distantes comme si elles étaient des fonctions locales. Nous passons des arguments comme pour des fonctions ordinaires et nous recevons une valeur de retour comme avec des fonctions ordinaires. Mais en coulisses, il se passe énormément de choses.

Allons voir en coulisses.

12.4. Débogage de services Web SOAP

Les bibliothèques SOAP fournissent une manière simple de voir ce qu'il se passe dans les coulisses.

Pour activer le débogage, il suffit d'assigner deux drapeaux dans la configuration de `SOAPProxy`.

Exemple 12.7. Débogage de services Web SOAP

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> n = 'urn:xmethods-Temperature'
>>> server = SOAPProxy(url, namespace=n)     ❶
>>> server.config.dumpSOAPOut = 1            ❷
>>> server.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('27502')    ❸
*** Outgoing SOAP ***************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*********************************************************************
*** Incoming SOAP ****************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>
</ns1:getTempResponse>
```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*********************************************************************

>>> temperature
80.0
```

❶ D'abord, nous créons `SOAPProxy` normalement, avec l'URL et l'espace de noms du service.

❷ Ensuite, nous activons le débogage en assignant `server.config.dumpSOAPIn` et `server.config.dumpSOAPOut`.

❸ Troisièmement, nous appelons la méthode SOAP distante comme d'habitude. La bibliothèque SOAP affiche aussi bien le document XML de requête sortant que le document de réponse XML entrant. On voit là tout le travail que `SOAPProxy` fait pour nous. C'est un peu intimidant, nous allons le décomposer.

La majeure partie du document XML de requête qui est envoyé au serveur est composée de code administratif. Vous pouvez ignorer toutes les déclarations d'espaces de noms, elles sont semblables pour tous les appels SOAP. Le coeur de "l'appel de fonction" est ce fragment à l'intérieur de l'élément `<Body>` :

```
<ns1:getTemp                                    ❶
  xmlns:ns1="urn:xmethods-Temperature"          ❷
  SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">27502</v1>            ❸
</ns1:getTemp>
```

❶ Le nom de l'élément est le nom de fonction, `getTemp`. `SOAPProxy` utilise `getattr` comme sélecteur. Au lieu d'appeler des méthodes locales différentes en fonction du nom de méthode, il utilise le nom de méthode pour construire le document XML de requête.

❷ L'élément XML de la fonction est contenu dans un espace de noms spécifique, qui est celui que nous avons spécifié lors de la création de l'objet `SOAPProxy` . Ne vous souciez pas de `SOAP-ENC:root`, c'est aussi du code administratif.

❸ L'argument de la fonction a également été traduit en XML. `SOAPProxy` examine par introspection chaque argument pour déterminer son type de données (dans le cas présent c'est une chaîne). Le type de données de l'argument va dans l'attribut `xsi:type`, suivi de sa valeur.

Le document XML de retour est tout aussi facile à comprendre, une fois que l'on sait ce qu'on doit ignorer. Observons ce fragment à l'intérieur de `<Body>`:

```
<ns1:getTempResponse                                    ❶
  xmlns:ns1="urn:xmethods-Temperature"                  ❷
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">80.0</return>              ❸
</ns1:getTempResponse>
```

❶ Le serveur enveloppe la valeur de retour de la fonction dans un élément `<getTempResponse>`. Par convention, cet élément-enveloppe est le nom de la fonction plus `Response`. Mais il pourrait être n'importe quoi d'autre, la chose importante que `SOAPProxy` prend en compte n'est pas le nom de l'élément, mais l'espace de noms.

❷ Le serveur retourne la réponse dans le même espace de noms que nous avons utilisé pour la requête, celui que nous avons spécifié à la création de `SOAPProxy`. Plus loin dans ce chapitre, nous verrons ce qu'il se passe si l'on oublie de spécifier l'espace de noms à la création de `SOAPProxy`.

❸ La valeur de retour est spécifiée, ainsi que son type de données (c'est un nombre à virgule flottante). `SOAPProxy` utilise ce type de données explicite pour créer un objet Python du type natif correspondant et le retourne.

12.5. Présentation de WSDL

Les appels de méthodes locaux sont délégués à la classe `SOAPProxy` qui les converti de manière transparente en appels de méthodes SOAP distants. Comme nous l'avons vu, c'est un gros travail et `SOAPProxy` le fait rapidement et de manière transparente. Mais ce que cette classe ne fait pas est de fournir un mode d'introspection de méthodes.

Considérez ceci : les deux sections précédentes ont montré un exemple d'appel distant à une méthode SOAP simple avec un seul argument et une seule valeur de retour, tous deux de types simples. Il faut pour cela connaître et gérer l'URL du service, l'espace de noms du service, le nom de la fonction, le nombre d'arguments et le type de données de chaque argument. Si l'une de ces informations est manquante ou fausse, l'appel ne se fait pas.

Cela ne devrait pas nous surprendre. Si nous voulons appeler une fonction locale, nous devons savoir dans quel paquetage ou module elle se trouve (l'équivalent de l'URL et de l'espace de noms du service). Nous devons aussi connaître le nom de la fonction et le nombre d'arguments. Python gère le typage de données sans types explicites, mais il nous faut quand même savoir combien d'arguments passer et combien de valeurs de retour attendre.

La grande différence, c'est l'introspection. Comme nous l'avons vu au Chapitre 4, Python excelle pour ce qui est de nous permettre de découvrir des informations sur les modules et les fonctions à l'exécution. Nous pouvons lister les fonctions d'un module et, avec un peu de travail, détailler les déclarations de fonctions et les arguments.

WSDL nous permet de faire cela avec les services Web SOAP. WSDL signifie "Web Services Description Language" (Langage de Description des Services Web). Bien que conçu de manière assez flexible pour décrire un grand nombre de types de services Web, il est le plus souvent utilisé pour décrire des services Web SOAP.

Un fichier WSDL est un fichier tout simple, plus précisement, c'est un fichier XML. En général il se trouve sur le serveur qui fournit les services Web SOAP qu'il décrit. Plus loin dans ce chapitre, nous téléchargerons le fichier WSDL de l'API Google et l'utiliserons localement. Cela ne veut pas dire que nous appellerons l'API Google localement, le fichier WSDL continuera de décrire les fonctions distantes fournies par le serveur de Google.

Un fichier WSDL contient une description de tout ce qui est nécessaire à l'appel d'un service Web SOAP :

- L'URL et l'espace de noms du service
- Le type de service Web (en général des appels de fonctions par SOAP, bien que, comme je l'ai dit plus haute, WSDL peut décrire une grande variété de services Web)
- La liste des fonctions disponibles
- Les arguments de chaque fonction
- Le type de données de chaque argument
- La valeur de retour de chaque fonction et le type de données de chaque valeur de retour

En d'autres termes, un fichier WSDL nous dit tout ce que nous avons besoin de savoir pour pouvoir appeler le service Web SOAP

12.6. Introspection de services Web SOAP avec WSDL

Comme beaucoup de choses dans le domaine des services Web, WSDL a un longue et tortueuse histoire, pleine de controverses politiques et d'intrigue. Je n'aborderais pas du tout cette histoire, je la trouve ennuyeuse à pleurer. Il y a eu des normes concurrentes pour remplir ces fonctions, mais WSDL a gagné, donc apprenons à l'utiliser.

La chose la plus importante que WSDL permet de faire est la découverte des méthodes offertes par un serveur SOAP.

Exemple 12.8. Découverte des méthodes disponibles

```
>>> from SOAPpy import WSDL                                    ❶
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)                              ❷
>>> server.methods.keys()                                      ❸
[u'getTemp']
```

❶ SOAPpy intègre un analyseur WSDL. Au moment où j'écris ces lignes, il est considéré comme aux premiers stades de son développement, mais je n'ai eu aucun problème pour analyser les fichiers WSDL que j'ai testé.

❷ Pour utiliser un fichier WSDL, nous utilisons à nouveau une classe de délégation, `WSDL.Proxy`, qui prend un seul argument : le fichier WSDL. Notez qu'ici nous lui passons l'URL d'un fichier WSDL situé sur le serveur distant, mais la classe de délégation marche tout aussi bien avec une copie locale du fichier WSDL. La création de la classe déclenchera le téléchargement et l'analyse du fichier WSDL, donc si il contient des erreurs (ou s'il est inaccessible à cause de problèmes de réseau) nous l'apprenons immédiatement.

❸ La classe de délégation WSDL expose les fonctions disponibles sous la forme d'un dictionnaire Python, `server.methods`. Obtenir la liste des méthodes disponibles consiste donc simplement à appeler la méthode de dictionnaire `keys()`.

Donc, nous savons que le serveur SOAP offre un méthode unique : `getTemp`. Mais comment l'appeler ? L'objet de délégation WSDL peut nous l'indiquer.

Exemple 12.9. Découverte des arguments d'une méthode

```
>>> callInfo = server.methods['getTemp']                       ❶
>>> callInfo.inparams                                          ❷
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AD0>]
>>> callInfo.inparams[0].name                                  ❸
u'zipcode'
>>> callInfo.inparams[0].type                                  ❹
(u'http://www.w3.org/2001/XMLSchema', u'string')
```

❶ Le dictionnaire `server.methods` contient une structure spécifique à SOAPpy appelée `CallInfo`. Un objet `CallInfo` contient des informations au sujet d'une fonction spécifique, y compris ses arguments.

❷ Les arguments de la fonction sont stockés dans `callInfo.inparams`, qui est une liste Python d'objets `ParameterInfo` qui contiennent des informations sur chaque paramètre.

❸ Chaque objet `ParameterInfo` contient un attribut `name`, qui est le nom de l'argument. Il n'est pas nécessaire de connaître le nom de l'argument pour appeler la fonction par SOAP, mais SOAP permet l'appel de fonction avec des arguments nommés (tout comme Python) et `WSDL.Proxy` fera la correspondance entre les arguments nommés et la fonction distante si ils sont utilisés.

❹ Chaque paramètre est explicitement typé, les types de données étant définis par XML Schema. Nous l'avons vu dans la sortie de la section précédente, l'espace de noms XML Schema faisait partie du "code administratif" que je vous avais dit d'ignorer et vous pouvez continuer de l'ignorer. Le paramètre `zipcode` est une chaîne et si vous passez une chaîne Python à l'objet `WSDL.Proxy` il l'enverra au serveur.

WSDL permet également de connaître par introspection les valeurs de retour d'une fonction.

Exemple 12.10. Découverte des valeurs de retour d'une fonction

```
>>> callInfo.outparams                                         ❶
[<SOAPpy.wstools.WSDLTools.ParameterInfo instance at 0x00CF3AF8>]
>>> callInfo.outparams[0].name                                 ❷
u'return'
>>> callInfo.outparams[0].type
```

```
(u'http://www.w3.org/2001/XMLSchema', u'float')
```

❶ L'équivalent de `callInfo.inparams` pour les valeurs de retour est `callInfo.outparams`. C'est aussi une liste, car les fonctions appelées par SOAP peuvent retourner des valeurs multiples, tout comme les fonctions Python.

❷ Chaque objet `ParameterInfo` contient des variables `name` et `type`. Ici, la fonction retourne une seule valeur, appelée `return` et qui est un nombre à virgule flottante.

Assemblons tout cela et appelons un service Web SOAP avec un objet de délégation WSDL

Exemple 12.11. Appel d'un service Web avec un objet de délégation WSDL

```
>>> from SOAPpy import WSDL
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl')
>>> server = WSDL.Proxy(wsdlFile)                    ❶
>>> server.getTemp('90210')                          ❷
66.0
>>> server.soapproxy.config.dumpSOAPOut = 1          ❸
>>> server.soapproxy.config.dumpSOAPIn = 1
>>> temperature = server.getTemp('90210')
*** Outgoing SOAP ***************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">90210</v1>
</ns1:getTemp>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*********************************************************************
*** Incoming SOAP ****************************************************
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
<ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:float">66.0</return>
</ns1:getTempResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*********************************************************************

>>> temperature
66.0
```

❶ La configuration est plus simple que pour appeler un service SOAP directement, car le fichier WSDL contient à la fois l'URL et l'espace de noms dont nous avons besoin pour appeler le service. La création de l'objet `WSDL.Proxy` entraîne le téléchargement et l'analyse du fichier WSDL et la configuration de l'objet `SOAPProxy` utilisé pour appelé le service Web SOAP

❷ Une fois l'objet `WSDL.Proxy` créé, nous pouvons appeler une fonction aussi simplement que nous l'avions fait avec l'objet `SOAPProxy`. Cela n'est pas surprenant, `WSDL.Proxy` n'est qu'une enveloppe autour de

`SOAPProxy` avec quelques méthodes d'introspection en plus, la syntaxe d'appel de fonctions est donc la même.

❸ Nous pouvons accéder à l'objet `SOAPProxy` de `WSDL.Proxy` par `server.soapproxy`. C'est utile pour activer le débogage, pour que l'objet `SOAPProxy` de l'objet de délégation WSDL affiche les documents XML en sortie et en entrée.

12.7. Recherche Google

Revenons au code d'exemple que nous avons vu au début du chapitre, qui effectue quelque chose de plus intéressant et de plus utile qu'obtenir la température.

Google fournit une API SOAP pour accéder aux résultats de recherche par la programmation. Pour l'utiliser, il faut s'inscrire aux Services Web Google.

Procédure 12.4. S'inscrire au Services Web Google

1. Allez à l'adresse http://www.google.com/apis/ et créez un compte Google. Il suffit d'une adresse e−mail. Après vous être inscrit, vous recevrez une clé de licence pour l'API Google par e−mail. Vous devrez passer cette clé en paramètre pour appeler les fonctions de recherche.
2. Toujours à l'adresse http://www.google.com/apis/, téléchargez le kit de développement des API Google. Il contient des exemples de code en de nombreux langages de programmation (mais pas en Python) et, surtout, il contient le fichier WSDL.
3. Décompressez le kit de développement et cherchez le fichier `GoogleSearch.wsdl`. Copiez ce fichier quelque part sur votre disque, vous en aurez besoin plus tard dans ce chapitre.

Une fois que vous avez une clé de licence et le fichier WSDL de Google, vous pouvez commencer vos expérimentation des services Web Google.

Exemple 12.12. Introspection des services Web Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl') ❶
>>> server.methods.keys()                                  ❷
[u'doGoogleSearch', u'doGetCachedPage', u'doSpellingSuggestion']
>>> callInfo = server.methods['doGoogleSearch']
>>> for arg in callInfo.inparams:                          ❸
...     print arg.name.ljust(15), arg.type
key             (u'http://www.w3.org/2001/XMLSchema', u'string')
q               (u'http://www.w3.org/2001/XMLSchema', u'string')
start           (u'http://www.w3.org/2001/XMLSchema', u'int')
maxResults      (u'http://www.w3.org/2001/XMLSchema', u'int')
filter          (u'http://www.w3.org/2001/XMLSchema', u'boolean')
restrict        (u'http://www.w3.org/2001/XMLSchema', u'string')
safeSearch      (u'http://www.w3.org/2001/XMLSchema', u'boolean')
lr              (u'http://www.w3.org/2001/XMLSchema', u'string')
ie              (u'http://www.w3.org/2001/XMLSchema', u'string')
oe              (u'http://www.w3.org/2001/XMLSchema', u'string')
```

❶ Pour commencer à utiliser les services Web Google, il suffit de créer un objet `WSDL.Proxy` et de lui indiquer votre copie locale du fichier WSDL de Google.

❷ D'après le fichier WSDL, Google fournit trois fonctions : `doGoogleSearch`, `doGetCachedPage` et `doSpellingSuggestion`. Elles font exactement ce que leur nom suggère : effectuer une recherche Google et retourner les résultats, obtenir la version dans le cache de Google d'une page Web et suggérer une orthographe différente pour les mots

couramment mal orthographiés dans les recherches.

❸ La fonction `doGoogleSearch` prend de nombreux paramètres de différents types. Notez que si le fichier WSDL peut vous dire le nom et le type des arguments, il ne peut pas vous dire leur signification et comment les utiliser. Il pourrait théoriquement vous indiquer la plage de valeurs légales pour chaque paramètre si seules certaines valeurs était acceptées, mais le fichier WSDL de Google n'est pas aussi détaillé. `WSDL.Proxy` ne fait pas de miracle, il ne peut vous donner que l'informtion contenue dans le fichier WSDL.

Voici un bref résumé de tous les paramètres de la fonction `doGoogleSearch` :

- `key` − La clé de licence reçue à l'inscription aux services Web Google.
- `q` − Le mot ou la phrase recherchés. La syntaxe est exactement la même que sur le formulaire Web de Google, vous pouvez donc utiliser les paramètres de recherche avancés.
- `start` − L'index de départ des résultats. Comme sur le site Web, la fonction retourne 10 résultats à la fois, si vous souhaitez accéder à la deuxième "page" de résultats, mettez `start` à 10.
- `maxResults` − Le nombre de résultats à retourner. Pour l'instant limité à 10, mais vous pouvez en demander moins si vous n'êtes intéressés que par quelques résultats et voulez préserver votre bande passante.
- `filter` − Si mis à `True`, Google filtrera les doublons des résultats.
- `restrict` − Donnez lui comme valeur `country` plus le code de pays pour obtenir des résultats d'un pays en particulier. Par exemple `countryUK` limitera la recherche au Royaume Uni. Vous pouvez également spécifier `linux`, `mac` ou `bsd` pour rechercher parmis un ensemble de sites techniques défini par Google ou `unclesam` pour rechercher des sites traitant du gouvernement des Etats−Unis.
- `safeSearch` − Si il est mis à `True`, Google filtrera les sites pornographiques.
- `lr` ("language restrict") − Donnez lui comme valeur le code d'une langue pour obtenir des résultats limités à une langue.
- `ie` et `oe` ("input encoding", encodage de l'entrée et "output encoding", encodage de la sortie) − N'est plus utilisé, l'entrée comme la sortie sont encodés en `utf−8`.

Exemple 12.13. Rechercher avec Google

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy('/path/to/your/GoogleSearch.wsdl')
>>> key = 'YOUR_GOOGLE_API_KEY'
>>> results = server.doGoogleSearch(key, 'mark', 0, 10, False, "",
...     False, "", "utf-8", "utf-8")                    ❶
>>> len(results.resultElements)                        ❷
10
>>> results.resultElements[0].URL                      ❸
'http://diveintomark.org/'
>>> results.resultElements[0].title
'dive into <b>mark</b>'
```

❶ Après l'initialisation de l'objet `WSDL.Proxy`, nous pouvons appeler `server.doGoogleSearch` avec les dix paramètres. Rappelez−vous d'utiliser votre propre clé de licence pour les services Web Google.

❷ Il y a beaucoup d'information retournée, mais regardons d'abord les résultats de la recherche proprement dite. Ils sont stockés dans `results.resultElements` et nous pouvons y accéder comme à n'importe quelle liste Python.

❸ Chaque élément de `resultElements` est un objet qui a de nombreux attributs utiles comme `URL`, `title`et `snippet`. A ce stade, nous pouvons utiliser les techniques d'introspection habituelles de Python comme **`dir(results.resultElements[0])`** pour voir les attributs disponibles. Nous pouvons aussi utiliser l'introspection sur l'objet de délégation WSDL pour examiner les `outparams` de la fonction. Les deux techniques vous donneront accès à la même information.

L'objet `results` contient plus que les résultats de la recherche proprement dits. Il contient également des informations sur la recherche elle−même, telles que le temps qu'elle a pris et le nombre de résultats trouvés (même si seuls 10 ont été retournés). L'interface Web Google montre ces informations et elle sont également disponibles par la programmation.

Exemple 12.14. Accéder aux informations secondaires de Google

```
>>> results.searchTime                          ❶
0.224919
>>> results.estimatedTotalResultsCount          ❷
29800000
>>> results.directoryCategories                 ❸
[<SOAPpy.Types.structType item at 14367400>:
 {'fullViewableName':
  'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark',
  'specialEncoding': ''}]
>>> results.directoryCategories[0].fullViewableName
'Top/Arts/Literature/World_Literature/American/19th_Century/Twain,_Mark'
```

❶ Cette recherche a pris 0.224919 secondes. Cela n'inclut pas le temps passé à envoyer et recevoir les documents XML SOAP. C'est uniquement le temps que Google a passé à traiter notre requête une fois celle−ci reçue.

❷ Au total, il y a approximativement 30 millions de résultats. Nous pouvons y accéder 10 par 10 en changeant le paramètre `start` et en appelant à nouveau `server.doGoogleSearch`.

❸ Pour certaines requêtes, Google retourne aussi une liste des catégories de l'Annuaire Google (http://directory.google.com/) qui s'y rapportent. Il suffit alors d'ajouter ces URL à http://directory.google.com/ pour construire le lien vers la page de la catégorie.

12.8. Recherche d'erreurs dans les services Web SOAP

Bien sûr, le monde des services Web SOAP n'est pas différent du reste. Parfois ça ne marche pas.

Comme nous l'avons vu au cours de ce chapitre, SOAP met en oeuvre un certain nombre de couches. Il y a la couche HTTP, puisque SOAP envoi des documents XML vers un serveur HTTP (et en reçoit en réponse). Donc, toutes les techniques de débogage que nous avons vu au Chapitre 11, *Services Web HTTP* entrent en jeu ici. Nous pouvons assigner **`httplib.HTTPConnection.debuglevel = 1`** pour afficher la communication HTTP.

Au−delà de la couche HTTP sous−jacente, il y a un certain nombre de choses qui peuvent mal se passer. SOAPpy remplit à merveille sa tâche de nous masquer la syntaxe SOAP, mais cela veut aussi dire qu'il peut être difficile de localiser le problème quand ça ne marche pas.

Voici quelques exemples d'erreurs communes que j'ai commises en utilisant SOAP et des erreurs qu'elles ont générées.

Exemple 12.15. Appel d'une méthode avec un objet de délégation mal configuré

```
>>> from SOAPpy import SOAPProxy
>>> url = 'http://services.xmethods.net:80/soap/servlet/rpcrouter'
>>> server = SOAPProxy(url)                                        ❶
>>> server.getTemp('27502')                                        ❷
<Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI:
Unable to determine object id from call: is the method element namespaced?>
```

❶ Avez-vous localisé l'erreur ? Nous créons un `SOAPProxy` manuellement et nous spécifions correctement l'URL du service, mais nous ne spécifions pas d'espace de noms. Puisque de multiples services peuvent être aiguillés depuis la même URL, l'espace de noms est essentiel pour déterminer le service auquel nous nous adressons et donc la méthode que nous appelons.

❷ Le serveur répond en envoyant une Faute SOAP, que SOAPpy transforme en exception Python de type `SOAPpy.Types.faultType`. Toutes les erreurs renvoyées par un serveur SOAP seront des Fautes SOAP, il est donc simple d'intercepter cette exception. Ici, la partie textuelle de la Faute SOAP nous donne un indice sur le problème : l'élément-méthode n'est pas dans un espace de noms car l'objet `SOAPProxy` n'a pas été configuré avec un espace de noms.

La mauvais configuration des éléments de base du service SOAP est un des problèmes que WSDL cherche à résoudre. Le fichier WSDL contient l'URL et l'espace de noms du service, il est donc impossible de se tromper. Bien sûr, il y a d'autres choses qui peuvent se passer.

Exemple 12.16. Appel de méthode avec de mauvais arguments

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> temperature = server.getTemp(27502)                                    ❶
<Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>      ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception while handling service request:
services.temperature.TempService.getTemp(int) -- no signature match>
```

❶ Avez-vous localisé l'erreur ? Elle est assez subtile : nous appelons `server.getTemp` avec un entier au lieu d'une chaîne. Comme nous l'avons vu avec l'introspection du fichier WSDL, la fonction SOAP `getTemp()` prend un seul argument, `zipcode`, qui doit être une chaîne. `WSDL.Proxy` *ne convertit pas* les types de données, vous devez fournir exactement les types que le serveur attend.

❷ A nouveau, le serveur retourne une Faute SOAP et la partie textuelle de l'erreur nous indique ou se trouve le problème : nous appelons la fonction `getTemp` avec un entier, mais il n'y a pas de fonction définie ayant ce nom et prenant un entier en paramètre. En théorie, SOAP permet de *surcharger* les fonctions, il peut donc y avoir dans un même service SOAP deux fonctions ayant le même nom et le même nombre d'arguments si les arguments sont de types différents. C'est pourquoi il est important de faire correspondre exactement les types de données et pourquoi `WSDL.Proxy` ne convertit pas les types pour nous. Si il le faisait, nous pourrions en fin de compte appeler une fonction complètement différente, ce qui serait très difficile à déboguer. Il vaut mieux qu'il soit pointilleux en ce qui concerne les types de données et qu'une erreur se produise le plus vite possible si ils ne sont pas corrects.

Il est également possible d'écrire du code Python qui attend un nombre de valeurs de retour différents de celui que la

fonction distante retourne effectivement.

Exemple 12.17. Appeler une méthode en attendant un nombre érroné de valeurs de retour

```
>>> wsdlFile = 'http://www.xmethods.net/sd/2001/TemperatureService.wsdl'
>>> server = WSDL.Proxy(wsdlFile)
>>> (city, temperature) = server.getTemp(27502)  ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unpack non-sequence
```

❶ Avez vous vu l'erreur ? `server.getTemp` ne retourne qu'une valeur, un nombre à virgule flottante, mais nous avons écrit du code qui considère qu'il va recevoir deux valeurs et essaye de les assigner à deux variables différentes. Notez que cela ne provoque pas de Faute SOAP. En ce qui concerne le serveur distant, tout s'est bien passé. L'erreur s'est produite *après* que la transaction SOAP se soit achevée, `WSDL.Proxy` a retourné un nombre à virgule flottante et l'interpréteur Python a tenté d'exécuter votre demande de le séparer entre deux variables différentes. Comme la fonction n'a retourné qu'une seule valeur, nous obtenons une exception Python et non une Faute SOAP.

Et le service Web de Google ? Le problème le plus courant que j'ai eu est d'oublier d'assigner la clé de licence correctement.

Exemple 12.18. Appel d'une méthode avec une erreur spécifique à l'application

```
>>> from SOAPpy import WSDL
>>> server = WSDL.Proxy(r'/path/to/local/GoogleSearch.wsdl')
>>> results = server.doGoogleSearch('foo', 'mark', 0, 10, False, "", ❶
...     False, "", "utf-8", "utf-8")
<Fault SOAP-ENV:Server:                                              ❷
 Exception from service object: Invalid authorization key: foo:
 <SOAPpy.Types.structType detail at 14164616>:
 {'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
```

```
'}>
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 453, in __call__
    return self.__r_call(*args, **kw)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 475, in __r_call
    self.__hd, self.__ma)
  File "c:\python23\Lib\site-packages\SOAPpy\Client.py", line 389, in __call
    raise p
SOAPpy.Types.faultType: <Fault SOAP-ENV:Server: Exception from service object:
Invalid authorization key: foo:
<SOAPpy.Types.structType detail at 14164616>:
{'stackTrace':
  'com.google.soap.search.GoogleSearchFault: Invalid authorization key: foo
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:220)
   at com.google.soap.search.QueryLimits.validateKey(QueryLimits.java:127)
   at com.google.soap.search.GoogleSearchService.doPublicMethodChecks(
     GoogleSearchService.java:825)
   at com.google.soap.search.GoogleSearchService.doGoogleSearch(
     GoogleSearchService.java:121)
   at sun.reflect.GeneratedMethodAccessor13.invoke(Unknown Source)
   at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
   at java.lang.reflect.Method.invoke(Unknown Source)
   at org.apache.soap.server.RPCRouter.invoke(RPCRouter.java:146)
   at org.apache.soap.providers.RPCJavaProvider.invoke(
     RPCJavaProvider.java:129)
   at org.apache.soap.server.http.RPCRouterServlet.doPost(
     RPCRouterServlet.java:288)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
   at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
   at com.google.gse.HttpConnection.runServlet(HttpConnection.java:237)
   at com.google.gse.HttpConnection.run(HttpConnection.java:195)
   at com.google.gse.DispatchQueue$WorkerThread.run(DispatchQueue.java:201)
Caused by: com.google.soap.search.UserKeyInvalidException: Key was of wrong size.
   at com.google.soap.search.UserKey.<init>(UserKey.java:59)
   at com.google.soap.search.QueryLimits.lookUpAndLoadFromINSIfNeedBe(
     QueryLimits.java:217)
   ... 14 more
'}>
```

❶ Avez vous vu l'erreur ? La syntaxe d'appel est correcte, ainsi que le nombre d'arguments et leurs types. Le problème est propre à l'application : le premier argument est supposé être ma clé de licence, mais `foo` n'est pas une clé Google valide.

❷ Le serveur Google répond par une Faute SOAP et un message d'erreur incroyablement long, qui comprend une trace de pile Java complète. Rappelez-vous que *toutes* les erreurs SOAP sont signalées par des Fautes SOAP : les erreurs de configurations, les erreurs dans les arguments de fonctions et les erreurs spécifiques à l'application, comme c'est le cas ici. Enterrée quelque part dans ce message d'erreur, il y a cette information cruciale : `Invalid authorization key: foo`.

Pour en savoir plus sur la recherche d'erreurs avec SOAP

- New developments for SOAPpy (http://www-106.ibm.com/developerworks/webservices/library/ws-pyth17.html) explique pas à pas une tentative de connexion à un service SOAP qui ne fonctionne pas comme il est dit.

12.9. Résumé

Les services Web SOAP sont très complexes. La spécification est très ambitieuse et tente de répondre à de nombreux cas d'utilisation différents des services Web. Ce chapitre a abordé quelques uns des cas d'utilisation les plus simples.

Avant de plonger dans le prochain chapitre, assurez−vous d'être à l'aise pour :

- Vous connecter à un serveur SOAP et appeler des fonctions distantes
- Charger un fichier WSDL et examiner les méthodes distantes par introspection
- Déboguer les appels SOAP avec le traçage des communications
- Rechercher les erreurs courantes avec SOAP

Chapitre 13. Tests unitaires

13.1. Introduction au chiffres romains

Dans les chapitres précédents, nous avons "plongé" en regardant immédiatement du code et en essayant de le comprendre le plus vite possible. Maintenant que vous connaissez un peu plus de Python, nous allons prendre un peu de recul et regarder ce qui se passe *avant* que le code soit écrit.

Dans ce chapitres et les suivants, nous allons écrire, déboguer et optimiser un ensemble de fonctions utilitaires pour convertir vers et depuis des chiffres romains. Nous avons vu la méthode de construction et de validation des chiffres romains au chapitre Section 7.3, «Exemple : chiffres romains», nous allons maintenant considérer ce qu il faut faire pour étendre cette méthode pour qu elle fonctionne dans les deux sens.

Les règles de construction des chiffres romains amènent à un certain nombre d observations intéressantes :

1. Il n y a qu une seule façon correcte de représenter une valeur en chiffres romains.
2. L inverse est aussi vrai : si une chaîne de caractères en chiffres romains est un nombre valide, elle ne représente qu un nombre (c.a.d. qu elle ne peut être lue que d une manière).
3. Il y a un intervalle limité de valeurs pouvant être exprimées en chiffres romains, les nombres de `1` à `3999` (les Romains avaient plusieurs manières d exprimer des nombres plus grand, par exemple en inscrivant une barre au dessus d un caractère pour signifier que sa valeur normale devait être multipliée par `1000`, mais nous n allons pas prendre ça en compte. Pour ce qui est de ce chapitre, les chiffres romains vont de `1` à `3999`).
4. Il n est pas possible de représenter `0` en chiffres romains (étonnamment, les anciens romains n avaient pas de notion du `0` comme chiffre. Les nombres servaient à compter les choses qu on avait, comment compter ce que l on n a pas ?).
5. Il n est pas possible de représenter les valeurs négatives en chiffres romains.
6. Il n est pas possible de représenter des fractions ou des nombres non−entiers en chiffres romains.

Sachant tout cela, que pouvons nous exiger d un ensemble de fonctions pour convertir vers et depuis les chiffres romains ?

Spécification de `roman.py`

1. `toRoman` doit retourner la représentation en chiffres romains de tous les entiers entre `1` et `3999`.
2. `toRoman` doit échouer s il lui est passé un entier hors de l intervalle `1` à `3999`.
3. `toRoman` doit échouer s il lui est passé une valeur non−entière.
4. `fromRoman` doit prendre un nombre en chiffres romains valide et retourner la valeur qu il représente.
5. `fromRoman` doit échouer s il lui est passé un nombre en chiffres romains invalide.
6. Si vous prenez un nombre, le convertissez en chiffres romains, puis le convertissez à nouveau en nombre, vous devez obtenir la même valeur que celle de départ. Donc `fromRoman(toRoman(n)) == n` pour tout `n` compris dans `1..3999`.
7. `toRoman` doit toujours retourner un des chiffres romains en lettres majuscules.
8. `fromRoman` doit seulement accepter des chiffres romains en majuscules (il doit échouer s il lui est passé une entrée en minuscules.

Pour en savoir plus

- Ce site (http://www.wilkiecollins.demon.co.uk/roman/front.htm) a plus d information sur les nombres romains, y compris une histoire (http://www.wilkiecollins.demon.co.uk/roman/intro.htm) fascinante de la manière dont les Romains et d autres civilisations les utilisaient vraiment (pour faire court, à l aveuglette et

sans cohérence).

13.2. Présentation de `romantest.py`

Maintenant que nous avons défini entièrement le comportement que nous attendons de nos fonctions de conversion, nous allons faire quelque chose d un peu inattendu : nous allons écrire une suite de tests qui évalue ces fonctions et s assure qu elle se comporte comme nous voulons qu elles le fassent. Vous avez bien lu, nous allons écrire du code pour tester du code que nous n avons pas encore écrit.

C est ce qu on appelle des tests unitaires (*unit test*), puisque l ensemble des deux fonctions de conversion peut être écrit et testé comme une unité, séparée de tout programme plus grand dont elle puisse faire partie plus tard. Python a une bibliothèque pour les tests unitaires, un module nommé tout simplement `unittest`.

`unittest` est inclus dans Python 2.1 et versions ultérieures. Les utilisateurs de Python 2.0 peuvent le télécharger depuis `pyunit.sourceforge.net` (http://pyunit.sourceforge.net/).
Les tests unitaires sont une partie importante d une stratégie générale de développement centrée sur les tests. Si vous écrivez des tests unitaires, il est important de les écrire tôt (de préférence avant d écrire le code qu ils testent) et de les maintenir à jour au fur et à mesure que le code et les spécifications changent. Les tests unitaires ne remplacent pas les tests fonctionnels ou de système à plus haut niveau, mais ils sont important dans toutes les phases de développement :

- Avant d écrire le code, ils obligent a préciser le détail des spécification d une manière utile.
- Pendant l écriture du code, ils empêchent de trop programmer. Quand tous les cas de test passent, la fonction est terminée.
- Pendant la refactorisation (*refactoring*) de code, ils garantissent que la nouvelle version se comporte comme l ancienne.
- Pendant la maintenance du code, ils permettent d être couvert si quelqu un vient hurler que votre dernière modification fait planter leur code. ("Les tests unitaires passaient à l'intégration de mon code...")
- Lorsqu on écrit du code en équipe, ils permettent de s assurer que le code que vous intégrez ne va pas interférer avec celui des autres puisque vous pouvez d abord exécuter leurs tests. J ai vu ce genre de chose au cours de *code sprints*. L équipe se partage les tâches, chacun écrit les tests unitaires pour sa tâche à partir de sa spécification puis partage ses tests avec le reste de l équipe. De cette manière, personne ne peut s égarer à écrire du code qui ne fonctionnera pas avec celui des autres.

13.3. Présentation de `romantest.py`

Voici la suite de tests complète de nos fonctions de conversion de chiffres romains, qui n ont pas encore été écrites mais le seront dans `roman.py`. La manière dont tout ça fonctionne ensemble n est pas immédiatement évidente, aucune de ces classes ou méthodes ne se référencent entre elles. Il y a de bonnes raisons à cela, comme nous le verrons bientôt.

Exemple 13.1. `romantest.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Unit test for roman.py"""

import roman
import unittest
```

```
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
```

```
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
```

```
                                roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

Further reading

- Le site Web de PyUnit (http://pyunit.sourceforge.net/) présente un traitement en profondeur de l usage du module `unittest` (http://pyunit.sourceforge.net/pyunit.html), y compris des fonctionnalités avancées non couvertes par ce chapitre.
- La FAQ PyUnit (http://pyunit.sourceforge.net/pyunit.html) explique pourquoi les cas de test sont stockés séparément (http://pyunit.sourceforge.net/pyunit.html#WHERE) du code qu ils testent.
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume le module `unittest` (http://www.python.org/doc/current/lib/module−unittest.html).
- ExtremeProgramming.org (http://www.extremeprogramming.org/) explique pourquoi vous devriez écrire des tests unitaires (http://www.extremeprogramming.org/rules/unittests.html).
- Le Portland Pattern Repository (http://www.c2.com/cgi/wiki) propose une discussion en cours sur les tests unitaires (http://www.c2.com/cgi/wiki?UnitTests), y compris une définition standard (http://www.c2.com/cgi/wiki?StandardDefinitionOfUnitTest), pourquoi vous devriez écrire les tests unitaires en premier (http://www.c2.com/cgi/wiki?CodeUnitTestFirst) et de nombreuses études de cas (http://www.c2.com/cgi/wiki?UnitTestTrial) en profondeur.

13.4. Tester la réussite

La partie fondamentale des tests unitaires est la construction des cas de test individuels. Un cas de test répond à une seule question à propos du code qu il teste.

Un cas de test doit pouvoir :

- être exécuté complètement seul, sans entrée humaine. Les tests unitaires sont une question d automatisation.
- déterminer lui−même si la fonction qu il teste passe ou échoue au test, sans interprétation humaine du résultat.
- être exécuté de manière isolée, séparée de tout autre cas de test (même concernant la même fonction). Chaque cas de test est une île.

Sachant cela, construisons notre premier cas de test. Nous avons la spécification suivante :

1. `toRoman` doit retourner la représentation en chiffres romains de tous les entiers entre `1` et `3999`.

Exemple 13.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):                    ❶
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
```

```
                (500, 'D'),
                (1000, 'M'),
                (31, 'XXXI'),
                (148, 'CXLVIII'),
                (294, 'CCXCIV'),
                (312, 'CCCXII'),
                (421, 'CDXXI'),
                (528, 'DXXVIII'),
                (621, 'DCXXI'),
                (782, 'DCCLXXXII'),
                (870, 'DCCCLXX'),
                (941, 'CMXLI'),
                (1043, 'MXLIII'),
                (1110, 'MCX'),
                (1226, 'MCCXXVI'),
                (1301, 'MCCCI'),
                (1485, 'MCDLXXXV'),
                (1509, 'MDIX'),
                (1607, 'MDCVII'),
                (1754, 'MDCCLIV'),
                (1832, 'MDCCCXXXII'),
                (1993, 'MCMXCIII'),
                (2074, 'MMLXXIV'),
                (2152, 'MMCLII'),
                (2212, 'MMCCXII'),
                (2343, 'MMCCCXLIII'),
                (2499, 'MMCDXCIX'),
                (2574, 'MMDLXXIV'),
                (2646, 'MMDCXLVI'),
                (2723, 'MMDCCXXIII'),
                (2892, 'MMDCCCXCII'),
                (2975, 'MMCMLXXV'),
                (3051, 'MMMLI'),
                (3185, 'MMMCLXXXV'),
                (3250, 'MMMCCL'),
                (3313, 'MMMCCCXIII'),
                (3408, 'MMMCDVIII'),
                (3501, 'MMMDI'),
                (3610, 'MMMDCX'),
                (3743, 'MMMDCCXLIII'),
                (3844, 'MMMDCCCXLIV'),
                (3888, 'MMMDCCCLXXXVIII'),
                (3940, 'MMMCMXL'),
                (3999, 'MMMCMXCIX'))                          ❷

    def testToRomanKnownValues(self):                         ❸
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)                   ❹ ❺
            self.assertEqual(numeral, result)                 ❻
```

❶ Pour écrire un cas de test, commencez par dériver de la classe `TestCase` du module `unittest`. Cette classe fournit de nombreuses méthodes utiles que vous pouvez utiliser dans vos cas de test pour tester de conditions spécifiques.

❷ Voici un liste de paires entier/romains que j ai vérifié manuellement. Elle comprend les dix plus petits nombres, le plus grand nombre, chaque nombre représenté par un seul caractère en chiffres romains et un échantillon aléatoire d autres nombres valides. Le but d un test unitaire n est pas de tester toutes les entrées possibles, mais d en tester un échantillon représentatif.

❸ Chaque test individuel est sa propre méthode, qui ne doit prendre aucun paramètre et ne retourner aucune valeur. Si la méthode sort normalement sans déclencher d exception, le test

on considère que le test est passé, si la méthode déclenche une exception, on considère que le test a échoué.

❹ Ici, nous appelons la véritable fonction `toRoman` (pour l instant la fonction n a pas encore été écrite, mais quand elle le sera, c est cette ligne qui l appellera). Remarquez que nous avons maintenant défini l interface de la fonction `toRoman` : elle doit prendre un entier en paramètre (le nombre à convertir) et renvoyer une chaîne (les chiffres romains). Si l interface est différente de ça, on considère que le test a échoué.

❺ Remarquez également que nous ne tentons d intercepter aucune exception quand nous appelons `toRoman`. C est intentionnel. `toRoman` ne devrait pas déclencher d exception lorsque nous l appelons avec des paramètres d entrée valides et ces valeurs sont toutes valides. Si `toRoman` déclenche une exception, on considère que le test a échoué.

❻ En supposant que la fonction `toRoman` a été définie correctement, appelée correctement, qu elle s est terminée avec succès et qu elle a retourné une valeur, la dernière étape et de vérifier qu elle a retourné la *bonne* valeur. C est une question courante et la classe `TestCase` fournit une méthode, `assertEqual`, pour vérifier si deux valeurs sont égales. Si le résultat retourné par `toRoman` (`result`) ne correspond pas à la valeur connue que nous attendions (`numeral`), `assertEqual` déclenche une exception et le test échoue. Si les deux valeurs sont égales, `assertEqual` ne fera rien. Si chaque valeur retourné par `toRoman` correspond à la valeur connue que nous attendons, `assertEqual` ne déclenchera jamais d exception, donc `testToRomanKnownValues` se terminera finalement normalement, ce qui signifie que `toRoman` a passé ce test.

13.5. Tester l échec

Il ne suffit pas de tester que nos fonctions réussissent lorsqu on leur passe des entrées correctes, nous devons aussi tester qu elles échouent lorsque les entrées sont incorrectes. Et pas seulement qu elles échouent, qu elles échouent de la manière prévue.

Rappelez−vous nos autres spécifications pour `toRoman` :

2. `toRoman` doit échouer s il lui est passé un entier hors de l intervalle `1` à `3999`.
3. `toRoman` doit échouer s il lui est passé une valeur non−entière.

En Python, les fonctions indiquent l échec en déclenchant des exceptions et le module `unittest` fournit des méthodes pour tester si une fonction déclenche une exception en particulier lorsqu on lui donne une entrée incorrecte.

Exemple 13.3. Test des entrées incorrectes pour `toRoman`

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000) ❶

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)    ❷

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, −1)

    def testNonInteger(self):
        """toRoman should fail with non−integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)  ❸
```

❶ La classe `TestCase` de `unittest` fournit la méthode `assertRaises`, qui prend les arguments suivants : l exception attendue, la fonction testée et les arguments à passer à cette fonction. (Si la fonction testée prend plus d un argument, passez−les tous à `assertRaises`, dans l ordre, qui les passera à la fonction.) Faites bien attention à ce que nous faisons ici : au lieu d appeler la fonction `toRoman` directement et de vérifier manuellement qu elle déclenche une exception particulière (en l entourant d un bloc `try...except`), `assertRaises` encapsule tout ça pour nous. Tout ce que nous faisons est de lui donner l exception (`roman.OutOfRangeError`), la fonction (`toRoman`) et les arguments de `toRoman` (`4000`) et `assertRaises` s occupe d appeler `toRoman` et de vérifier qu elle décleche l exception `roman.OutOfRangeError`. (Est−ce que j ai dit récemment comme il est pratique que tout en Python est un objet, y compris les fonctions et les exceptions ?)

❷ En plus de tester les nombres trop grand, nous devons tester les nombres trop petits. Rappelez−vous, les chiffres romains ne peuvent exprimer `0` ou des valeurs négatives, donc nous avons un cas de test pour chacun (`testZero` et `testNegative`). Dans `testZero`, nous testons que `toRoman` déclenche une exception `roman.OutOfRangeError` lorsqu on l appelle avec `0`, si l exception `roman.OutOfRangeError` n est *pas* déclenchée (soit parce qu une valeur est retournée, soit parce qu une autre exception est déclenchée), le test est considéré comme ayant échoué.

❸ La spécification n°3 précise que `toRoman` ne peut accepter de non−entier, nous testons donc ici le déclenchement d une exception `roman.NotIntegerError` lorsque `toRoman` est appelée avec un nombre décimal (`0.5`). Si `toRoman` ne déclenche pas l exception `roman.NotIntegerError`, les test est considéré comme ayant échoué.

Les deux spécifications suivantes sont similaires aux trois premières, excepté le fait qu elles s appliquent à `fromRoman` au lieu de `fromRoman` :

4. `fromRoman` doit prendre un nombre en chiffres romains valide et retourner la valeur qu il représente.
5. `fromRoman` doit échouer s il lui est passé un nombre romain invalide.

La spécification n°4 est prise en charge de la même manière que la spécification n°1, en parcourant un échantillon de valeurs connues et en les testant une à une. La spécification n°5 est prise en charge de la même manière que les spécifications n°2 et 3, en testant une série d entrées incorrectes et en s assurant que `fromRoman` déclenche l exception appropriée.

Exemple 13.4. Test des entrées incorrectes pour `fromRoman`

```
class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s) ❶

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

❶ Il n y a pas grand chose de nouveau à dire, c est la même méthode que celle que nous avons employé pour tester les entrées incorrectes pour `toRoman`. Je mentionne juste qu il y a une nouvelle exception : `roman.InvalidRomanNumeralError`. Cela fait un total de trois exceptions personnalisées à définir dans

`roman.py` (avec `roman.OutOfRangeError` et `roman.NotIntegerError`). Nous verrons comment définir ces exceptions quand nous commenceront vraiment l écriture de `roman.py` au chapitre suivant.

13.6. Tester la cohérence

Il est fréquent qu une unité de code contiennent un ensemble de fonctions réciproques, habituellement sous la forme de fonctions de conversion où l une converti de A à B et l autre de B à A. Dans ce cas, il est utile de créer un test de cohérence pour s assurer qu une conversion de A à B puis de B à A n introduit pas de perte de précision décimale, d erreurs d arrondi ou d autres bogues.

Considérez cette spécification :

6. Si vous prenez un nombre, le convertissez en chiffres romains, puis le convertissez à nouveau en nombre, vous devez obtenir la même valeur que celle de départ. Donc `fromRoman(toRoman(n)) == n` pour tout `n` compris dans `1..3999`.

Exemple 13.5. Test de `toRoman` et `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):        ❶ ❷
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result) ❸
```

❶ Nous avons déjà vu la fonction `range`, mais ici elle est appelée avec deux arguments, ce qui retourne une liste d entiers commençant au premier argument (`1`) et comptant jusqu au second argument (`4000`) *non compris*. L intervalle retourné est donc `1..3999`, ce qui est l étendue des valeurs pouvant être converties en nombre romains valides.

❷ Juste une note au passage, `integer` n est pas un mot−clé de Python, ici c est un nom de variable comme un autre.

❸ La logique de test elle−même est très simple : on prend une valeur (`integer`), on la converti en chiffres romains (`numeral`), puis on converti ce nombre en chiffres romains en une valeur (`result`) qui doit être la même que celle de départ. Dans le cas contraire, `assertEqual` déclenche une exception et le test sera immédiatement considéré comme ayant échoué. Si tous les nombres correspondent, `assertEqual` s exécutera silencieusement, la méthode `testSanity` entière s achèvera silencieusement et le test sera considéré comme ayant passé.

Les deux dernières spécifications sont différentes des autres car elles semblent à la fois arbitraire et triviales :

7. `toRoman` doit toujours retourner des chiffres romains en majuscules.
8. `fromRoman` doit seulement accepter des chiffres romains en majuscules (il doit échouer s il lui est passé une entrée en minuscules.

En fait, elles sont un peu arbitraire. Nous aurions pu stipuler, par exemple, que `fromRoman` accepterait une entrée en minuscules ou en casse mélangée. Mais elles ne sont pas totalement arbitraire pour autant, si `toRoman` retourne toujours une sortie en majuscule, `fromRoman` doit au moins accepter une entrée en majuscules, sinon notre test de cohérence (spécification n°6) échouera. Le fait qu il accepte *seulement* des majuscules est arbitraire, mais comme tout intégrateur système vous le dira, la casse est toujours importante, mieux vaut donc spécifier le comportement face à la casse dès le début. Et si cela vaut la peine d être spécifié, cela vaut la peine d être testé.

Exemple 13.6. Tester la casse

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())          ❶

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())                    ❷ ❸
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())  ❹
```

❶ Le plus intéressant dans ce cas de test, c est toutes les choses qu il ne teste pas. Il ne teste pas que la valeur retournée par `toRoman` est correcte ni même cohérente, ces questions sont traitées par d autres cas de test. Nous avons un cas de test uniquement consacré à la casse. On pourrait être tenté de le combiner avec le test de cohérence, puisque ces deux tests parcourent toute l étendue des valeurs et appellent `toRoman`. [7] Mais cela serait une violation de nos règles fondamentales : chaque cas de test doit répondre à une seule question. Imaginez que vous combiniez cette vérification de la casse avec le test de cohérence et que le test échoue. Vous auriez à faire une analyse en profondeur pour savoir quelle partie du cas de test en serait la cause. Si vous devez analyser les résultats de vos tests unitaires rien que pour savoir ce qu ils signifient, il est certain que vous avez mal conçus vos cas de test.

❷ Il y a ici une leçon similaire : même si "nous savons" que `toRoman` retourne toujours des majuscules, nous convertissons explicitement sa valeur de retour en majuscules pour tester que `fromRoman` accepte une entrée en majuscule. Pourquoi ? Parce que le fait que `toRoman` retourne toujours des majuscules est une spécification indépendante. Si nous changions cette spécification de manière, par exemple, à ce qu il retourne toujours des minuscules, le cas de test `testToRomanCase` devrait être modifié, mais celui−ci passerait toujours. C est une autre de nos règles fondamentales : chaque cas de test doit fonctionner de manière isolée de tous les autres. Chaque cas de test est un îlot.

❸ Notez que nous n assignons pas la valeur retournée par `fromRoman`. C est syntaxiquement légal en Python, si une fonction retourne un valeur mais que l appelant ne l assigne pas, Python se contente de jeter cette valeur de retour. Dans le cas présent, c est ce que nous voulons. Ce cas de test ne teste rien qui concerne la valeur de retour, il teste seulement que `fromRoman` accepte une entrée en majuscule sans déclencher d exception.

❹ Cette ligne est compliquée, mais elle est très similaire à ce que nous avons fait dans les test `ToRomanBadInput` et `FromRomanBadInput`. Nous testons que l appel d une fonction spécifique (`roman.fromRoman`) avec une fonction spécifique (`numeral.lower()`, la version en minuscules des chiffres romains en cours dans la boucle) déclenche une exception spécifique (`roman.InvalidRomanNumeralError`). Si c est le cas (à chaque itération de la boucle) le test passe, s il se passe quelque chose d autre ne serait−ce qu une fois (par exemple le déclenchement d une autre exception ou le retour d une valeur sans déclencher d exception) le test échoue.

Dans le chapitre suivant, nous verrons comment écrire le code qui passera ces tests.

---

[7] "Je peux résister à tout, sauf à la tentation." Oscar Wilde

Chapitre 14. Ecriture des tests en premier

14.1. `roman.py`, étape 1

Maintenant que nos tests unitaires sont complets, il est temps d écrire le code que nos cas de test essaient de tester. Nous allons faire cela par étapes, de manière à voir tous les cas échouer, puis à les voir passer un par un au fur et à mesure que nous remplissons les trous de `roman.py`.

Exemple 14.1. `roman1.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage1/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass                ❶
class OutOfRangeError(RomanError): pass          ❷
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass ❸

def toRoman(n):
    """convert integer to Roman numeral"""
    pass                                         ❹

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ C est de cette manière que l on défini ses propres exceptions en Python. Les exceptions sont des classes, on en crée de nouvelles en dérivant des exceptions existantes. Il est fortement recommandé (mais pas obligatoire) de dériver `Exception`, qui est la classe de base dont toutes les exceptions héritent. Ici, je définis `RomanError` (dérivée de `Exception`) comme classe de base de toutes mes autres exceptions à venir. C est une question de style, j aurais tout aussi bien pu dériver chaque exception directement de la classe `Exception`.

❷ Les exceptions `OutOfRangeError` et `NotIntegerError` seront utilisées plus tard par `toRoman` pour signaler diverses sortes d entrées invalides, tel que spécifié par `ToRomanBadInput`.

❸ L exception `InvalidRomanNumeralError` sera utilisée plus tard par `fromRoman` pour signaler une entrée invalide, comme spécifié par `FromRomanBadInput`.

❹ A cette étape, nous voulons définir l API de chacune de nos fonctions, mais nous ne voulons pas encore en écrire le code, nous les mettons donc en place à l aide du mot réservé Python `pass`.

Et maintenant, l instant décisif (roulement de tambour) : nous allons exécuter notre test unitaire avec cette ébauche de module. A ce niveau, chaque cas de test devrait échouer. En fait, si un cas de test passe à l étape 1, il faut retourner à `romantest.py` et rechercher comment nous avons écrit un test inutile au point de passer avec des fonctions ne faisant rien.

Exécutez `romantest1.py` avec l option de ligne de commande `−v`, qui donne une sortie plus détaillée pour voir exactement ce qui se passe à mesure que chaque test s exécute. Si tout se passe bien, votre sortie devrait ressembler à

ceci :

Exemple 14.2. Sortie de `romantest1.py` avec `roman1.py`

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in testRepeatedPairs
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93, in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in testNonInteger
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input                                  ❶
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError                                         ❷
----------------------------------------------------------------------
Ran 12 tests in 0.040s                                                  ❸
```

```
FAILED (failures=10, errors=2)                                    ❹
```

❶ Lancer le script exécute `unittest.main()`, qui exécute chaque cas de test, c est à dire chaque méthode de chaque classe dans `romantest.py`. Pour chaque cas de test, il affiche la `doc string` de la méthode et le résultat du test. Comme il était attendu, aucun de nos cas de test ne passe.

❷ Pour chaque test échoué, `unittest` affiche la trace de pile montrant exactement ce qui s est passé. Dans le cas présent, notre appel à `assertRaises` (appelé aussi `failUnlessRaises`) a déclenché une exception `AssertionError` car il s attendait à ce que `toRoman` déclenche une exception `OutOfRangeError`, ce qui ne s est pas produit.

❸ Après le détail, `unittest` affiche en résumé le nombre de tests réalisés et le temps que cela a pris.

❹ Le test unitaire dans son ensemble a échoué puisqu au moins un cas de test n est pas passé. Lorsqu un cas de test ne passe pas, `unittest` distingue les échecs des erreurs. Un échec est un appel à une méthode `assertXYZ`, comme `assertEqual` ou `assertRaises`, qui échoue parce que la condition de l assertion n est pas vraie ou que l exception attendue n a pas été déclenchée. Une erreur est tout autre sorte d exception déclenchée dans le code que l on teste ou dans le test unitaire lui−même. Par exemple, la méthode `testFromRomanCase` ("`fromRoman` doit seulement accepter une entrée en majuscules") provoque une erreur parce que l appel à `numeral.upper()` déclenche une exception `AttributeError`, `toRoman` étant supposé retourner une chaîne mais ne l ayant pas fait. Mais `testZero` ("`fromRoman` doit échouer avec 0 en entrée") provoque un échec parce que l appel à `fromRoman` n a pas déclenché l exception `InvalidRomanNumeral` que `assertRaises` attendait.

14.2. `roman.py`, étape 2

Maintenant que nous avons la structure de notre module `roman` en place, il est temps de commencer à écrire du code et à passer les cas de test.

Exemple 14.3. `roman2.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage2/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),                                  ❶
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
```

```
def toRoman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:      ❷
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ `romanNumeralMap` est un tuple de tuples qui définit trois choses :

1. La représentation en caractères des chiffres romains les plus élémentaires. Notez qu il ne s agit pas seulement des chiffres romains à un seul caractère mais que nous définissons également des paires comme `CM` ("cent de moins que mille"). Cela rendra notre code pour `toRoman` plus simple.
2. L ordre des chiffres romains. Ils sont listés par ordre décroissant de valeur, de `M` jusqu à `I`.
3. La valeur de chaque chiffre romain. Chaque tuple est une paire de `(`*`romain, valeur`*`)`.

❷ C est ici que nous bénéficions de notre structure de données élaborée, nous n avons pas besoin de logique particulière pour prendre en charge la règle de soustraction. Pour convertir en chiffres romains, nous parcourons simplement `romanNumeralMap` à la recherche de la plus grande valeur entière inférieure ou égale à notre entrée. Une fois que nous l avons trouvée, nous ajoutons sa représentation en chiffres romains à la fin de la sortie, soustrayons la valeur de l entrée et répétons l opération.

Exemple 14.4. Comment `toRoman` fonctionne

Si vous n êtes pas sûr de comprendre comment fonctionne `toRoman`, ajoutez une instruction `print` à la fin de la boucle `while` :

```
        while n >= integer:
            result += numeral
            n -= integer
            print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

`toRoman` a donc l air de marcher, du moins pour notre petit test manuel. Mais passera−t−il l test unitaire ? Et bien non, pas complètement.

Exemple 14.5. Sortie de `romantest2.py` avec `roman2.py`

Rappelez−vous d exécuter `romantest2.py` avec l option de ligne de commande `−v` pour obtenir une sortie détaillée.

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok                    ❶
```

```
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok          ❷
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL              ❸
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

❶ `toRoman` retourne bien toujours des majuscules puisque notre `romanNumeralMap` définit les représentation en nombres romains en majuscules. Donc ce test passe.

❷ Voici la grande nouvelle : cette version de la fonction `toRoman` passe le test des valeurs connues. Rappelez−vous, elle n est pas exhaustive mais elle teste largement la fonction avec un ensemble d entrées correctes, y compris les entrées pour chaque nombre romain d un caractère, l entrée la plus grande possible (`3999`) et l entrée produisant le nombre romain le plus long (`3888`). Arrivé là, on peut être raisonnablement confiant que la fonction marche pour toute valeur correcte qu il lui est soumise.

❸ Par contre, la fonction ne "marche" pas pour les valeurs incorrectes, elle échoue pour tous les tests de valeurs incorrectes. Cela semble logique puisque nous n avons pas écrit de vérification d entrée. Ces cas de test attendent le déclenchement d exceptions spécifiques (à l aide de `assertRaises`) et nous ne les déclenchons jamais. Nous le ferons à l étape suivante.

Voici le reste de la sortie du test unitaire, détaillant tous les échecs. Nous n en sommes plus qu à 10.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
```

```
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testNonInteger
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
----------------------------------------------------------------------
Ran 12 tests in 0.320s

FAILED (failures=10)
```

14.3. `roman.py`, étape 3

Maintenant que `toRoman` se comporte correctement avec des entrées correctes (des entiers de `1` à `3999`), il est temps de faire en sorte qu il se comporte bien avec des entrées incorrectes (tout le reste).

Exemple 14.6. `roman3.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage3/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):                                              ❶
        raise OutOfRangeError, "number out of range (must be 1..3999)"  ❷
    if int(n) <> n:                                                     ❸
        raise NotIntegerError, "non−integers can not be converted"

    result = ""                                                         ❹
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n −= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

❶ Voici un beau raccourci Pythonique : les comparaisons multiples. C est l équivalent de `if not ((0 < n) and (n < 4000))`, mais en beaucoup plus lisible. C est notre vérification d étendue, elle doit intercepter les entrées trop grandes, négatives ou égales à zéro.

❷

Pour déclencher vous−même une exception, utilisez l instruction `raise`. Vous pouvez déclencher n importe quelle exception prédéfinie ou que vous avez défini vous−même. Le deuxième paramètre, le message d erreur, est optionnel, il est affiché dans la trace de pile qui est affichée si l exception n est pas prise en charge.

❸ Ceci est notre vérification de nombre décimal. Les nombres décimaux ne peuvent pas être convertis en chiffres romains.

❹ Le reste de la fonction est inchangé.

Exemple 14.7. Gestion des entrées incorrectes par `toRoman`

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "non−integers can not be converted"
NotIntegerError: non−integers can not be converted
```

Exemple 14.8. Sortie de `romantest3.py` avec `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok ❶
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non−integer input ... ok       ❷
toRoman should fail with negative input ... ok          ❸
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ `toRoman` passe toujours le test des valeurs connues, ce qui est réconfortant. Tous les tests qui passaient à l étape 2 passent toujours, donc notre nouveau code n a rien endommagé.

❷ Plus enthousiasmant, maintenant notre test de valeurs incorrectes passe. Ce test, `testDecimal`, passe grâce à la vérification `int(n) <> n`. Lorsqu un nombre décimal est passé à `toRoman`, `int(n) <> n` le voit et déclenche l exception `NotIntegerError`, qui est ce que `testDecimal`attend.

❸ Ce test, `testNegative`, passe grâce à la vérification `not (0 < n < 4000)`, qui déclenche une exception `OutOfRangeError`, qui est ce que `testNegative` attend.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
```

```
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
----------------------------------------------------------------------
Ran 12 tests in 0.401s

FAILED (failures=6) ❶
```

❶ Nous n en sommes plus qu à 6 échecs, tous ayant trait à `fromRoman` : le test de valeurs connues, les trois tests de valeurs incorrectes, le test de casse et le test de cohérence. Cela signifie que `toRoman` a passé tous les tests qu il peut passer par lui−même. (Il joue un rôle dans le test de cohérence, mais ce test à également besoin de `fromRoman`, qui n est pas encore écrit.) Cela veut dire que nous devons arrêter d écrire le code de `toRoman` immédiatement. Pas de réglages, pas de bidouilles et pas de vérification supplémentaires "au cas où". Arrêtez. Maintenant. Ecartez vous du clavier.

La chose la plus importante que des tests unitaires complets vous disent est quand vous arrêter d écrire du code. Quand tous les tests unitaires d une fonction passent, arrêtez d écrire le code de la fonction. Quand tous les tests d un module passent, arrêtez d écrire le code du module.

14.4. `roman.py`, étape 4

Maintenant que `toRoman` est terminé, nous devons passer à `fromRoman`. Grâce à notre structure de données élaborée qui fait correspondre les nombres romains à des valeurs entières, ce n est pas plus difficile que pour `toRoman`.

Exemple 14.9. `roman4.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage4/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: ❶
            result += integer
            index += len(numeral)
    return result
```

❶ Le principe est le même que pour `toRoman`. Nous parcourons notre structure de données de chiffres romains (un tuple de tuples) et au lieu de chercher la plus grande valeur possible, nous cherchons la chaîne représentant les chiffres romains les plus "haut" possible.

Exemple 14.10. Comment `fromRoman` fonctionne

Si vous n êtes pas sûr de comprendre comment fonctionne `fromRoman`, ajoutez une instruction `print` à la fin de la boucle `while` :

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'found', numeral, 'of length', len(numeral), ', adding', integer

>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , of length 1, adding 1000
found CM , of length 2, adding 900
found L , of length 1, adding 50
found X , of length 1, adding 10
found X , of length 1, adding 10
found I , of length 1, adding 1
found I , of length 1, adding 1
1972
```

Exemple 14.11. Output of `romantest4.py` against `roman4.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok ❶
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok                  ❷
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ Il y a deux bonnes nouvelles. La première est que `fromRoman` marche pour des entrées correctes, au moins pour toutes les valeurs connues que nous testons.

❷ La deuxième est que notre test de cohérence passe également. Ces deux résultats nous permettent d être raisonnablement sûrs que `toRoman` comme `fromRoman` marchent correctement pour toutes les valeurs correctes. (Cela n est pas garanti, il est théoriquement possible que `toRoman` ait un bogue qui produise des chiffres romains erronés pour certaines entrées, *et* que `fromRoman` ait un bogue correspondant produisant les mêmes valeurs entières erronées pour les mêmes chiffres romains. En fonction de votre application et de vos besoins, cette possibilité peut vous déranger ; dans ce cas écrivez des tests plus exhaustifs jusqu à ce que vous ayez l esprit tranquille.)

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
```

```
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 12 tests in 1.222s

FAILED (failures=4)
```

14.5. `roman.py`, étape 5

Maintenant que `fromRoman` fonctionne pour des entrées correctes, nous devons mettre en place la dernière pièce du puzzle : le faire fonctionner avec des entrées incorrectes. Cela veut dire trouver une manière d examiner une chaîne et de déterminer si elle constitue un nombre en chiffres romains valide. C est intrinsèquement plus difficile que de valider une entrée numérique dans `toRoman`, mais nous avons un outil puissant à notre disposition : les expressions régulières.

Si vous n êtes pas familiarisé avec les expressions régulières et que vous n avez pas lu le Chapitre 7, *Expressions régulières*, il est sans doute temps de le faire.

Comme nous l avons vu au Section 7.3, «Exemple : chiffres romains», il y a plusieurs règles simples pour construire des nombres en chiffres romains à l aide des lettres `M`, `D`, `C`, `L`, `X`, `V` et `I`. Récapitulons ces règles :

1. Les caractères sont additifs. `I` est `1`, `II` est `2` et `III` est `3`. `VI` est `6` (littéralement "`5` et `1`"), `VII` est `7` et `VIII` est `8`.
2. Les caractères en un (`I`, `X`, `C`, and `M`) peuvent être répétés jusqu à trois fois. A `4`, vous devez soustraire du prochain caractère en cinq. Vous ne pouvez pas représenter `4` par `IIII`, au lieu de ça il est représenté par `IV` ("`1` de moins que `5`"). `40` s écrit `XL` ("`10` de moins que `50`"), `41` s écrit `XLI`, `42` `XLII`, `43` `XLIII` et `44` `XLIV` ("`10` de moins que `50`, puis `1` de moins que `5`").
3. De manière similaire, à `9`, vous devez soustraire du prochain caractère en un : `8` est `VIII` mais `9` est `IX` ("`1` de moins que `10`"), pas `VIIII` (puisque le caractère `I` ne peut être répété quatre fois). `90` est `XC` et `900` `CM`.
4. Les caractères en cinq ne peuvent être répétés. `10` est toujours représenté par `X`, jamais par `VV`. `100` est toujours `C`, jamais `LL`.
5. Les chiffres romains sont toujours écrits du plus haut vers le plus bas et lus de gauche à droite, l ordre des caractères est donc très important. `DC` est `600`, `CD` est un nombre totalement différent (`400`, "`100` de moins que `500`"). `CI` est `101`, `IC` n est même pas valide en chiffres romains (puisqu on ne peut pas soustraire `1` directement de `100`, il faudrait l écrire `XCIX`, "`10` de moins que `100`, puis `1` de moins que `10`").

Exemple 14.12. `roman5.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage5/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "non−integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n −= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):                                    ❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ C est simplement l extension du motif que nous avons vu au Section 7.3, «Exemple : chiffres romains». Les dizaines sont soit `XC` (`90`), soit `XL` (`40`), soit un `L` optionnel suivi de 0 à 3 `X` optionnels. Les unités sont soit `IX` (`9`), soit `IV` (`4`), soit un `V` optionnel suivi de 0 à 3 `I` optionnels.

❷

Une fois toute cette logique encodée dans notre expression régulière, le code vérifiant la validité des nombres romain est une formalité. Si `re.search` retourne un objet, alors l expression régulière à reconnu la chaîne et notre entrée est valide, sinon notre entrée est invalide.

A ce stade, vous avez le droit d être sceptique quant à la capacité de cette expression régulière longue et disgracieuse d intercepter tous les types de nombres romains invalides. Mais vous n avez pas à me croire sur parole, observez plutôt les résultats :

Exemple 14.13. Sortie de `romantest5.py` avec `roman5.py`

```
fromRoman should only accept uppercase input ... ok          ❶
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok      ❷
fromRoman should fail with repeated pairs of numerals ... ok ❸
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 2.864s

OK                                                           ❹
```

❶ Il y a une chose que je n ai pas mentionné à propos des expressions régulières, c est que par défaut, elles sont sensibles à la casse. Comme notre expression régulière `romanNumeralPattern` est exprimée en majuscules, notre vérification `re.search` rejettera toute entrée qui n est pas entièrement en majuscules. Donc notre test d entrée en majuscules uniquement passe.

❷ Plus important encore, notre test d entrée incorrecte passe. Par exemple, le test d antécédent mal formé vérifie les cas comme `MCMC`. Comme nous l avons vu, cela ne correspond pas à notre expression régulière, donc `fromRoman` déclenche une exception `InvalidRomanNumeralError`, ce qui est ce que le cas de test d antécédent mal formé attend, donc le test passe.

❸ En fait, tous les tests d entrées incorrectes passent. Cette expression régulière intercepte tout ce que nous avons pu imaginer quand nous avons écrit nos cas de test.

❹ Et le prix du triomphe modeste est attribué au petit "`OK`" qui est affiché par le module `unittest` quand tous les tests passent.

Quand tous vos tests passent, arrêtez d écrire du code.

Chapitre 15. Refactorisation

15.1. Gestion des bogues

Malgré tous vos efforts pour écrire des tests unitaires exhaustifs, vous aurez à faire face à des bogues. Mais qu est−ce que je veux dire par "bogue" ? Un bogue est un cas de test que vous n avez pas encore écrit.

Exemple 15.1. Le bogue

```
>>> import roman5
>>> roman5.fromRoman("") ❶
0
```

❶ Vous vous rappelez que dans la section précédente nous avons vu à chaque fois qu une chaîne vide était reconnue par l expression régulière que nous utilisons pour vérifier la validité des nombres romains. En fait, c est toujours vrai pour la version finale de l expression régulière. Et c est un bogue, nous voulons qu une chaîne vide déclenche une exception `InvalidRomanNumeralError` comme toute autre séquence de caractères qui ne représente pas un nombre romain valide.

Après avoir reproduit le bogue et avant de le corriger, vous devez écrire un cas de test qui échoue, de manière à l illustrer.

Exemple 15.2. Test du bogue (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):

    # previous test cases omitted for clarity (they haven't changed)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "") ❶
```

❶ C est plutôt simple. On appelle `fromRoman` avec une chaîne vide et on s assure qu un exception `InvalidRomanNumeralError` est déclenchée. Le plus dur était de trouver le bogue, maintenant qu on le connaît, le tester est facile.

Puisque notre code a un bogue et que nous avons maintenant un cas de test pour ce bogue, le cas de test va échouer :

Exemple 15.3. Sortie de `romantest61.py` avec `roman61.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non−integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

```
======================================================================
FAIL: fromRoman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 13 tests in 2.864s

FAILED (failures=1)
```

Maintenant nous pouvons corriger le bogue.

Exemple 15.4. Correction du bogue (`roman62.py`)

Ce fichier est disponible dans le sous−répertoire `py/roman/stage6/` du répertoire des exemples.

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s: ❶
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ Seulement deux lignes de code sont nécessaires : une vérification explicite de chaîne non nulle et une instruction `raise`.

Exemple 15.5. Sortie de `romantest62.py` avec `roman62.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok ❶
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 2.834s

OK ❷
```

❶ Le cas de test pour la chaîne vide passe maintenant, le bogue est donc corrigé.

❷ Les autres cas de test passent toujours, ce qui veut dire que la correction du bogue n a pas endommagé d autre code. Tous les tests passent, on arrête d écrire du code.

Programmer de cette manière ne rend pas la correction de bogues plus simple. Les bogues simples (comme ici) nécessitent des cas de tests simples, les bogues complexes de cas de tests complexes. Dans un environnement centré sur les tests, il peut *sembler* que la correction d un bogue prend plus de temps puisque vous devez définir exactement par du code ce qu est le bogue (pour écrire le cas de test) avant de corriger le bogue proprement dit. Puis, si le cas de test ne passe pas immédiatement, vous devez déterminer si la correction est erronée ou si le cas de test a lui−même un bogue. Cependant, à terme, ces aller−retours entre le code de test et le code testé est rentable car il rend plus probable la correction des bogues du premier coup. De plus, puisque vous pouvez facilement lancer *tous* les cas de tests en même temps que le nouveau, vous êtes beaucoup moins susceptibles d endommager une partie de l ancien code en corrigeant le nouveau. Les tests unitaires d aujourd hui sont les tests de non régression de demain.

15.2. Gestion des changements de spécification

Malgré vos meilleurs efforts pour plaquer vos clients au sol et leur extirper une définition de leurs besoins grâce à la menace, les spécifications vont changer. La plupart des clients ne savent pas ce qu ils veulent jusqu à ce qu ils le voient et même ceux qui le savent ne savent pas vraiment comment l exprimer. Et même ceux qui savent l exprimer voudront plus à la version suivante de toute manière. Préparez−vous donc à mettre à jour vos cas de test à mesure que vos spécifications changent.

Supposez, par exemple, que nous souhaitions élargir la portée de nos fonctions de conversion de chiffres romains. Vous vous rappelez de la règle disant qu aucun caractère ne peut être répété plus de trois fois ? Et bien, les Romains faisaient une exception à cette règle pour permettre de représenter `4000` par 4 `M`. Si nous faisons cette modification, nous pourrons agrandir l étendue de nombres que nous pouvons convertir de `1..3999` à `1..4999`. Mais d abord, nous devons modifier nos cas de test.

Exemple 15.6. Modification des cas de test pour prendre en charge de nouvelles spécifications (`romantest71.py`)

Ce fichier est disponible dans le sous−répertoire `py/roman/stage7/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
import roman71
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
```

```
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'),
                    (4000, 'MMMM'),                                    ❶
                    (4500, 'MMMMD'),
                    (4888, 'MMMMDCCCLXXXVIII'),
                    (4999, 'MMMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000) ❷
```

```
    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -1)

    def testNonInteger(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):     ❸
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):                                    ❹
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

❶ Les valeurs connues existantes ne changent pas (elles sont toujours des valeurs qu il est raisonnable de tester), mais nous devons en ajouter quelques unes au-dessus de `4000`. Nous incluons donc `4000` (le plus court), `4500` (le second en longueur), `4888` (le plus long) et `4999` (la plus grande valeur).

❷ La définition de "grande valeur d entrée" a changé. Ce test appelait `toRoman` avec `4000` et attendait une erreur, maintenant que `4000-4999` sont des valeurs correctes, nous devons remplacer l argument par `5000`.

❸ La définition de "trop de nombres romains répétés" a aussi changé. Ce test appelait `fromRoman` avec `'MMMM'` et attendait une erreur, maintenant que `MMMM` est considéré comme un nombre romain valide, nous devons le remplacer par `'MMMMM'`.

❹ Le test de cohérence et les tests de casse bouclent sur tous les nombres de `1` à `3999`. Maintenant que l étendue est agrandie, ces boucles `for` doivent être modifiées pour aller jusqu à `4999`.

Maintenant, nos cas de test sont à jours par rapport à nos nouvelles spécifications, mais notre code ne l est pas, on peut donc s attendre à ce que plusieurs tests échouent.

Exemple 15.7. Sortie de `romantest71.py` avec `roman71.py`

```
fromRoman should only accept uppercase input ... ERROR          ❶
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR   ❷
toRoman should give known result with known input ... ERROR     ❸
fromRoman(toRoman(n))==n for all n ... ERROR                    ❹
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

❶ Les vérifications de casse échouent puisqu elles bouclent de `1` à `4999` et que `toRoman` n accepte que des nombres de `1` à `3999`, la fonction échoue dès que le test lui passe `4000` comme argument.

❷ Le test de valeurs connues pour `fromRoman` échoue dès qu il arrive à `'MMMM'` puisque `fromRoman` considère toujours que c est un nombre romain non valide.

❸ Le test de valeurs connues pour `toRoman` échoue dès qu il arrive à `4000` puisque `toRoman` considère toujours que c est hors de l étendu valide.

❹ Le test de cohérence échoue également à `4000` puisque `toRoman` refuse cette valeur.

```
======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownValues
```

```
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
======================================================================
ERROR: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValues
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
----------------------------------------------------------------------
Ran 13 tests in 2.213s

FAILED (errors=5)
```

Maintenant que nous avons des cas de test qui échouent à cause des nouvelles spécifications, nous pouvons nous tourner vers la correction du code pour le mettre en concordance avec les tests. (Une des choses qui demande un peu de temps pour s y habituer lorsque vous commencez à utiliser les tests unitaires est que le code que l on teste n est jamais "en avance" sur les cas de test. Tant qu il est derrière, vous avez du travail à faire et dès qu il rattrape les cas de test, vous vous arrêtez d écrire du code.)

Exemple 15.8. Ecrire le code des nouvelles spécifications (`roman72.py`)

Ce fichier est disponible dans le sous−répertoire `py/roman/stage7/` du répertoire des exemples.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))
```

```
def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):                                                    ❶
        raise OutOfRangeError, "number out of range (must be 1..4999)"
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' ❷

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ `toRoman` n a besoin que d une petite modification, la vérification d étendue. Là où nous vérifions que `0 < n < 4000`, nous vérifions maintenant que `0 < n < 5000`. Nous changeons aussi le message d erreur de l instruction `raise` pour qu il corresponde à la nouvelle étendue (`1..4999` au lieu de `1..3999`). Nous n avons pas besoin de modifier le reste de la fonction, elle prend déjà en compte les nouveaux cas. (Elle ajoute `'M'` pour chaque mille qu elle trouve, pour `4000` elle donnera `'MMMM'`. La seule raison pour laquelle elle ne le faisait pas auparavant est que nous la stoppions explicitement par la vérification d étendue.)

❷ Nous n avons aucune modification à faire à `fromRoman`. La seule modification est pour `romanNumeralPattern`, si vous regardez attentivement, vous verrez que nous avons ajouté un autre `M` optionnel dans la première section de l expression régulière. Cela permet jusqu à 4 `M` au lieu de 3, ce qui veut dire que nous permettons l équivalent en nombres romains de `4999` au lieu de `3999`. La fonction `fromRoman` proprement dite est totalement générale, elle ne fait que rechercher des caractères représentant des nombres romains et les additionne, sans s occuper de savoir combien de fois ils sont répétés. La seule raison pour laquelle elle ne prenait pas `'MMMM'` en charge auparavant est que nous la stoppions explicitement avec le motif de l expression régulière.

Vous pouvez douter que ces deux petites modifications soient tout ce qui est nécessaire. Vous n avez pas à me croire sur parole, voyez par vous-même :

Exemple 15.9. Sortie de `romantest72.py` avec `roman72.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
```

```
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 3.685s

OK ❶
```

❶ Tous les cas de test passent. Arrêtez d écrire du code.

Des test unitaires exhaustifs permettent de ne jamais dépendre d un programmeur qui dit "Faites−moi confiance."

15.3. Refactorisation

Le meilleur avec des tests unitaires exhaustifs, ce n est pas le sentiment que vous avez quand tous vos cas de test finissent par passer, ni même le sentiment que vous avez quand quelqu un vous reproche d avoir endommagé leur code et que vous pouvez véritablement *prouver* que vous ne l avez pas fait. Le meilleur, c est que les tests unitaires vous permettent la refactorisation continue de votre code.

La refactorisation est le processus par lequel on fait d un code qui fonctionne un code qui fonctionne mieux. Souvent, "mieux" signifie "plus vite", bien que cela puisse aussi vouloir dire "avec moins de mémoire", "avec moins d espace disque" ou simplement "de manière plus élégante". Quoi que cela signifie pour vous, pour votre projet, dans votre environnement, la refactorisation est importante pour la santé à long terme de tout programme.

Ici, "mieux" veut dire "plus vite". Plus précisément, la fonction `fromRoman` est plus lente qu elle ne le devrait, à cause de cette énorme expression régulière que nous utilisons pour valider les nombres romains. Cela ne vaut sans doute pas la peine de se priver complètement de l expression régulière (cela serait difficile et ne serait pas forcément plus rapide), mais nous pouvons rendre la fonction plus rapide en précompilant l expression régulière.

Exemple 15.10. Compilation d expressions régulières

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')                ❶
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern) ❷
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)                  ❸
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')           ❹
<SRE_Match object at 01104928>
```

❶ C est la syntaxe que nous avons déjà vu : `re.search` prend une expression régulière sous forme de chaîne (`motif`) et une chaîne dont on va tester la correspondance (`'M'`). Si le motif reconnaît la chaîne, la fonction retourne un objet correspondance qui peut être interrogé pour savoir exactement ce qui a été reconnu et comment.

❷ C est une nouvelle syntaxe : `re.compile` prend une expression régulière sous forme de chaîne et retourne

un objet motif. Notez qu il n y a pas ici de chaîne à reconnaître. Compiler une expression régulière n a rien à voir avec la reconnaissance d une chaîne en particulier (comme `'M'`), cela n implique que l expression régulière elle−même.

❸ L objet motif compilé retourné par `re.compile` a plusieurs fonctions qui ont l air utile, parmi lesquelles plusieurs (comme `search` et `sub`) sont directement disponible dans le module `re`.

❹ Appeler la fonction `search` de l objet motif compilé avec la chaîne `'M'` accomplit la même chose qu appeler `re.search` avec l expression régulière et la chaîne `'M'`. Mais c est beaucoup, beaucoup plus rapide. (En fait, la fonction `re.search` se contente de compiler l expression régulière et d appeler la méthode `search` de l objet motif résultant pour vous.)

A chaque fois que vous allez utiliser une expression régulière plus d une fois, il vaut mieux la compiler pour obtenir un objet motif et appeler ses méthodes directement.

Exemple 15.11. Expressions régulières compilées dans `roman81.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage8/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$') ❶

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):                                    ❷
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

❶ Cela à l air très similaire mais en fait beaucoup a changé. `romanNumeralPattern` n est plus une chaîne, c est un objet motif qui est retourné par `re.compile`.

❷ Cela signifie que nous pouvons appeler des méthodes de `romanNumeralPattern` directement. Cela sera beaucoup plus rapide que d appeler `re.search` à chaque fois. L expression régulière est compilée une seule fois et est stockée dans `romanNumeralPattern` quand le module est importé pour la première fois, puis, à chaque fois que nous appelons `fromRoman`, nous pouvons immédiatement tester la correspondance de la chaîne d entrée avec l expression régulière, sans que des étapes intermédiaire interviennent en coulisse.

Mais à quel point est−ce plus rapide de compiler notre expression régulière ? Voyez vous−même :

Exemple 15.12. Sortie de `romantest81.py` avec `roman81.py`

```
.............                                                    ❶
----------------------------------------------------------------------
Ran 13 tests in 3.385s ❷
```

```
OK                                                              ❸
```

❶ Juste une note en passant : cette fois, j ai lancé le test unitaire *sans* l option `-v`, donc au lieu d avoir la `doc string` complète pour chaque test, nous avons un point pour chaque test qui passe. (Si un test échouait, nous aurions un `F` (*failed*) et si il y avait une erreur, nous aurions un `E`. Nous aurions quand même la trace de pile pour chaque échec ou erreur de manière à pouvoir localiser les problèmes.)

❷ Nous avons exécuté `13` tests en `3,385` secondes, au lieu de `3,685` secondes sans précompilation de l expression régulière. C est une amélioration de `8%` et rappelez−vous que la plus grande partie du temps passé dans le test unitaire est consacré à d autres chose. (J ai testé séparément l expression régulière et j ai découvert que sa compilation accélère la fonction `search` de `54%` en moyenne.) Pas mal pour une modification aussi simple.

❸ Oh, au cas ou vous vous le demandiez, la précompilation de l expression régulière n a rien endommagé et nous venons de le prouver.

Il y a une autre optimisation que je veux essayer. Etant donnée la complexité de la syntaxe des expressions régulières, il n est pas étonnant qu il y ait souvent plus d une manière d écrire la même expression. Après une discussion à propos de ce module sur comp.lang.python (http://groups.google.com/groups?group=comp.lang.python), quelqu un m a suggéré d utiliser la syntaxe `{m,n}` pour des caractères répétés optionnels.

Exemple 15.13. `roman82.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage8/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')

#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$') ❶
```

❶ Nous avons remplacé `M?M?M?M?` par `M{0,4}`. Les deux signifient la même chose : "reconnais de 0 à 4 `M`". De même, `C?C?C?` est devenu `C{0,3}` ("reconnais de 0 à 3 `C`") et ainsi de suite pour `X` et `I`.

Cette forme d expression régulière est un petit peu plus courte (mais pas plus lisible). La question est, est−elle plus rapide ?

Exemple 15.14. Sortie de `romantest82.py` avec `roman82.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s                                          ❶

OK                                                              ❷
```

❶ Dans l ensemble, les tests unitaires sont accélérés de 2% avec cette forme d expression régulière. Cela n a pas l air d être grand chose mais rappelez−vous que la fonction `search` n est qu une petite partie de l ensemble de nos tests unitaires, la plus grande partie du temps est passée à faire autre chose. (En testant séparément l expression régulière, j ai découvert que la fonction `search` est accélérée de `11%` avec cette

syntaxe.) En précompilant l expression régulière et en en récrivant une partie, nous avons amélioré la performance de l expression régulière de plus de 60% et amélioré la performance d ensemble des tests unitaires de plus de 10%.

❷ Plus important que tout bénéfice de performance est le fait que le module fonctionne encore parfaitement. C est là la liberté dont je parlais plus haut : la liberté d ajuster, de modifier ou de récrire n importe quelle partie et de vérifier que rien n a été endommagé durant ce processus. Ce n est pas une autorisation de fignoler indéfiniment le code pour le plaisir, nous avions un objectif spécifique (rendre `fromRoman` plus rapide) et nous avons rempli cet objectif sans que subsiste le doute d avoir introduit de nouveaux bogues.

Il y a une autre modification que j aimerais faire et ensuite je promet que j arrêterai de refactoriser ce module. Comme nous l avons vu de manière répétée, les expressions régulières peuvent devenir emberlificotées et illisibles assez vite. Je voudrais pouvoir revenir à ce module dans six mois et être capable de le maintenir. Bien sûr les cas de tests passent, je sais donc qu il fonctionne mais si je ne peux pas comprendre *comment* il fonctionne, je ne serai pas capable d ajouter des fonctionnalités, de corriger de nouveaux bogues et plus généralement de le maintenir. Comme nous l avons vu au Section 7.5, «Expressions régulières détaillées», Python fournit une manière de documenter vos expressions régulières ligne à ligne.

Exemple 15.15. `roman83.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage8/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

#new version
romanNumeralPattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    ''', re.VERBOSE) ❶
```

❶ La fonction `re.compile` peut prendre un second argument optionnel, un ensemble d un *flag* ou plus qui contrôle diverses options pour l expression régulière compilée. Ici, nous spécifions le *flag* `re.VERBOSE`, qui signale à Python qu il y a des commentaires à l intérieur de l expression régulière. Les commentaires ainsi que les espaces les entourant ne sont *pas* considérés comme faisant partie de l expression régulière, la fonction `re.compile` les enlève purement et simplement lorsqu elle compile l expression. Cette nouvelle version documentée est identique à l ancienne mais elle est beaucoup plus lisible.

Exemple 15.16. Sortie de `romantest83.py` avec `roman83.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s ❶
```

```
OK                      ❷
```

❶ Cette nouvelle version documentée s exécute exactement à la même vitesse que l ancienne. En fait, l objet motif compilé est le même, puisque la fonction `re.compile` supprime tout ce que nous avons ajouté.

❷ Cette nouvelle version passe tous les tests que passait l ancienne. Rien n a changé, sauf que le programmeur qui se penchera à nouveau sur ce module dans six mois aura une chance de comprendre le fonctionnement de la fonction.

15.4. Postscriptum

Un lecteur astucieux a lu la section précédente et l a amené au niveau supérieur. Le point le plus compliqué (et pesant le plus sur les performances) du programme tel qu il est écrit actuellement est l expression régulière, qui est nécessaire puisque nous n avons pas d autre moyen de subdiviser un nombre romain. Mais il n y a que 5000 nombres romains, pourquoi ne pas construire une table de référence une fois, puis simplement la lire ? Cette idée est encore meilleure quand on réalise qu il n y a pas besoin d utiliser les expressions régulière du tout. Au fur et à mesure que l on construit la table de référence pour convertir les entiers en nombres romains, on peut construire la table de référence inverse pour convertir les nombres romains en entiers.

Et le meilleur de tout, c est que nous avons déjà un jeu complet de tests unitaires. Le lecteur a modifié la moitié du code du module, mais les tests unitaires sont restés les mêmes, ce qui lui a permis de prouver que son code fonctionnait tout aussi bien que l original.

Exemple 15.17. `roman9.py`

Ce fichier est disponible dans le sous−répertoire `py/roman/stage9/` du répertoire des exemples.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
```

```
toRomanTable = [ None ]  # Skip an index since Roman numerals have no zero
fromRomanTable = {}

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "non-integers can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]

def toRomanDynamic(n):
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()
```

Alors, est-ce que c est rapide ?

Exemple 15.18. Sortie de `romantest9.py` avec `roman9.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 0.791s

OK
```

Rappelez-vous que la meilleure performance que nous avons obtenu dans la version originale était 13 tests en 3,315 secondes. Bien sûr, ce n est pas une comparaison entièrement juste, puisque cette version prendra plus de temps à importer (lorsqu elle remplit les tables de référence). Mais comme l importation n est faite qu une seule fois, c'est négligeable au bout du compte.

La morale de l histoire ?

- La simplicité est une vertu.
- Particulièrement avec les expressions régulières.
- Les tests unitaires vous donnent la confiance de conduire des refactorisations à grande échelle... même si vous n avez pas écrit le code originel.

15.5. Résumé

Les tests unitaires forment un concept puissant qui, s il est implémenté correctement, peut à la fois réduire les coûts de maintenance et augmenter la flexibilité d un projet à long terme. Il faut aussi comprendre que les tests unitaires ne sont pas une panacée, une baguette magique ou une balle d argent. Ecrire de bons cas de test est difficile et les tenir à jour demande de la discipline (surtout quand les clients réclament à hauts cris la correction de bogues critiques). Les tests unitaires ne sont pas destinés à remplacer d autres formes de tests comme les tests fonctionnels, les tests d intégration et les tests utilisateurs. Mais ils sont réalisables et ils marchent et une fois que vous les aurez vu marcher, vous vous demanderez comment vous avez pu vous en passer.

Ce chapitre a couvert un large sujet et une bonne partie n était pas spécifique à Python. Il y a des *frameworks* de tests unitaires pour de nombreux langages qui tous exigent que vous compreniez les mêmes concepts de base :

- Concevoir des cas de tests spécifiques, automatisés et indépendants
- Ecrire les cas de tests *avant* le code qu ils testent
- Ecrire des tests qui testent des entrées correctes et vérifient l obtention de résultats corrects
- Ecrire des tests qui testent des entrées incorrectes et vérifient qu un échec se produit
- Ecrire et mettre à jour des cas de test pour illustrer des bogues ou refléter des nouvelles spécifications
- Refactoriser en profondeur pour améliorer la performance, la montée en charge, la lisibilité, la facilité de maintenance ou tout autre facteur dont vous manquez

En plus, vous devez vous sentir à l aise pour des choses plus spécifiques à Python :

- Dériver `unittest.TestCase` et écrire des méthodes pour des cas de test individuels
- Utiliser `assertEqual` pour vérifier qu une fonction retourne une valeur identifiée
- Utiliser `assertRaises` pour vérifier qu une fonction déclenche une exception identifiée
- Appeler `unittest.main()` dans votre clause `if __name__` pour exécuter tous vos cas de tests en une fois
- Exécuter les tests unitaires en mode détaillé ou normal

Pour en savoir plus

- XProgramming.com (http://www.xprogramming.com/) a des liens pour télécharger des *frameworks* de tests unitaires (http://www.xprogramming.com/software.htm) pour de nombreux langages.

Chapitre 16. Programmation fonctionnelle

16.1. Plonger

Au Chapitre 13, *Tests unitaires*, vous avez appris la philosophie des tests unitaires. Au Chapitre 14, *Ecriture des tests en premier*, vous avez suivi pas à pas l'implémentation de tests unitaires en Python. Au Chapitre 15, *Refactorisation*, vous avez vu comment les tests unitaires facilitent la refactorisation à grand échelle. Ce chapitre va poursuivre le développement de ces programmes, mais cette fois en mettant l'accent sur des techniques avancées de Python plutôt que sur les test unitaires proprement dit.

Voici un programme Python complet qui remplit le rôle de *framework* pour les tests de régression. Il prend les tests unitaires que vous avez écrits pour chaque modules, les assemble en une seule suite de tests et les exécute en une seule fois. J'utilise ce script pendant la procédure de construction de ce livre, j'ai des tests unitaires pour plusieurs programmes d'exemple (pas seulement le module `roman.py` du Chapitre 13, *Tests unitaires*) et la première chose que mon script de construction automatique fait est d'exécuter ce programme pour être sûr que tous mes exemples fonctionnent encore. Si ce test de régression échoue, la construction s'arrête immédiatement. Je n'ai pas plus envie de publier un exemple qui ne marche pas que vous de le télécharger et de vous gratter la tête face à l'écran en vous demandant ce qui ne va pas.

Exemple 16.1. `regression.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

À l'exécution de ce script dans le même répertoire que le reste des scripts d'exemple de ce livre, l'ensemble des tests unitaires, nommés *`module`*`test.py`, sont trouvés, exécutés comme un seul test, réussissant ou échouant ensemble.

Exemple 16.2. Exemple de sortie de `regression.py`

```
[you@localhost py]$ python regression.py -v
help should fail with no object ... ok                                  ❶
help should return known result for apihelper ... ok
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok                ❷
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok                     ❸
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

----------------------------------------------------------------------
Ran 29 tests in 2.799s

OK
```

❶ Les 5 premiers tests viennent de `apihelpertest.py`, qui teste le script d'exemple du Chapitre 4, *Le pouvoir de l introspection*.

❷ Les 5 tests suivant viennent de `odbchelpertest.py`, qui teste le script d'exemple du Chapitre 2, *Votre premier programme Python*.

❸ Les tests restant viennent de `romantest.py`, que vous avez étudié en détail au Chapitre 13, *Tests unitaires*.

16.2. Trouver le chemin

Lorsque vous exécutez des scripts Python depuis la ligne de commande, il est parfois utile de savoir l'emplacement sur le disque du script en cours d'exécution.

C'est un de ces trucs obscurs qui sont pratiquement impossibles à deviner seul, mais qu'il est simple de se souvenir une fois que vous l'avez vu. L'élément-clé en est `sys.argv`. Comme vous l'avez vu au Chapitre 9, *Traitement de données XML*, c'est une liste qui contient la liste des arguments de ligne de commande. Mais elle contient également le nom du script en cours d'exécution comme il a été appelé depuis la ligne de commande et cela suffit à déterminer son emplacement.

Exemple 16.3. `fullpath.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython-examples-5.4.zip) du livre.

```
import sys, os

print 'sys.argv[0] =', sys.argv[0]              ❶
pathname = os.path.dirname(sys.argv[0])        ❷
print 'path =', pathname
print 'full path =', os.path.abspath(pathname) ❸
```

❶ Indépendamment de la manière dont vous exécutez un script, `sys.argv[0]` contient toujours le nom de ce script, exactement comme il apparaît sur la ligne de commande. Le chemin peut être inclus ou non, comme nous allons le voir.

❷ `os.path.dirname` prend un nom de fichier sous forme de chaîne et en retourne la partie qui représente le chemin. Si le nom de fichier ne contient pas d'information de chemin, `os.path.dirname` retourne une chaîne vide.

❸ `os.path.abspath`, l'élément−clé, prend un chemin, qui peut être partiel ou même vide et retourne un chemin complet.

`os.path.abspath` demande une explication plus détaillée. C'est une fonction très souple, qui peut prendre n'importe quel type de chemin en argument.

Exemple 16.4. Explication détaillée de `os.path.abspath`

```
>>> import os
>>> os.getcwd()                          ❶
/home/you
>>> os.path.abspath('')                  ❷
/home/you
>>> os.path.abspath('.ssh')              ❸
/home/you/.ssh
>>> os.path.abspath('/home/you/.ssh')    ❹
/home/you/.ssh
>>> os.path.abspath('.ssh/../foo/')      ❺
/home/you/foo
```

❶ `os.getcwd()` retourne le répertoire de travail en cours.

❷ L'appel de `os.path.abspath` avec une chaîne vide retourne le répertoire de travail en cours, comme `os.getcwd()`.

❸ L'appel de `os.path.abspath` avec un chemin partiel construit un chemin complet à partir de ce chemin partiel, basé sur le répertoire de travail en cours.

❹ L'appel de `os.path.abspath` avec un chemin complet le retourne simplement.

❺ `os.path.abspath` *normalise* également le chemin qu'il retourne. Notez que cet exemple marche alors que je n'ai pas de répertoire 'foo', `os.path.abspath` ne vérifie jamais votre disque, ce n'est que de la manipulation de chaînes.

Les chemins et noms de fichier que vous passez à `os.path.abspath` n'ont pas besoin d'exister sur le disque.

`os.path.abspath` ne construit pas seulement des chemins complets, il les normalise. Cela signifie que si vous êtes dans le répertoire `/usr/`, `os.path.abspath('bin/../local/bin')` retournera `/usr/local/bin`. Il normalise le chemin en le rendant aussi simple que possible. Si vous voulez normaliser un chemin de cette manière sans le transformer en chemin complet, utilisez `os.path.normpath`.

Exemple 16.5. Exemple de sortie de `fullpath.py`

```
[you@localhost py]$ python /home/you/diveintopython/common/py/fullpath.py ❶
sys.argv[0] = /home/you/diveintopython/common/py/fullpath.py
path = /home/you/diveintopython/common/py
```

```
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ python common/py/fullpath.py              ❷
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/you/diveintopython/common/py
[you@localhost diveintopython]$ cd common/py
[you@localhost py]$ python fullpath.py                                    ❸
sys.argv[0] = fullpath.py
path =
full path = /home/you/diveintopython/common/py
```

❶ Dans le premier cas, `sys.argv[0]` inclut le chemin complet du script. Vous pouvez ensuite utiliser la fonction `os.path.dirname` pour enlever le nom du script et retourner le chemin complet du répertoire, `os.path.abspath` retournera simplement ce que vous lui donnez.

❷ Si le script est exécuté avec un chemin partiel, `sys.argv[0]` contient exactement ce qui a été tapé en ligne de commande. `os.path.dirname` vous retournera un chemin partiel (relatif au répertoire en cours) et `os.path.abspath` construira le chemin complet à partir du chemin partiel.

❸ Si le script est exécuté depuis le répertoire en cours sans donner de chemin, `os.path.dirname` retournera simplement une chaîne vide. À partir d'une chaîne vide, `os.path.abspath` retourne le répertoire en cours, ce qui est bien la valeur recherchée puisque le script a été exécuté depuis le répertoire en cours.

Comme les autres fonctions des modules `os` et `os.path`, `os.path.abspath` est multiplate−forme. Les résultats que vous obtiendrez seront légèrement ifférents si vous utilisez Windows (qui utilise le le *backslash* comme séparateur de chemin) ou Mac OS (qui utilise les deux points), mais le script fonctionnera. C'est là le rôle du module `os`.

Addendum. Un lecteur ne s'est pas trouvé satisfait par cette solution, il voulait pouvoir exécuter tous les tests unitaires depuis le répertoire en cours, pas celui où `regression.py` est situé. Il suggère l'approche suivante :

Exemple 16.6. Exécuter les scripts dans le répertoire en cours

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       ❶
    sys.path.append(path)    ❷
    files = os.listdir(path) ❸
```

❶ Au lieu de changer le chemin vers le répertoire ou le script en cours d'exécution est situé, on lui assigne le répertoire en cours. C'est le répertoire dans lequel vous étiez quand vous avez lancé le script, qui n'est pas forcément le même que celui dans lequel se trouve le script (relisez cette phrase jusqu'à que vous ayez compris).

❷ Ce chemin est ajouté au chemin de recherche des bibliothèques de Python, de manière à ce que Python puisse trouver les modules de tests unitaires lorsque vous les importerez. Vous n'aviez pas besoin de le faire lorsque le chemin était le répertoire du script en cours d'exécution, car Python cherche toujours dans ce répertoire.

❸ Le reste de la fonction est inchangé.

Cette technique vous permettra de réutiliser ce script `regression.py` dans d'autres projets. Le script est mis dans un répertoire commun et vous l'exécutez depuis le répertoire du projet. Tous les tests unitaires de ce projet sont trouvés et exécutés, au lieu des tests unitaires du répertoire commun dans lequel est situé `regression.py`.

16.3. Le filtrage de liste revisité

Vous êtes déjà familier avec l'utilisation des *list comprehensions* pour le filtrage de listes. Il y a une autre manière de faire la même chose que certaines personnes trouvent plus expressive.

Python a une fonction `filter` prédéfinie qui prend deux arguments, une fonction et une liste et retourne une liste.[8] La fonction passée comme premier argument à `filter` doit elle−même prendre un argument et la liste que `filter` retournera contiendra tous les éléments de la liste passée en argument pour lesquels la fonction passée à `filter` retourne vrai.

Vous avez tout compris ? Ce n'est pas aussi difficile que ça en à l'air.

Exemple 16.7. Présentation de `filter`

```
>>> def odd(n):                    ❶
...     return n % 2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, −3]
>>> filter(odd, li)                ❷
[1, 3, 5, 9, −3]
>>> [e for e in li if odd(e)]      ❸
>>> filteredList = []
>>> for n in li:                   ❹
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, −3]
```

❶ `odd` utilise la fonction prédéfinie de modulo "%" pour retourner `True` si `n` est impair et `False` si `n` est pair.

❷ `filter` prend deux arguments, une fonction (`odd`) et une liste (`li`). `odd` est appelé avec chaque élément, au fur et à mesure que la liste est parcourue. Si `odd` retourne une valeur vraie (rappelez−vous qu'en Python, toute valeur différente de zéro est vraie), l'élément est inclut dans la liste retournée, sinon il est filtré. Le résultat est une liste comprenant uniquement les nombres impairs de la liste originelle, dans le même ordre qu'ils apparaissaient en entrée.

❸ Vous pouvez accomplir la même chose en utilisant les *list comprehensions*, comme vous l'avez vu à la Section 4.5, «Filtrage de listes».

❹ Vous pourriez aussi accomplir la même tâche avec une boucle `for`. Selon vos habitudes de programmation, cela peut vous sembler plus "logique", mais les fonctions comme `filter` sont bien plus expressives. Non seulement elles sont plus simples à écrire, elles sont aussi plus simple à lire. Lire la boucle `for` est comme regarder un tableau de trop près, on voit tous les détails, mais il faut quelques secondes pour comprendre l'ensemble : "Oh, c'est juste un filtrage de liste !"

Exemple 16.8. `filter` dans `regression.py`

```
    files = os.listdir(path)                                  ❶
    test = re.compile("test\.py$", re.IGNORECASE)             ❷
    files = filter(test.search, files)                        ❸
```

❶ Comme vous l'avez vu à la Section 16.2, «Trouver le chemin», `path` peut contenir le chemin complet ou partiel du répertoire du script en cours d'exécution, ou une chaîne vide si le script est exécuté depuis le répertoire en cours. De toute manière, `files` contiendra les noms des fichiers qui sont dans le même répertoire que le script en cours d'exécution.

❷ Ceci est une expression régulière compilée. Comme vous l'avez vu à la Section 15.3, «Refactorisation», si vous devez utiliser la même expression régulière de manière répétée, il vaut mieux la compiler pour obtenir de meilleures performances. L'objet compilé a une méthode `search` qui prend un seul argument, la chaîne à rechercher. Si l'expression régulière reconnaît la chaîne, la méthode `search` retourne un objet `Match` contenant des informations sur ce qu'elle a trouvé, sinon elle retourne `None`, la valeur nulle de Python.

❸ Pour chaque élément de la liste `files`, la méthode `search` de l'objet expression régulière compilée `test` va être appelée. Si l'expression régulière reconnaît la chaîne, la méthode retourne un objet `Match`, que Python considère comme ayant pour valeur vrai, donc l'élément sera inclus dans la liste retournée par `filter`. Si l'expression régulière ne reconnaît pas la chaîne, la méthode `search` retourne `None`, ce que Python considère comme ayant pour valeur faux et donc l'élément ne sera pas inclus.

Note historique. Les versions de Python antérieures à la version 2.0 n'avaient pas de *list comprehensions*, donc on ne pouvait pas filtrer avec des *list comprehensions*; la fonction `filter` était la seule possibilité. Malgré l'introduction des *list comprehensions* dans la version 2.0, certaines personnes préfèrent encore l'ancien style avec `filter` (et sa fonction associée, `map`, que vous verrez un peu plus loin dans ce chapitre). Les deux techniques fonctionnent actuellement, le de l'une ou l'autre est donc une question de style. Il est question de ne plus supporter `map` et `filter` dans les futures versions de Python, mais aucune décision n'a été prise pour le moment.

Exemple 16.9. Filtrage avec des *list comprehensions*

```
files = os.listdir(path)
test = re.compile("test\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] ❶
```

❶ On obtient exactement le même résultat qu'avec la fonction `filter`. Quelle est la manière la plus expressive ? À vous de le dire.

16.4. La mutation de liste revisitée

Vous avez déjà vu comment appliquer les *list comprehensions* aux mutations de listes. Il y a une autre manière d'obtenir la même chose en utilisant la fonction prédéfinie `map`. Elle fonctionne de manière similaire à la fonction `filter`.

Exemple 16.10. Présentation de `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> map(double, li)                          ❶
[2, 4, 6, 10, 18, 20, 512, -6]
>>> [double(n) for n in li]                  ❷
[2, 4, 6, 10, 18, 20, 512, -6]
>>> newlist = []
>>> for n in li:                             ❸
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, -6]
```

❶ `map` prend une fonction et une liste[9] et retourne une nouvelle liste en appelant la fonction avec chaque élément de la liste dans l'ordre. Dans ce cas, la fonction multiplie simplement chaque élément par 2.

❷ Vous pouvez accomplir la même tâche avec une *list comprehension*. Les *list comprehensions* ont été introduite pour la première fois dans Python 2.0, `map` en a toujours fait partie.

❸ Vous pouvez, si vous insistez pour penser comme un programmeur Visual Basic, utiliser une boucle `for` pour accomplir la même tâche.

Exemple 16.11. `map` avec des listes de types mélangés

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                    ❶
[10, 'aa', (2, 'b', 2, 'b')]
```

❶ Je précise que `map` fonctionne tout à fait avec des listes de types mélangés, tant que la fonction que vous utilisez est capable de traiter chaque type. Dans ce cas, la fonction `double` multiplie simplement l'argument qui lui est passé par 2 et Python A Le Bon Réflexe et agit en fonction du type de l'argument. Pour les entiers, cela veut dire les multiplier effectivement par 2, pour les chaînes, les concaténer avec elles−mêmes, pour les tuples, en créer un nouveau contenant deux fois la série d'éléments du précédent.

Mais passons maintenant à du véritable code.

Exemple 16.12. `map` dans `regression.py`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
moduleNames = map(filenameToModuleName, files)          ❷
```

❶ Comme vous l'avez vu dans la Section 4.7, «Utiliser des fonctions lambda», `lambda` définit une fonction incluse. Et comme vous l'avez vu dans l'Exemple 6.17, «Division de noms de chemins», `os.path.splitext` prend un nom de fichier et retourne un tuple (*`nom, extension`*). Donc `filenameToModuleName` est une fonction qui prend un nom de fichier et en supprime l'extension, ne retournant que le nom.

❷ L'appel de `map` prend chaque nom de fichier de `files`, le passe à la fonction `filenameToModuleName` et retourne une liste des valeurs de retour de chacun de ces appels de fonction. Autrement dit, on enlève l'extension de chaque nom de fichier et on stocke la liste de ces noms sans extension dans `moduleNames`.

Comme vous le verrez dans le reste de ce chapitre, vous pouvez étendre ce type d'approche centrée sur les données jusqu'au but final, qui est de définir et d'exécuter une suite de teste unique contenant les tests de toutes les suites individuelles.

16.5. Programmation centrée sur les données

Maintenant, vous vous demandez certainement pourquoi tout ça est mieux que d'utiliser des boucle `for` et de simples appels de fonction. C'est une question tout à fait justifiée. En fait, c'est avant tout une question de perspective, utiliser `map` et `filter` vous oblige à centrer votre réflexion sur les données.

Dans le cas présent, nous avons commencé absolument sans données, la première chose que nous avons faite est d'obtenir le chemin du répertoire du script en cours, puis une liste des fichiers de ce répertoire. Cette amorce nous a amené de véritables données avec lesquelles travailler : une liste de noms de fichiers.

Cependant, nous ne nous intéressons pas à tous ces fichiers, seulement à ceux qui sont des suites de tests. Nous avions *trop de données*, nous avions donc besoin de les filtrer. Comment savoir quelles données conserver ? Nous avions besoins d'un test pour le décider, nous en avons donc créé un que nous avons passé à la fonction `filter`. Dans le cas présent, nous avons utilisé une expression régulière, mais le concept serait le même quelle que soit la manière dont le test serait constitué.

Nous avions donc les noms de fichiers de chaque suite de tests (et seulement des suites de tests puisque le reste a été filtré), mais ce que nous voulions précisément c'était des noms de modules. Nous avions la quantité de données

correcte, mais elles étaient *dans un mauvais format*. Nous avons donc défini une fonction qui transformerait un nom de fichier en nom de module et l'avons appliquée à toute la liste. D'un nom de fichier, nous obtenons un nom de module, d'une liste de noms de fichiers, une liste de noms de modules.

À la place de `filter`, nous aurions pu utiliser une boucle `for` avec une instruction `if`. Au lieu de `map`, nous aurions plus utiliser une boucle `for` avec un appel de fonction. Mais utiliser des boucles de cette manière est un travail de tâcheron. Au mieux, c'est une perte de temps, au pire, on introduit des bogues difficile à détecter. Par exemple, il faut de toute manière définir la condition de test pour "ce fichier est−il une suite de test ?", c'est la logique spécifique de l'application et aucun langage ne va l'écrire à notre place. Mais une fois que l'on a résolu cette question, pourquoi s'épuiser à définir une nouvelle liste vide, écrire une boucle `for`, une instruction `if` et appeller manuellement `append` pour ajouter chaque élément à la nouvelle liste si il répond à la condition et ensuite garder trace de quelle variable contient la nouvelle liste et de celle qui contient l'ancienne ? Pourquoi ne pas nous contenter de définir la condition de test et laisser Python faire le reste du travail pour nous ?

Oh, bien sûr, vous pouvez montrer plus d'astuce et supprimer les éléments en place sans créer de nouvelle liste. Mais cela vous a déjà joué des tours. Essayer de modifier une structure de données que vous êtes en train de parcourir peut être périlleux. Vous supprimez un élément, bouclez sur le suivant et voilà que vous en avez sauté un. Est−ce que Python fait partie des langages qui fonctionnent de cette manière ? Combien de temps vous faudrait−il pour le découvrir ? Et est−ce que vous vous rappellerez si c'est sûr la prochaine fois que vous essayerez ? Les programmeurs passent tellement de temps et avec tellement d'erreurs, à se préoccuper de problème purement techniques de cet ordre, tout cela est inutile. Cela n'avance en rien votre programme, ce n'est que du travail de tâcheron.

J'étais réticent aux *list comprehensions* quand j'ai appris Python et j'ai résisté à `filter` et `map` encore plus longtemps. Je tenais à me rendre la vie plus difficile en me cantonnant à la manière familière des boucles `for` et des instructions `if`, à la programmation pas à pas, centrée sur le code. Et mes programmes Python ressemblaient beaucoup à des programmes Visual Basic, détaillant chaque étape de chaque opération dans chaque fonction. Et ils avaient tous les mêmes petits problèmes et les mêmes bogues difficiles à détecter. Et tout cela était inutile.

Laissons tout cela derrière nous. Le code de tâcheron n'est pas important. Les données sont importantes et les données ne sont pas compliquées. Ce ne sont que des données, s'il y en a trop, filtrez−les, si elles ne sont pas au bon format, transformez−les. Concentrez−vous sur les données, abandonnez le travail de tâcheron.

16.6. Importation dynamique de modules

Assez de discours philosophiques. Parlons plutôt de l'importation dynamique de modules.

D'abord, regardons à l'importation normale de modules. La syntaxe `import` *`module`* regarde dans le chemin de recherche si il y a un module portant ce nom et l'importe. Vous pouvez également importer plusieurs modules en une seule fois de cette manière, en les séparant par des virgules. Nous l'avons fait à la toute première ligne du script de ce chapitre.

Exemple 16.13. Importation de plusieurs modules à la fois

```
import sys, os, re, unittest ❶
```

❶ Cette instruction importe quatre modules à la fois : `sys` (pour les fonctions systèmes et l'accès aux paramètres de ligne de commande), `os` (pour les fonctions liées au système d'exploitation comme obtenir la liste de fichiers du répertoire), `re` (pour les expressions régulières) et `unittest` (pour les tests unitaires).

Maintenant, nous allons faire la même chose mais par importation dynamique.

Exemple 16.14. Importation dynamique de modules

```
>>> sys = __import__('sys')                    ❶
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys                                        ❷
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

❶ La fonction prédéfinie `__import__` remplit la même tâche que l'utilisation de l'instruction `import`, mais c'est une fonction qui prend une chaîne en argument.

❷ La variable `sys` est maintenant le module `sys`, comme si nous avions juste écrit `import sys`. La variable `os` est maintenant le module `os` etc.

Donc `__import__` importe un module, mais prend une chaîne en argument. Dans ce cas, le module que nous importons est une chaîne littérale, mais il pourrait tout au si bien s'agir d'une variable ou du résultat d'un appel de fonction. Et le nom de la variable à laquelle vous assignez le module n'a pas besoin de correspondre au nom du module. Nous pourrions importer une série de modules et les assigner à une liste.

Exemple 16.15. Importation d'une liste de modules

```
>>> moduleNames = ['sys', 'os', 're', 'unittest'] ❶
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames)        ❷
>>> modules                                       ❸
[<module 'sys' (built-in)>,
<module 'os' from 'c:\Python22\lib\os.pyc'>,
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version                            ❹
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
```

❶ `moduleNames` est juste une liste de chaînes. Rien de particulier si ce n'est que les chaînes ressemblent à des noms de modules que nous pourrions importer.

❷ Et voilà, nous voulions les importer et nous l'avons fait en appliquant la fonction `__import__` à la liste, ce qui prend chaque élément de la liste (`moduleNames`), le passe en argument à la fonction (`__import__`) et construit une nouvelle liste avec les valeurs retournées par cette fonction.

❸ ous avons donc construit une liste de modules à partir d'une liste de chaînes (Vos chemins peuvent être différents en fonction de votre système d'exploitation, de l'emplacement de votre installation Python etc.)

❹ Pour bien retenir qu'il s'agit de véritables modules, regardons quelques uns de leurs attributs. Rappelez−vous que `modules[0]` *est* le module `sys`, donc `modules[0].version` *est* `sys.version`. Tous les autres attributs et méthodes de ces modules sonts également disponibles. Il n'y a rien de magique à propos de l'instruction `import`, ni à propos des modules. Les modules sont des objets. Tout est objet.

Maintenant vous devez être en mesure d'assembler tout cela et de comprendre ce que fait la majeure partie du code d'exemple de ce chapitre.

16.7. Assembler les pièces

Vous en avez assez appris pour déconstruire les sept premières lignes du code d'exemple de ce chapitre : lire un répertoire et importer des modules sélectionnés parmi ceux qu'il contient.

Exemple 16.16. La fonction `regressionTest`

```
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
load = unittest.defaultTestLoader.loadTestsFromModule
return unittest.TestSuite(map(load, modules))
```

Regardons cela ligne par ligne. Supposons que le répertoire en cours est `c:\diveintopython\py`, qui contient les exemples du livre, y compris le script de ce chapitre. Comme vous l'avez vu à la Section 16.2, «Trouver le chemin», le répertoire du script est assigné à la variable `path`, commençons donc à cette étape.

Exemple 16.17. Etape 1 : Obtenir la liste des fichiers

```
>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files ❶
['BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py', 'apihelpertest.py',
'argecho.py', 'autosize.py', 'builddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pluraltest.py', 'pyfontify.py', 'regression.py', 'roman.py', 'romantest.py',
'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp', 'plural', 'roman',
'colorize.py']
```

❶ `files` est une liste de tous les fichiers et les répertoires du répertoire du script (si vous avez déjà exécuté certains exemples, vous verrez également des fichiers `.pyc`).

Exemple 16.18. Etape 2 : Filtrage des fichiers

```
>>> test = re.compile("test\.py$", re.IGNORECASE)          ❶
>>> files = filter(test.search, files)                     ❷
>>> files                                                  ❸
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py', 'pluraltest.py', 'romantest.py']
```

❶ Cette expression régulière reconnaît toutes les chaînes qui finissent par `test.py`. Notez que nous devons utiliser le caractère d'échappement pour le point, un point dans une expression régulière signifiant "n'importe quel caractère", ce que nous voulons c'est bien un point.

❷ L'expression régulière compilée agit comme une fonction, nous pouvons donc l'utiliser pour filtrer la liste de fichiers et de répertoires.

❸ Ce qu'il reste est la liste des scripts de tests unitaires puisque ce sont les seuls nommés `QUELQUECHOSEtest.py`.

Exemple 16.19. Etape 3 : Mutation des noms de fichiers en noms de modules

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
>>> filenameToModuleName('romantest.py')                    ❷
'romantest'
>>> filenameToModuleName('odchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)          ❸
>>> moduleNames                                             ❹
['apihelpertest', 'kgptest', 'odbchelpertest', 'pluraltest', 'romantest']
```

❶ Comme vous l'avez vu à la Section 4.7, «Utiliser des fonctions lambda», `lambda` est une manière rapide de créer des fonctions incluses d'une ligne. Celle−ci prend un nom de fichier avec une extension et le retourne sans son extension en utilisant la fonction de la bibliothèque standard `os.path.splitext` que vous avez vu à l'Exemple 6.17, «Division de noms de chemins».

❷ `filenameToModuleName` est une fonction. Il n'y a rien qui différencie les fonctions `lambda` des fonctions habituelles définies par l'instruction `def`. Nous pouvons appeler la fonction `filenameToModuleName` comme n'importe quelle autre et elle fait exactement ce que nous voulons qu'elle fasse : enlever l'extension du nom de fichier passé en argument.

❸ Maintenant nous pouvons appliquer cette fonction à chaque nom de fichier de la liste de fichier de tests unitaires à l'aide de `map`.

❹ Le résultat est bien ce que nous souhaitons : une liste de modules sous forme de chaînes.

Exemple 16.20. Etape 4 : Mutation des noms de modules en modules

```
>>> modules = map(__import__, moduleNames)                  ❶
>>> modules                                                 ❷
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>,
<module 'pluraltest' from 'pluraltest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]                                             ❸
<module 'romantest' from 'romantest.py'>
```

❶ Comme vous l'avez vu à la Section 16.6, «Importation dynamique de modules», nous pouvons utiliser `map` et `__import__` pour transformer une liste de noms de modules (sous forme de chaînes) en une liste de modules (que nous pouvons appeler comme n'importe quel autre module).

❷ `modules` est maintenant une liste de modules, totalement accessibles comme tout autre module.

❸ Le dernier module de la liste *est* le module `romantest`, comme si nous avions écrit `import romantest`.

Exemple 16.21. Etape 5 : Chargement des modules en une suite de tests

```
>>> load = unittest.defaultTestLoader.loadTestsFromModule
>>> map(load, modules)                                      ❶
[<unittest.TestSuite tests=[
  <unittest.TestSuite tests=[<apihelpertest.BadInput testMethod=testNoObject>]>,
  <unittest.TestSuite tests=[<apihelpertest.KnownValues testMethod=testApiHelper>]>,
  <unittest.TestSuite tests=[
    <apihelpertest.ParamChecks testMethod=testCollapse>,
    <apihelpertest.ParamChecks testMethod=testSpacing>]>,
    ...
  ]
]
>>> unittest.TestSuite(map(load, modules))                  ❷
```

❶ Ce sont de véritable objets−modules. Nous pouvons non seulement y accéder comme à tout autre module, instancier des classes et appeler des fonctions, nous pouvons également utiliser l'instrospection pour déterminer quelles fonctions et classes il contient. C'est ce que la méthode `loadTestsFromModule` fait : elle utilise l'instrospection et retourne un objet `unittest.TestSuite` pour chaque module. Chaque objet `TestSuite` contient en fait une liste d'objets `TestSuite`, un pour chaque classe `TestCase` du module et chacun de ces objets `TestSuite` contient une liste de tests, un pour chaque méthode de test du module.

❷ Finalement, nous regroupons la liste d'objets `TestSuite` en une seule suite de tests. Le module `unittest` n'a aucun mal à parcourir cet arbre de suites de tests imbriquées, il recherche une méthode de test, l'exécute, vérifie que le test passe ou échoue et continue de parcourir l'arbre jusqu'à la prochaine méthode de test.

Ce processus d'introspection est ce que le module `unittest` fait d'habitude pour nous. Vous vous rappelez de cette fonction magique `unittest.main()` que nos modules de test appelaient pour démarrer le processus ? `unittest.main()` crée en fait une instance de `unittest.TestProgram`, qui crée à son tour une instance de `unittest.defaultTestLoader` et le charge avec le module appelant (comment obtient−il une référence au module appelant sans qu'on lui en donne une ? En utilisant une instruction tout aussi magique, `__import__('__main__')`, qui importe dynamiquement le module en cours d'exécution. Je pourrais écrire un livre sur tous les trucs et les techniques utilisé dans le module `unittest`, mais dans ce cas je ne finirais jamais celui−ci).

Exemple 16.22. Etape 6 : Passage de la suite de tests à `unittest`

```
if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest") ❶
```

❶ Au lieu de laisser le module `unittest` opérer sa magie pour nous, nous avons fait la majeure partie du travail nous−même. Nous avons créé une fonction (`regressionTest`) qui importe les modules, appelé `unittest.defaultTestLoader` et regroupé l'ensemble en une suite de tests. Maintenant, tout ce dont nous avons besoins est de dire à `unittest` qu'il doit, au lieu de rechercher des tests et de construire une suite de tests de la manière habituelle, appeler simplement la fonction `regressionTest`, qui retourne une `TestSuite` prête à l'emploi.

16.8. Résumé

Le programme `regression.py` et sa sortie doivent maintenant être parfaitement clairs.

Vous devez maintenant être à l'aise avec les notions suivantes :

- Manipuler les information de chemin depuis la ligne de commande.
- Filtrer des listes à l'aide de `filter` au lieu des *list comprehensions*.
- Faire des mutations de listes à l'aide de `map` au lieu des *list comprehensions*.
- Importer des modules dynamiquement.

---

[8] Techniquement, le deuxième argument de `filter` peut être n'importe quelle séquence, y compris les listes, les tuples et les classes se comportant comme des séquences en définissant la méthode spéciale `__getitem__`. Si possible, `filter` retournera le même type de données que vous lui avez passé, donc filtrer une liste retourne une liste, mais filtrer un tuple retourne un tuple.

[9] Encore une fois, il faut préciser que `map` peut prendre une liste, un tuple ou un objet quelconque agissant comme une séquence. Voir la note précédente au sujet de `filter`.

Chapitre 17. Fonctions dynamiques

17.1. Plonger

Je vais vous parler du pluriel des noms (en anglais). Nous verrons ensuite les fonctions qui retournent d'autres fonctions, les expressions régulières avancées et les générateurs. Les générateurs sont une nouveauté de Python 2.3. Mais commençons par le pluriel des noms.

Si vous n'avez pas lu le Chapitre 7, *Expressions régulières*, c'est un bon moment pour le faire. Ce chapitre part du principe que vous comprenez les bases des expressions régulières et traite de questions plus avancées.

L'anglais est une langue complexe qui emprunte à de nombreuses autres langues, ses règles pour le pluriel des noms sont multiples et complexes. Il y a les règles, les exceptions à ces règles et les exceptions à ces exceptions.

Si vous avez grandi dans un pays de langue anglaise ou si vous avez appris l'anglais à l'école, vous connaissez sans doute les règles de base :

1. Si un mot se termine par S, X, ou Z, ajouter ES. "Bass" donne "basses", "fax" donne "faxes" et "waltz" donne "waltzes".
2. Si un mot se termine par un H prononcé, ajouter ES, si il se termine par un H silencieux, ajouter S. Qu'est−ce qu'un H prononcé ? C'est un H associé à d'autres lettres pour produire un son. Donc "coach" donne "coaches" et "rash" donne "rashes", car on prononce les sons CH et SH. Mais "cheetah" devient "cheetahs", car le H est silencieux.
3. Si un mot se termine par un Y prononcé comme I, remplacer le Y par IES, si le Y est associé à une voyelle pour donner un autre son, ajouter S. Donc "vacancy" donne "vacancies", mais "day" donne "days".
4. Si le mot ne répond à aucune de ces règles, ajouter un S et prier pour que ce soit juste.

(Je sais qu'il y a beaucoup d'exceptions. "Man" donne "men" et "woman" donne "women", mais "human" donne "humans". "Mouse" donne "mice" et "louse" donne "lice", mais "house" donne "houses". "Knife" donne "knives" et "wife" donne "wives", mais "lowlife" donne "lowlifes". Sans parler des mots qui sont invariables, comme "sheep", "deer" et "haiku".)

Les autres langues ont, bien sûr, des règles complètement différentes.

Nous allons concevoir un module qui met les noms au pluriel. Nous commencerons par l'anglais, avec ce quatre règles, mais gardez à l'esprit qu'il faudra inévitablement ajouter d'autres règles et peut−être d'autres langues.

17.2. `plural.py`, étape 1

Nous avons donc des mots, qui, en anglais du moins, sont constitués de chaînes de caractères. Par ailleurs, nous avons des règles qui disent que nous devons reconnaître différentes combinaisons de caractères, et leur faire subir certaines modifications. C'est un problème qui semble être fait pour les expressions régulières.

Exemple 17.1. `plural1.py`

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):             ❶
        return re.sub('$', 'es', noun)        ❷
    elif re.search('[^aeioudgkprt]h$', noun):
```

```
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

❶ C'est une expression régulière, mais elle utilise une syntaxe que vous n'avez pas vue au Chapitre 7, *Expressions régulières*. Les crochets signifient "reconnaître exactement un de ces caractères". Donc `[sxz]` signifie "`s` ou `x` ou `z`", mais seulement l'une de ces trois lettres. Le `$` doit vous être familier, il reconnaît la fin de la chaîne. Il s'agit donc de vérifier si `noun` se termine par `s`, `x` ou `z`.

❷ La fonction `re.sub` effectue des remplacements à partir d'une expression régulière. Examinons−la en détail.

Exemple 17.2. Présentation de `re.sub`

```
>>> import re
>>> re.search('[abc]', 'Mark')  ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark') ❷
'Mork'
>>> re.sub('[abc]', 'o', 'rock') ❸
'rook'
>>> re.sub('[abc]', 'o', 'caps') ❹
'oops'
```

❶ La chaîne `Mark` contient−elle `a`, `b` ou `c` ? Oui, elle contient `a`.

❷ Maintenant, recherchons `a`, `b` ou `c`, et remplaçons−le par `o`. `Mark` devient `Mork`.

❸ La même fonction transforme `rock` en `rook`.

❹ Vous auriez pu croire qu'elle transformerait `caps` en `oaps`, mais ce n'est pas le cas. `re.sub` remplace *tout* ce qui a été reconnu, pas seulement la première occurrence. Cette expression régulière transforme donc `caps` en `oops`, aussi bien le `c` que le `a` sont remplacés par `o`.

Exemple 17.3. Retour à `plural1.py`

```
import re

def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun)        ❶
    elif re.search('[^aeioudgkprt]h$', noun): ❷
        return re.sub('$', 'es', noun)        ❸
    elif re.search('[^aeiou]y$', noun):
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

❶ Retour à la fonction `plural`. Que faisons nous ? Nous remplaçons la fin de chaîne par `es`. En d'autre termes, nous ajoutons `es` à la chaîne. Nous pourrions accomplir la même chose avec une concaténation, par exemple `noun + 'es'`, mais j'utilise les expressions régulières pour tous les cas pour des raisons de cohérence, raisons qui deviendront plus claires plus loin dans le chapitre.

❷ Regardez attentivement, c'est encore une nouvelle variation. Le `^` comme premier caractère entre les crochets signifie quelque chose de spécial : la négation. `[^abc]` signifie "n'importe quel caractère unique *sauf* `a`, `b` ou `c`". Donc `[^aeioudgkprt]` signifie n'importe quel caractère sauf `a`, `e`, `i`, `o`, `u`, `d`, `g`, `k`, `p`, `r` ou `t`. Ensuite, ce caractère doit être suivi de `h`, suivi par la fin de chaîne. Nous cherchons les mots qui finissent par H dans lesquels le H est prononcé.

❸ Même motif ici : reconnaître les mots qui finissent par Y, dans lesquels le caractère qui précède Y n'est *pas* `a`, `e`, `i`, `o` ou `u`. Nous recherchons des mots qui se finissent par un Y prononcé comme I.

Exemple 17.4. Expressions régulières avec négation

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy') ❶
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy')     ❷
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita')    ❸
>>>
```

❶ `vacancy` est reconnu par cette expression régulière parce qu'il se termine par `cy`, `c` n'est pas `a`, `e`, `i`, `o` ou `u`.

❷ `boy` n'est pas reconnu, il se termine par `oy` et nous avons spécifié que le caractère précédent le `y` ne pouvait pas être `o`. `day` n'est pas reconnu car il se termine par `ay`.

❸ `pita` n'est pas reconnu parce qu'il ne se termine pas par `y`.

Exemple 17.5. Fonctionnement de `re.sub`

```
>>> re.sub('y$', 'ies', 'vacancy')                ❶
'vacancies'
>>> re.sub('y$', 'ies', 'agency')
'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy') ❷
'vacancies'
```

❶ Cette expression régulière transforme `vacancy` en `vacancies` et `agency` en `agencies`, ce qui est le but recherché. Notez qu'elle transformerait également `boy` en `boies`, mais cela n'arrivera jamais puisque nous faison d'abord le `re.search` pour savoir si nous devons effectuer le `re.sub`.

❷ Juste au passage, je veux souligner qu'il est possible de combiner ces deux expressions régulières (une pour savoir si la règle s'applique, l'autre pour l'appliquer effectivement) en une seule expression régulière. Voilà à quoi ça ressemblerait. Elle devrait être familière : elle utilise un groupe mémorisé, ce que vous avez appris à la Section 7.6, «Etude de cas : reconnaissance de numéros de téléphone», pour se rappeler du caractère avant le `y`. Ensuite, dans la chaîne de substitution, elle utilise une nouvelle syntaxe, `\1`, qui signifie "hep, ce premier groupe que tu as mémorisé, met le ici". Dans ce cas, elle a mémorisé le `c` devant le `y` et donc elle substitue un `c` au `c` et `ies` à `y` (si on a plus d'un groupe mémorisé, on utilise `\2` et `\3` etc.).

Les remplacements par expressions régulières sont extrêmement puissants et la syntaxe `\1` les rend encore plus puissants. Mais combiner l'opération entière en une seule expression régulière est également beaucoup plus difficile à lire et ne correspond pas à la manière dont les règles de pluriel des noms ont été définies au début. Les règles se présente de la manière suivante : "si le mot se termine par S, X ou Z, alors ajouter ES". Si vous regardez cette fonction, vous verrez deux lignes de codes qui disent "si le mot se termine par S, X ou Z, alors ajouter ES". Difficile de trouver une correspondance plus directe.

17.3. `plural.py`, étape 2

Maintenant, nous allons ajouter un niveau d'abstraction. Nous avons commencé par définir une liste de règles : si telle condition est remplie, alors effectuer telle action, sinon passer à la règle suivante. Nous allons temporairement rendre plus complexe une partie du programme pour pouvoir en simplifier une autre.

Exemple 17.6. `plural2.py`

```
import re

def match_sxz(noun):
    return re.search('[sxz]$', noun)

def apply_sxz(noun):
    return re.sub('$', 'es', noun)

def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)

def apply_h(noun):
    return re.sub('$', 'es', noun)

def match_y(noun):
    return re.search('[^aeiou]y$', noun)

def apply_y(noun):
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return 1

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
         )                                      ❶

def plural(noun):
    for matchesRule, applyRule in rules:        ❷
        if matchesRule(noun):                   ❸
            return applyRule(noun)              ❹
```

❶ Cette version a l'air plus compliqué (en tout cas, elle est certainement plus longue), mais elle fait exactement la même chose : elle tente de reconnaître les mêmes quatre règles, dans l'ordre, et applique l'expression régulière appropriée lorsqu'un motif est reconnu. La différence est que chaque règle de recherche et de modification est définie dans sa propre fonction et que ces fonctions sont assemblées et assignées à la variable `rules`, qui est un tuple de tuples.

❷ À l'aide d'une boucle `for`, nous pouvons appliquer deux règles à la fois (une de recherche et une de transformation) à partir du tuple `rules`. À la première itération de la boucle `for` loop, `matchesRule` référencera `match_sxz` et `applyRule` `apply_sxz`. À la seconde itération (si il y en a une), `matchesRule` référencera `match_h` et `applyRule` `apply_h`.

❸ Rappelez vous que tout est objet en Python, y compris les fonctions. `rules` contient des fonctions, pas seulement des noms de fonctions. Lorsqu'elles sont assignées dans la boucle `for`, les variables `matchesRule` et `applyRule` sont des fonctions que vous pouvez appeler. Donc, à la première itération de la boucle `for`, cette ligne est l'équivalent d'un appel à `matches_sxz(noun)`.

❹ À la première itération de la boucle `for`, ceci est l'équivalent d'un appel à `apply_sxz(noun)`, etc

Si ce niveau d'abstraction vous semble obscur, essayer de déplier les fonctions. Cette boucle `for` est l'équivalent de ce qui suit :

Exemple 17.7. La fonction `plural` dépliée

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

L'avantage ici est que la fonction `plural` est simplifiée. Elle prend une liste de règles, définie ailleurs, et les parcours de manière générique. On prend une règle de recherche, la chaîne est−elle reconnue ? Alors on appelle la règle de transformation. Les règles pourraient être définies à n'importe quel endroit et de n'importe quelle manière, la fonction `plural` ne s'en occupe pas.

Maintenant, est−ce que l'ajout de ce niveau d'abstraction en valait la peine ? Et bien, pas pour le moment. Considérons ce qu'il faudrait faire pour ajouter une nouvelle règle. Dans la version précédente, il aurait fallu ajouter une instruction `if` à la fonction `plural`. Dans cette version, il faut ajouter deux fonctions, `match_foo` et `apply_foo`, et mettre à jour la liste de règles `rules` pour spécifier entre quelles autres règles la nouvelle règle doit être appelée.

Cette étape n'est qu'une base pour la prochaine section, continuons donc.

17.4. `plural.py`, étape 3

Définir de fonctions séparément pour chaque règle de recherche et de transformation n'est pas vraiment nécessaire. Nous ne les appelons jamais séparément, elles sont définies dans la liste de règles `rules` et appelées à partir de cette liste. Nous allons simplifier la définition des règles en rendant ces fonctions anonymes.

Exemple 17.8. `plural3.py`

```
import re

rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )                                          ❶

def plural(noun):
    for matchesRule, applyRule in rules:      ❷
        if matchesRule(noun):
```

```
        return applyRule(noun)
```

❶ C'est le même ensemble de règles qu'à l'étape 2. La seule différence est qu'au lieu de définir des fonctions nommées comme `match_sxz` et `apply_sxz`, nous avons "inclus" ces définitions directement dans la liste `rules`, à l'aide de fonctions lambda.

❷ Notez que la fonction `plural` n'a pas changé. Elle parcourt l'ensemble des fonctions de règles, vérifie la première règle et, si celle−ci retourne vrai, appelle la seconde règle et retourne sa valeur. C'est la même fonction qu'à l'étape précédent, mot pour mot. La seule différence est dans la définition des fonctions de règles, mais la fonction `plural` ne s'en occupe pas, elle sait qu'on lui fournit une liste de règles et elle les applique aveuglément.

Maintenant, pour ajouter une nouvelle règle, il suffit de définir les fonctions directement dans la liste `rules` : une règle de recherche et une règle de transformation. Mais définir les fonctions de règles de cette manière met en valeur une duplication inutile. Nous avons quatre paires de fonctions qui suivent toutes le même motif. La fonction de recherche est un simple appel à `re.search` et la fonction de transformation un simple appel à `re.sub`. Nous allons extraire ces similarités.

17.5. `plural.py`, étape 4

Nous allons extraire la duplication de code pour rendre plus facile la définition de nouvelles règles.

Exemple 17.9. `plural4.py`

```
import re

def buildMatchAndApplyFunctions((pattern, search, replace)):
    matchFunction = lambda word: re.search(pattern, word)      ❶
    applyFunction = lambda word: re.sub(search, replace, word) ❷
    return (matchFunction, applyFunction)                      ❸
```

❶ `buildMatchAndApplyFunctions` est une fonction qui construit d'autres fonctions dynamiquement. Elle prend en argument `pattern`, `search` et `replace` (en fait elle prend un tuple, mais nous expliquerons cela plus loin) et elle construit la fonction de recherche à l'aide de `lambda` comme une fonction prenant un argument (`word`) et appelant `re.search` avec le motif `pattern` passé à `buildMatchAndApplyFunctions` et l'argument `word` qui lui a été passé. Ouf !

❷ La construction de la fonction de transformation se fait de la même manière. C'est une fonction qui prend un argument et appelle `re.sub` avec les arguments `search` et `replace` passés à `buildMatchAndApplyFunctions`, ainsi que cet argument `word`. Cette technique consistant à utiliser les valeurs d'arguments externes est appelée *fermeture*. Nous définissons en fait des constantes dans la fonction de transformation que nous construisons : elle prend un argument (`word`), mais agit ensuite sur cet argument et sur deux autres valeurs (`search` et `replace`) qui ont été définies lorsque la fonction de transformation a été créée.

❸ Finalement, la fonction `buildMatchAndApplyFunctions` retourne un tuple de deux valeurs : les deux fonctions qui viennent d'être créées. Les constantes définies dans ces deux fonctions (`pattern` dans `matchFunction`, `search` et `replace` dans `applyFunction`) sont conservées avec ces fonctions, y compris après le retour de l'appel à `buildMatchAndApplyFunctions`. C'est incroyablement puissant.

Si cela semble totalement confus (et ça l'est certainement, c'est très inhabituel), la manière dont cette fonction est utilisée devrait éclaircir ces définitions.

Exemple 17.10. `plural4.py`, suite

```
patterns = \
  (
    ('[sxz]$', '$', 'es'),
    ('[^aeioudgkprt]h$', '$', 'es'),
    ('(qu|[^aeiou])y$', 'y$', 'ies'),
    ('$', '$', 's')
  )                                                    ❶
rules = map(buildMatchAndApplyFunctions, patterns)     ❷
```

❶ Nos règles de pluriel des noms sont maintenant définies comme une série de chaînes (pas de fonctions). La première chaîne est l'expression régulière que l'on utilise avec `re.search` pour vérifier si la règle s'applique, les deuxième et troisième sont les expressions de recherche et de remplacement que l'on utilise avec `re.sub` pour appliquer effectivement la règle.

❷ Cette ligne est magique. Elle prend la liste de chaînes de `patterns` et la transforme en une liste de fonctions. Comment ? En appliquant les chaînes à la fonction `buildMatchAndApplyFunctions`, qui prend trois chaînes en argument et retourne un tuple de deux fonctions. Cela veut dire que `rules` contient exactement la même chose que dans la version précédente : une liste de tuples de deux fonctions, la première étant la fonction de recherche qui appelle `re.search` et la seconde la fonction de transformation qui appelle `re.sub`.

Je vous assure que c'est vrai : `rules` contient exactement la même liste de fonction que dans la version précédente. Déplions la définition de `rules`, nous obtenons ce qui suit :

Exemple 17.11. La définition de `rules` dépliée

```
rules = \
  (
    (
     lambda word: re.search('[sxz]$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeioudgkprt]h$', word),
     lambda word: re.sub('$', 'es', word)
    ),
    (
     lambda word: re.search('[^aeiou]y$', word),
     lambda word: re.sub('y$', 'ies', word)
    ),
    (
     lambda word: re.search('$', word),
     lambda word: re.sub('$', 's', word)
    )
   )
```

Exemple 17.12. `plural4.py`, suite et fin

```
def plural(noun):
    for matchesRule, applyRule in rules:           ❶
        if matchesRule(noun):
            return applyRule(noun)
```

❶ Puisque la liste `rules` est la même que dans la version précédente, il n'est pas étonnant que la fonction `plural` ne soit pas modifiée. Rappelez−vous qu'elle est totalement générique, elle prend une liste de fonctions de règle et les appelles dans l'ordre. Elle ne s'occupe pas de la manière dont ces règles sont définies. À l'étape 2, elles étaient définies comme des fonctions nommées indépendantes. À l'étape 3, elles étaient définies comme des fonctions anonymes `lambda`. Maintenant, à l'étape 4, elles sont construites dynamiquement par la fonction

`buildMatchAndApplyFunctions` à partir d'une liste de chaînes. De toute manière, la fonction `plural` agit de la même façon.

Au cas où tout cela ne suffirait pas à vous tourner la tête, je dois ajouter qu'il y a une petite subtilité dans la définition de `buildMatchAndApplyFunctions` que je n'ai pas expliquée. Regardons à nouveau cette définition.

Exemple 17.13. Retour sur `buildMatchAndApplyFunctions`

```
def buildMatchAndApplyFunctions((pattern, search, replace)):  ❶
```

❶ Avez−vous remarqué les doubles parenthèses ? Cette fonction ne prend en fait pas trois arguments mais un seul : un tuple de trois éléments. Mais le tuple est développé lorsque la fonction est appelée et les trois éléments sont assignés à trois variables différentes : `pattern`, `search` et `replace`. Vous vous demandez pourquoi ? Voyons cela en détail

Exemple 17.14. Développement des tuples à l'appel de fonctions

```
>>> def foo((a, b, c)):
...     print c
...     print b
...     print a
>>> parameters = ('apple', 'bear', 'catnap')
>>> foo(parameters) ❶
catnap
bear
apple
```

❶ L'appel à la fonction `foo` se fait avec un tuple de trois éléments. Lorsque la fonction est appelée, les éléments sont assignés à trois variables locales à `foo`.

Maintenant, examinons pourquoi il est nécessaire d'utiliser le développement automatique de tuple. `patterns` est une liste de tuples, chacun ayant trois éléments. Lorsque nous appelons `map(buildMatchAndApplyFunctions, patterns)`, cela signifie que `buildMatchAndApplyFunctions` n'est *pas* appelé avec trois arguments. Utiliser `map` pour appliquer une liste à une fonction appelle cette fonction avec à chaque fois un seul paramètre : chacun des éléments de la liste. Dans le cas de `patterns`, chaque élément de la liste est un tuple, donc `buildMatchAndApplyFunctions` est à chaque fois appelé avec le tuple et nous utilisons le développement automatique de tuple dans la définition de `buildMatchAndApplyFunctions` pour assigner les éléments de ce tuple à des variables locales avec lesquelles nous pouvons travailler.

17.6. `plural.py`, étape 5

Nous avons extrait toute duplication de code et ajouter assez d'abstraction pour que les règles de pluriel des noms soient définies sous forme d'une liste de chaînes. La prochaine étape est logiquement de mettre ces chaînes dans un fichier séparé, pour qu'elles puissent être modifiées séparément du code qui les utilise.

D'abord, nous allons créer un fichier texte qui contient les règles. Ici, pas de structures de données sophistiquées, seulement des chaînes délimitées par des espaces (ou des tabulations) en trois colonnes. Nous l'appelons `rules.en`, "en" pour English. Ce sont les règles du pluriel des noms pour l'anglais. Nous pourrons ajouter d'autres fichiers de règles pour d'autre langues plus tard.

Exemple 17.15. `rules.en`

```
[sxz]$                  $               es
[^aeioudgkprt]h$        $               es
[^aeiou]y$              y$              ies
$                       $               s
```

Maintenant, voyons comment utiliser ce fichier de règles.

Exemple 17.16. `plural5.py`

```
import re
import string

def buildRule((pattern, search, replace)):
    return lambda word: re.search(pattern, word) and re.sub(search, replace, word) ❶

def plural(noun, language='en'):                                    ❷
    lines = file('rules.%s' % language).readlines()                 ❸
    patterns = map(string.split, lines)                             ❹
    rules = map(buildRule, patterns)                                ❺
    for rule in rules:
        result = rule(noun)                                         ❻
        if result: return result
```

❶ Nous utilisons encore la technique des fermetures (construire dynamiquement une fonction qui utilise des variables définies à l'extérieur de cette fonction), mais maintenant nous avons combiné les fonctions de recherche et de transformation en une seule fonction (la raison de cette modification apparaîtra à la prochaine section). Cela nous permettra de faire la même chose qu'avec les deux fonctions, mais l'appel en sera différent, comme nous allons le voir.

❷ Notre fonction `plural` prend maintenant un second argument optionnel, `language`, qui vaut `en` par défaut.

❸ Nous utilisons l'argument `language` pour construire un nom de fichier, puis nous ouvrons ce fichier et copions sont contenu dans une liste. Si le langage est `en`, nous allons donc : ouvrir `rules.en`, le lire en entier, le segmenter à chaque retour à la ligne et retourner une liste. Chaque ligne du fichier est un élément de la liste.

❹ Comme vous l'avez vu, chaque ligne du fichier contient en fait trois valeurs, séparées par des espaces (espaces ou tabulations ne font pas de différence). En appliquant la fonction `string.split` à la liste, nous créons une nouvelle liste dans laquelle chaque élément est un tuple de trois chaînes. Donc, une ligne comme `[sxz]$ $ es` sera segmentée en un tuple `('[sxz]$', '$', 'es')`. Cela signifie que `patterns` contient une liste de tuples, comme nous l'avions fait à la main à l'étape 4.

❺ Si `patterns` est une liste de tuples, alors `rules` sera une liste de fonctions créées dynamiquement par chaque appel à `buildRule`. L'appel à Calling `buildRule(('[sxz]$', '$', 'es'))` retourne une fonction qui prend un seul argument, `word`. Quand cette fonction retournée est appelée, elle exécute `re.search('[sxz]$', word) et re.sub('$', 'es', word)`.

❻ Comme nous construisons maintenant une fonction combinée de recherche et de transformation, nous devons l'appeler différemment. Nous appelons simplement cette fonction et si elle retourne quelque chose, c'est le pluriel du nom, si elle ne retourne rien (`None`), alors la règle ne s'applique pas et il faut essayer la règle suivante.

L'amélioration ici est que nous avons complètement séparé les règles de pluriel des noms dans un fichier externe. Non seulement ce fichier peut−être maintenu séparément du code, mais nous avons défini une règle de nommage de fichier grâce à laquelle la même fonction `plural` peut utiliser des fichiers de règles différents en fonction d'un argument `language`.

L'inconvénient est que nous lisons le fichier à chaque fois que nous appelons la fonction `plural`. Je pensais pouvoir finir ce livre sans utiliser la phrase "laissé en exercice au lecteur", mais c'est raté : le développement d'un mécanisme de cache pour les fichiers de règles qui se rafraîchisse automatiquement lorsque un fichier de règle est modifié entre

deux appels est *laissé en exercice au lecteur*. Amusez−vous bien.

17.7. `plural.py`, étape 6

Maintenant, vous êtes prêts à une discussion sur les générateurs.

Exemple 17.17. `plural6.py`

```
import re

def rules(language):
    for line in file('rules.%s' % language):
        pattern, search, replace = line.split()
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word)

def plural(noun, language='en'):
    for applyRule in rules(language):
        result = applyRule(noun)
        if result: return result
```

Nous utilisons ici une technique appelée un générateur, que je ne vais même pas tenter d'expliquer avant de vous montrer un exemple plus simple.

Exemple 17.18. Présentation des générateurs

```
>>> def make_counter(x):
...     print 'entering make_counter'
...     while 1:
...         yield x                 ❶
...         print 'incrementing x'
...         x = x + 1
...
>>> counter = make_counter(2) ❷
>>> counter                   ❸
<generator object at 0x001C9C10>
>>> counter.next()            ❹
entering make_counter
2
>>> counter.next()            ❺
incrementing x
3
>>> counter.next()            ❻
incrementing x
4
```

❶ La présence du mot−clé `yield` dans `make_counter` signale qu'il ne s'agit pas d'une fonction ordinaire. C'est un genre de fonction spécial qui génère des valeurs une par une. Vous pouvez considérer cela comme une fonction qui reprend sont activité là où elle l'a laissée. L'appeler retourne un générateur qui peut être utilisée pour générer des valeurs successives de `x`.

❷ Pour créer une instance du générateur `make_counter`, il suffit de l'appeler comme toute autre fonction. Notez que cela n'éxécute pas le code de la fonction, cela se voit au fait que la première ligne de `make_counter` est une instruction `print`, mais que rien n'est encore affiché.

❸ La fonction `make_counter` retourne un objet générateur.

❹

La première fois que vous appelez la méthode `next()` de l'objet générateur, elle exécute le code de `make_counter` jusqu'à la première instruction `yield`, puis retourne la valeur produite par `yield`. Dans ce cas, il s'agit de `2`, puisque nous avons créé le générateur par l'appel `make_counter(2)`.

❺ À chaque appel successif à `next()`, l'objet générateur *reprend l'exécution au point où il l'avait laissé* et continue jusqu'à l'instruction `yield` suivante. Les lignes de code attendant d'être exécutées sont l'instruction `print` qui affiche `incrementing x`, puis l'instruction `x = x + 1` qui incrémente la variable. Ensuite, on boucle le `while` et on revient à l'instruction `yield x`, ce qui retourne la valeur actuelle de `x` (maintenant égale à 3).

❻ La seconde fois que nous appelons `counter.next()`, nous faisons à nouveau la même chose, mais cette fois `x` vaut `4`. Et cela continue de la même manière. Comme `make_counter` définit une boucle infinie, nous pourrions théoriquement continuer pour l'éternité, le générateur continuerait d'incrémenter `x` et de produire sa valeur. Mais nous allons examiner un exemple plus productif de l'utilisation des générateurs.

Exemple 17.19. Utilisation des générateurs à la place de la récursion

```
def fibonacci(max):
    a, b = 0, 1          ❶
    while a < max:
        yield a          ❷
        a, b = b, a+b    ❸
```

❶ La suite de Fibonacci est une séquence de nombres dans laquelle chaque nombre est la somme des deux nombres qui le précèdent. Elle commence par `0` et `1`, augmente doucement au début, puis de plus en plus vite. Pour commencer la séquence, nous utilisons deux variables : `a` commence à `0` et `b` à `1`.

❷ `a` est le nombre en cours dans la séquence, donc nous le produisons par `yield`.

❸ `b` est le nombre suivant dans la séquence, nous l'assignons donc à `a`, mais nous calculons également la prochaine valeur (`a+b`) et l'assignons à `b` pour l'utiliser au prochain appel. Notez que les assignements sont faits en parallèle, si `a` vaut `3` et `b` vaut `5`, alors `a, b = b, a+b` mettra `a` à `5` (la valeur précédente de `b`) et `b` à `8` (la somme des valeurs précédentes de `a` et `b`).

Donc, nous obtenons une fonction qui génère les nombres de Fibonacci. Bien sûr, vous pourriez le faire par récursion, mais cette manière est beaucoup plus simple à lire. De plus, elle fonctionne bien avec une boucle `for`.

Exemple 17.20. Les générateurs dans des boucles `for`

```
>>> for n in fibonacci(1000): ❶
...     print n,               ❷
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

❶ Nous pouvons utiliser un générateur comme `fibonacci` dans une boucle `for` directement. La boucle `for` crée l'objet générateur et appelle successivement la méthode `next()` pour obtenir des valeurs à assigner à la variable d'index de boucle (`n`).

❷ À chaque parcours de la boucle `for` loop, `n` obtient une nouvelle valeur de l'instruction `yield` de `fibonacci` et tout ce que nous faisons est de l'afficher. Une fois que `fibonacci` n'a plus de valeur à retourner (`a` est plus grand que `max`, dans ce cas que `1000`), alors la boucle `for` s'achève simplement.

Revenons maintenant à notre fonction `plural` et voyons l'usage que nous faisons de tout cela.

Exemple 17.21. Les générateurs pour produire des fonctions dynamiques

```
def rules(language):
```

```
    for line in file('rules.%s' % language):                                    ❶
        pattern, search, replace = line.split()                                 ❷
        yield lambda word: re.search(pattern, word) and re.sub(search, replace, word) ❸

def plural(noun, language='en'):
    for applyRule in rules(language):  ❹
        result = applyRule(noun)
        if result: return result
```

❶ `for line in file(...)` est un idiome habituel pour lire le contenu d'un fichier ligne par ligne. Cela fonctionne parce que *`file` renvoit en fait un générateur* dont la méthode `next()` retourne la ligne suivante du fichier. Personnellement, je trouve ça absolument génial.

❷ Rien de magique ici. Rappelez−vous que les lignes du fichier de règles contiennent trois valeurs séparées par des espaces, `line.split()` retourne donc un tuple de trois valeurs qui sont assignées à trois variables locales.

❸ *Ici, nous utilisons yield.* Qu'est−ce que nous produisons ? Une fonction construite dynamiquement avec `lambda`, qui est en fait une fermeture (elle utilise les variables locales `pattern`, `search` et `replace` comme constantes). Autrement dit, `rules` est un générateur de fonctions de règles.

❹ Comme `rules` est un générateur, nous pouvons l'utiliser directement dans une boucle `for`. À la première itération à travers la boucle, nous appelons la fonction `rules`, qui ouvre le fichier de règles, en lit la première ligne, construit dynamiquement une fonction de recherche et de transformation pour la première règle définie dans le fichier et produit la fonction construite dynamiquement. À la seconde itération, nous reprenons `rules` au point où nous l'avons laissé (c'est à dire au milieu de la boucle `for line in file(...)`), qui lit la seconde ligne, construit dynamiquement une nouvelle fonction de recherche et de transformation pour la seconde règle et la produit. Et ainsi de suite.

Qu'avons nous gagné par rapport à l'étape 5 ? À l'étape 5, nous lisions le fichier de règles entièrement pour construire une liste de toutes les règles avant même d'essayer la première. Maintenant, grâce aux générateurs, nous pouvons faire tout cela de manière paresseuse : nous lisons la première règle et testons si elle s'applique, et si c'est le cas nous ne lisons pas l'ensemble du fichier ni ne créons d'autre fonctions.

Pour en savoir plus

- PEP 255 (http://www.python.org/peps/pep−0255.html) définit les générateurs.
- Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) a de nombreux autres exemples de générateurs (http://www.google.com/search?q=generators+cookbook+site:aspn.activestate.com).

17.8. Résumé

Nous avons vu de nombreuses techniques avancées dans ce chapitre. Ces techniques ne sont pas appropriées dans toutes les situations.

Vous devez maintenant vous sentir à l'aise avec toutes ces techniques :

- Effectuer des remplacement de chaînes avec les expressions régulières.
- Traiter les fonctions comme des objets, les stocker dans des listes, les assigner à des variables et les appeler par l'intermédiaire de ces variables.
- Construire des fonctions dynamiques avec `lambda`.
- Construire des fermetures, des fonctions dynamiques contenant les variables de leur environnement sous forme de constantes.
- Construire des générateurs, des fonctions pouvant reprendre leur exécution au point où elle l'ont laissée et qui retournent des valeurs différentes de manière incrémentale à chaque fois qu'elle sont appelées.

Ajouter des abstractions, construire des fonctions dynamiquement, construire des fermetures et utiliser des générateurs sont autant de techniques qui peuvent rendre votre code plus simple, plus lisible et plus souple. Mais elles peuvent également le rendre plus difficile à déboguer par la suite. À vous de trouver le bon équilibre entre simplicité et puissance.

Chapitre 18. Ajustements des performances

Vous allez découvrir les merveilles de l'ajustement des performances. Ce n'est pas parce que Python est un langage interprété que vous ne devez pas vous soucier de l'optimisation du code. Mais ne vous en souciez pas *trop*.

18.1. Plonger

Il y a tellement de dangers liés à l'optimisation du code qu'il est difficile de savoir par quoi commencer.

Commençons par ceci :*êtes−vous sûr que c'est vraiment nécessaire ?* Votre code est−il si mauvais ? Cela vaut−il la peine de prendre le temps de l'ajuster ? Pendant la durée de vie de votre application, combien de temps sera passé à exécuter ce code, par rapport au temps passé à attendre un serveur de base de données distant ou à attendre une entrée utilisateur ?

Ensuite, *êtes vous sûr que vous avez fini d'écrire le code ?* L'optimisation prématurée, c'est comme mettre une cerise sur un gâteau à moitié cuit. Vous passez des heures ou des jours (ou plus) à optimiser les performances de votre code, simplement pour découvrir qu'il ne remplit pas sa tâche. C'est du temps perdu.

Cela ne veut pas dire que l'optimisation du code est inutile, mais vous devez considérer l'ensemble du système et décider si c'est la meilleure façon d'employer votre temps. Chaque minute que vous passez à optimiser votre code est une minute que vous n'utilisez pas pour ajouter des fonctionnalités, à écrire de la documentation, à jouer avec vos enfants ou à écrire des tests unitaires.

Ah oui, les tests unitaires. Cela va sans dire que vous devez avoir un jeu complet de tests unitaires avant de commencer les ajustements de performances. La dernière chose dont vous avez besoin est d'introduire des nouveaux bogues en modifiant vos algorithmes.

Ces mises en garde étant faites, examinons certaines techniques d'optimisation de code Python. Le code en question est une implémentation de l'algorithme Soundex. Soundex est une méthode utilisée au début du 20e siècle pour classer les patronymes pour le recensement aux Etats−Unis. Il groupe les noms dont la prononciation est semblable, de manière à ce que même si un nom était mal orthographié, il serait possible de le retrouver. Soundex est toujours utilisé pour la même raison, mais bien sûr nous utilisons maintenant des bases de données informatisées. La plupart des logiciels de base de données comprennent une fonction Soundex.

Il y a plusieurs variantes de l'algorithme Soundex. Voici celle que nous utiliserons dans ce chapitre.

1. La première lettre du nom n'est pas modifiée
2. Les lettres suivantes sont converties en chiffres, en fonction de la table suivante :
    - B, F, P et V deviennent 1
    - C, G, J, K, Q, S, X et Z deviennent 2.
    - D et T deviennent 3.
    - L devient 4.
    - M et N deviennent 5.
    - R devient 6.
3. Les doublons consécutifs sont supprimés.
4. Les 9 sont supprimés.
5. Si le résultat fait moins de quatre caractères (la première lettre puis 3 chiffres), le résultat est complété de zéros.
6. Si le résultat fait plus de quatre caractères, tout ce qui suit le quatrième caractère est supprimé.

Par exemple, mon nom, `Pilgrim`, devient P942695. Il n'y a pas de doublons consécutifs, donc rien à faire à cette

étape. Nous enlevons ensuite les 9, ce qui laisse P4265. C'est trop long, nous supprimons les caractères surnuméraires, ce qui laisse P426.

Un autre exemple : `Woo` devient W99, qui devient W9, qui devient W, qui est complété de zéros pour faire W000.

Voici une première tentative de fonction Soundex

Exemple 18.1. `soundex/stage1/soundex1a.py`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−5.4.zip) du livre.

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    "convert string to Soundex equivalent"

    # Soundex requirements:
    # source string must be at least 1 character
    # and must consist entirely of letters
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"

    # Soundex algorithm:
    # 1. make first character uppercase
    source = source[0].upper() + source[1:]

    # 2. translate all other characters to Soundex digits
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
```

```
    # 3. remove consecutive duplicates
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d

    # 4. remove all "9"s
    digits3 = re.sub('9', '', digits2)

    # 5. pad end with "0"s to 4 characters
    while len(digits3) < 4:
        digits3 += "0"

    # 6. return first 4 characters
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Pour en savoir plus sur Soundex

- Soundexing and Genealogy (http://www.avotaynu.com/soundex.html) présente une chronologie de l'évolution de Soundex et de ses variantes régionales.

18.2. Utilisation du module `timeit`

La chose la plus importante que vous devez savoir à propos de l'optimisation de code Python est que vous ne devez pas écrire vos propres fonctions de chronométrage.

Le chronométrage de petits segments de code est incroyablement complexe. Combien de temps processeur est consacré par votre ordinateur à l'exécution de code ? Y-a-t-il des choses tournant en tâche de fond ? En êtes vous sûr ? Tous les ordinateurs récents ont des processus s'exécutant en tâche de fond, certains tout le temps, d'autres de manière intermittente. Des tâches cron s'exécutent à intervalles réguliers, des services en tâche de fond se "réveillent" pour accomplir des tâches comme relever votre courrier électronique, se connecter à des serveurs de messagerie instantanée, vérifier les mises à jour disponibles, rechercher des virus, regarder si un disque a été inséré dans votre lecteur CD dans les 100 dernières nanosecondes etc. Avant de démarrer vos tests de chronométrage, fermez tous ces services et déconnectez-vous du réseau. Ensuite, fermez tout ce que vous avez oublié de fermer la première fois, puis fermez le service qui vérifie de manière incessante si vous êtes connecté au réseau, puis...

Il y a aussi la question des variations introduites par le framework de chronométrage lui-même. L'interpréteur Python a-t-il un cache des tables de méthodes ? Un cache des blocs de code compilés ? Des expressions régulières ? Votre code produit-il des effets de bord si il est exécuté plus d'une fois ? N'oubliez pas qu'il s'agit de fractions de secondes et que des petites erreurs dans votre framework de chronométrage produirons des distorsions irréparables des résultats

La communauté Python a un proverbe : "Python est livré avec les piles." N'écrivez pas votre propre framework de chronométrage. Python 2.3 est livré avec un très bon framework appelé `timeit`.

Exemple 18.2. Présentation de `timeit`

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython-examples-5.4.zip) du livre.

```
>>> import timeit
>>> t = timeit.Timer("soundex.soundex('Pilgrim')",
...     "import soundex")    ❶
>>> t.timeit()               ❷
8.21683733547
>>> t.repeat(3, 2000000)     ❸
[16.48319309109, 16.46128984923, 16.44203948912]
```

❶ Le module `timeit` définit une classe, `Timer`, qui prend deux arguments. Les deux arguments sont des chaînes. Le premier argument est l'instruction que nous voulons chronométrer, dans le cas présent nous chronométrons un appel à la fonction Soundex du module `soundex` avec pour argument `'Pilgrim'`. Le deuxième argument de la classe `Timer` est l'instruction d'importation qui met en place l'environnement de l'instruction. En interne, `timeit` met en place un environnement virtuel isolé, exécute l'instruction de mise en place (importation du module `soundex`), puis compile et exécute l'instruction à chronométrer (appel de la fonction Soundex).

❷ Une fois que nous avons l'objet `Timer`, la chose la plus simple à faire est d'appeler `timeit()`, qui appelle notre fonction 1 million de fois et retourne le nombre de secondes que cela a pris.

❸ L'autre méthode importante de l'objet `Timer` est `repeat()`, qui prend deux arguments optionnels. Le premier argument est le nombre de répétition du test et le second est le nombre de fois que l'instruction sera exécutée pour chaque test. Les deux arguments sont optionnels, leurs valeurs par défaut sont respectivement `3` et `1000000`. La méthode `repeat()` retourne une liste du temps que chaque cycle de test a pris, en secondes.

Vous pouvez utiliser le module `timeit` en ligne de commande pour tester un programme Python existant sans en modifier le code. Voir http://docs.python.org/lib/node396.html pour la documentation des paramètres de ligne de commande.

Notez que `repeat()` retourne une liste de temps. Les temps ne seront pratiquement jamais identiques, à cause des variations du temps processeur alloué à l'interpréteur Python (et de toutes les tâches de fond dont on ne peut pas se débarrasser). Il est tentant de se dire "Prenons la moyenne et considérons que c'est le nombre correct."

En fait, c'est presque certainement faux. Les tests qui ont pris plus de temps ne l'ont pas fait à cause de variations dans notre code ou dans l'interpréteur Python, ils ont pris plus de temps à cause des tâches de fond, ou d'autres facteurs externes à l'interpréteur Python que nous ne pouvons pas entièrement éliminer. Si les différents résultats varient en pourcentage de plus de quelques points, il y a trop de variation pour que nous puissions nous y fier. Sinon, c'est le temps minimum qu'il faut prendre en compte et ne pas tenir compte du reste..

Python a une fonction `min` très utile qui prend une liste et retourne la valeur la plus petite de la liste :

```
>>> min(t.repeat(3, 1000000))
8.22203948912
```

Le module `timeit` ne fonctionne que si vous savez déjà quelle partie de votre code optimiser. Si vous avez un programme Python plus grand et que vous ne savez pas où se trouve le problème de performances, allez voir le module `hotshot`. (http://docs.python.org/lib/module-hotshot.html)

18.3. Optimisation d'expressions régulières

La première chose que la fonction Soundex vérifie est que l'entrée est une chaîne non-vide composée de lettres. Quelle est la meilleure manière de faire cela ?

Si vous pensez que c'est avec une expression régulière, c'est que vous vous êtes laissé entrainer par vos bas instincts. Les expressions régulières ne sont presque jamais la bonne réponse, elles doivent être évitées à chaque fois que c'est possible. Pas seulement pour des raisons de performances, mais parce qu'elles sont difficiles à déboguer et à maintenir. Et aussi pour des raisons de performances.

Ce fragment de code tiré de `soundex/stage1/soundex1a.py` vérifie que l'argument de la fonction `source` est un mot constitué uniquement de lettres, long d'au moins une lettre (pas une chaîne vide) :

```
    allChars = string.uppercase + string.lowercase
    if not re.search('^[%s]+$' % allChars, source):
        return "0000"
```

Quelles sont les performances de `soundex1a.py` ? Pour rendre les choses plus simples, la section `__main__` du scrupt contient le code suivant, qui appelle le module `timeit`, met en place un chronométrage avec trois différents noms et affiche le temps minimum de chaque test :

```
if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

Alors, quelles sont les performances de `soundex1a.py` avec cette expression régulière ?

```
C:\samples\soundex\stage1>python soundex1a.py
Woo             W000 19.3356647283
Pilgrim         P426 24.0772053431
Flingjingwaller F452 35.0463220884
```

Comme nous pouvions nous y attendre, l'algorithme prend plus de temps lorsqu'on l'appelle avec un nom plus long. Il y a un certain nombre de choses que nous pouvons faire pour réduire cet écart, mais la nature de cet algorithme fait qu'il ne s'exécutera jamais en temps constant.

Une autre chose qu'il faut garder à l'esprit est que nous testons un échantillon représentatif de noms. `Woo` est un cas trivial puisqu'il est réduit à une seule lettre puis complété de zéros. `Pilgrim` est un cas normal, de longueur moyenne et composé d'un mélange de lettres significatives et ignorées. `Flingjingwaller` est extraordinairement long et contient des doublons consécutifs. D'autres tests pourraient être utiles, mais ceux-ci nous donnent déjà un bon échantillon de cas.

Et cette expression régulière ? Et bien, elle n'est pas efficiente. Puisque l'expression recherche un ensemble de caractères (`A-Z` en majuscules et `a-z` en minuscules), nous pouvons utiliser une syntaxe d'expression régulière abrégée. Voici `soundex/stage1/soundex1b.py` :

```
    if not re.search('^[A-Za-z]+$', source):
        return "0000"
```

`timeit` nous dit que `soundex1b.py` est un peu plus rapide que `soundex1a.py`, mais rien de transcendant :

```
C:\samples\soundex\stage1>python soundex1b.py
Woo             W000 17.1361133887
Pilgrim         P426 21.8201693232
Flingjingwaller F452 32.7262294509
```

Nous avons vu à la Section 15.3, «Refactorisation» que les expressions régulières peuvent être compilées et réutilisées pour avoir des résultats plus rapides. Puisque cette expression régulière ne change jamais d'un appel de la fonction à un autre, nous pouvons la compiler une fois pour toutes. Voici `soundex/stage1/soundex1c.py` :

```
isOnlyChars = re.compile('^[A-Za-z]+$').search
def soundex(source):
    if not isOnlyChars(source):
        return "0000"
```

L'utilisation d'une expression régulière compilée dans `soundex1c.py` est nettement plus rapide :

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5348347346
Pilgrim         P426 19.2784703084
Flingjingwaller F452 30.0893873383
```

Mais est-ce la bonne manière ? La logique ici est simple : l'entrée `source` doit être non-vide et ne contenir que des lettres. Ne serait-il pas plus rapide d'écrire une boucle pour tester chaque caractère et de supprimer l'expression régulière ?

Voici `soundex/stage1/soundex1d.py`:

```
    if not source:
        return "0000"
    for c in source:
        if not ('A' <= c <= 'Z') and not ('a' <= c <= 'z'):
            return "0000"
```

En fait, cette technique dans `soundex1d.py` n'est *pas* plus rapide qu'avec une expression régulière compilée (bien qu'elle soit plus rapide qu'avec une expression régulière non-compilée) :

```
C:\samples\soundex\stage1>python soundex1d.py
Woo             W000 15.4065058548
Pilgrim         P426 22.2753567842
Flingjingwaller F452 37.5845122774
```

Pourquoi `soundex1d.py` n'est-il pas plus rapide ? La réponse est à trouver dans la nature interprétée de Python. Le moteur d'expressions régulières est écrit en C et compilé pour s'exécuter nativement sur votre ordinateur. Cette boucle, par contre, est écrite en Python et est exécutée par l'interpréteur Python. Même si cette boucle est relativement simple, elle n'est pas assez simple pour compenser la pénalité due à l'interprétation. Les expressions régulières ne sont jamais la bonne solution... sauf quand elles le sont.

Il se trouve que Python propose une méthode de chaîne peu connue. Vous pouvez être pardonné de ne pas la connaître car elle n'a jamais été mentionnée dans ce livre. Cette méthode est `isalpha()` et elle vérifie qu'une chaîne ne contient que des lettres.

Voici `soundex/stage1/soundex1e.py`:

```
    if (not source) and (not source.isalpha()):
        return "0000"
```

Quels gain de performances nous a apporté l'utilisation de cette méthode dans `soundex1e.py` ? Un gain assez important.

```
C:\samples\soundex\stage1>python soundex1e.py
Woo             W000 13.5069504644
```

```
Pilgrim         P426 18.2199394057
Flingjingwaller F452 28.9975225902
```

Exemple 18.3. Le meilleur résultat jusqu'à maintenant : `soundex/stage1/soundex1e.py`

```
import string, re

charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    if (not source) and (not source.isalpha()):
        return "0000"
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.4. Optimisation de la lecture d'un dictionnaire

La deuxième étape de l'algorithme Soundex est de convertir les caractères en chiffres suivant des règles précises. Quelle est la meilleure manière de procéder ?

La solution la plus évidente est de définir un dictionnaire ayant les caractères comme clés et le chiffre leur correspondant comme valeurs et de faire une lecture du dictionnaire pour chaque caractère. C'est cette méthode qui est employée dans `soundex/stage1/soundex1c.py` (la version la plus performante jusqu'à maintenant) :

```
charToSoundex = {"A": "9",
                 "B": "1",
                 "C": "2",
                 "D": "3",
                 "E": "9",
                 "F": "1",
                 "G": "2",
                 "H": "9",
                 "I": "9",
                 "J": "2",
                 "K": "2",
                 "L": "4",
                 "M": "5",
                 "N": "5",
                 "O": "9",
                 "P": "1",
                 "Q": "2",
                 "R": "6",
                 "S": "2",
                 "T": "3",
                 "U": "9",
                 "V": "1",
                 "W": "9",
                 "X": "2",
                 "Y": "9",
                 "Z": "2"}

def soundex(source):
    # ... input check omitted for brevity ...
    source = source[0].upper() + source[1:]
    digits = source[0]
    for s in source[1:]:
        s = s.upper()
        digits += charToSoundex[s]
```

Nous avons déjà chronométré `soundex1c.py`, voici le résultat que nous avions obtenu :

```
C:\samples\soundex\stage1>python soundex1c.py
Woo             W000 14.5341678901
Pilgrim         P426 19.2650071448
Flingjingwaller F452 30.1003563302
```

Ce code évident, mais est−ce la meilleure solution ? L'appel de `upper()` pour chaque caractère semble peu efficient, il vaudrait sans doute mieux appeler `upper()` une seule fois pour la chaîne entière.

Il y a aussi le fait de construire la chaîne `digits` de manière incrémentale. Construire des chaînes de cette manière est peu efficient car Python crée une nouvelle chaîne à chaque itération et efface l'ancienne.

Python est meilleur pour manipuler des liste et il peut traiter une chaîne comme une liste de caractères

automatiquement. Les listes sont facilement transformables en chaînes grâce à la méthode de chaîne `join()`.

Voici `soundex/stage2/soundex2a.py`, qui convertit les lettres en chiffres à l'aide de ¦ et `lambda`:

```
def soundex(source):
    # ...
    source = source.upper()
    digits = source[0] + "".join(map(lambda c: charToSoundex[c], source[1:]))
```

Ce qui est surprenant, c'est que `soundex2a.py` n'est pas plus rapide :

```
C:\samples\soundex\stage2>python soundex2a.py
Woo             W000 15.0097526362
Pilgrim         P426 19.254806407
Flingjingwaller F452 29.3790847719
```

Le coût de l'appel de la fonction `lambda` efface tous les gains de performances obtenus en considérant la chaîne comme une liste de caractères.

`soundex/stage2/soundex2b.py` utilise une *list comprehension* au lieu de ¦ et `lambda`:

```
    source = source.upper()
    digits = source[0] + "".join([charToSoundex[c] for c in source[1:]])
```

L'emploi d'une *list comprehension* dans `soundex2b.py` est plus rapide qu'utiliser ¦ et `lambda` dans `soundex2a.py`, mais toujours pas plus rapide que le code originel (la construction incrémentale d'une chaîne dans `soundex1c.py`) :

```
C:\samples\soundex\stage2>python soundex2b.py
Woo             W000 13.4221324219
Pilgrim         P426 16.4901234654
Flingjingwaller F452 25.8186157738
```

Il est temps d'essayer une approche radicalement différente. La lecture de dictionnaire est un outil générique. Les clés de dictionnaire peuvent être des chaînes de taille variable (ou beaucoup d'autres types de données), mais dans le cas présent, nous n'utilisons que des clés d'un seul caractère *et* des valeurs d'un seul caractère. Il se trouve que Python a une fonction spécialisée pour ce genre de situations : la fonction `string.maketrans`.

Voici `soundex/stage2/soundex2c.py` :

```
allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "9123912992245591262391929291239129922455912623919292" * 2)
def soundex(source):
    # ...
    digits = source[0].upper() + source[1:].translate(charToSoundex)
```

Que se passe−t−il exactement ici ? `string.maketrans` crée une matrice de traduction entre deux chaînes : le premier argument et le second. Dans le cas présent, le premier argument est la chaîne `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz` et le second la chaîne `9123912992245591262391929291239129922455912623919292`. C'est exactement le motif de conversion que nous avions mis en place avec le dictionnaire. A correspond à 9, B à 1, C à 2 et ainsi de suite. Mais ce n'est pas un dictionnaire, c'est une structure de données spécialisée à laquelle on accède à l'aide de la méthode de chaîne `translate`, qui traduit chaque caractère vers le chiffre correspondant, selon la matrice définie par `string.maketrans`.

`timeit` montre que `soundex2c.py` est plus rapide que de définir un dictionnaire et construire la chaîne de manière incrémentale dans une boucle :

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 11.437645008
Pilgrim         P426 13.2825062962
Flingjingwaller F452 18.5570110168
```

Nous n'obtiendrons pas grand chose de mieux que ça. Python a une fonction spécialisée qui fait exactement ce que nous souhaitons, utilisons−là et passons à la suite.

Exemple 18.4. Meilleur résultat jusqu'à maintenant : `soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A-Za-z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.5. Optimisation des opérations sur les listes

La troisième étape de l'algorithme Soundex est l'élimination des doublons successifs. Quelle est la meilleure manière de faire cela ?

Voici le code que nous avons pour l'instant dans `soundex/stage2/soundex2c.py` :

```
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[-1] != d:
            digits2 += d
```

Voici les résultats du chronométrage de `soundex2c.py` :

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
```

```
Flingjingwaller F452 19.7774882003
```

La première chose à considérer est l'efficience de vérifier `digits[-1]` à chaque itération de la boucle. Est-ce que les index de listes sont coûteux ? Vaudrait-il mieux garder le dernier chiffre dans une variable et vérifier cette variable ?

Pour répondre à cette question, voici `soundex/stage3/soundex3a.py` :

```
    digits2 = ''
    last_digit = ''
    for d in digits:
        if d != last_digit:
            digits2 += d
            last_digit = d
```

`soundex3a.py` n'est pas plus rapide que `soundex2c.py` et peut-être même un peu plus lent (bien que la différence ne soit pas assez importante pour être sûr) :

```
C:\samples\soundex\stage3>python soundex3a.py
Woo             W000 11.5346048171
Pilgrim         P426 13.3950636184
Flingjingwaller F452 18.6108927252
```

Pourquoi `soundex3a.py` n'est-il pas plus rapide ? Il se trouve que les index de listes en Python sont extrêmement efficient. Accéder à `digits2[-1]` de manière répétée n'est pas du tout un problème. Par contre, maintenir manuellement une variable pour le dernier chiffre fait que nous avons *deux* affectations de variables pour chaque chiffre que nous stockons, ce qui supprime les petits bénéfices que nous pourrions avoir reçu de l'élimination de l'accès à la liste.

Essayons quelque chose de radicalement différent. S'il est possible de traiter une chaîne comme une liste de caractères, il devrait être possible d'utiliser une *list comprehension* pour parcourir la liste. Le problème est que notre code a besoin d'accéder au caractère précédent dans la liste et ce n'est pas facile à faire avec une simple *list comprehension*.

Cependant, il est possible de créer une liste d'index à l'aide de la fonction prédéfinie `range()` et d'utiliser cette liste pour chercher dans la liste chaque caractère qui diffère de celui qui le précède. Cela nous donnera une liste de caractères que nous pourrons convertir en chaîne avec la méthode de chaîne `join()`.

Voici `soundex/stage3/soundex3b.py` :

```
    digits2 = "".join([digits[i] for i in range(len(digits))
                       if i == 0 or digits[i-1] != digits[i]])
```

Est-ce plus rapide ? En un mot, non.

```
C:\samples\soundex\stage3>python soundex3b.py
Woo             W000 14.2245271396
Pilgrim         P426 17.8337165757
Flingjingwaller F452 25.9954005327
```

Peut-être que les techniques employées jusqu'à maintenant sont un peu trop centrées sur les chaînes. Python peut convertir une chaîne en liste de caractères en une seule commande : `list('abc')` retourne `['a', 'b', 'c']`. De plus, les listes peuvent être *modifiée sur place* très rapidement. Au lieu de construire une nouvelle liste (ou une nouvelle chaîne) de manière incrémentale, nous pourrions déplacer les éléments à l'intérieur de la liste.

Voici `soundex/stage3/soundex3c.py`, qui modifie une liste sur place et en supprime les doublons successifs :

```
    digits = list(source[0].upper() + source[1:].translate(charToSoundex))
    i=0
    for item in digits:
        if item==digits[i]: continue
        i+=1
        digits[i]=item
    del digits[i+1:]
    digits2 = "".join(digits)
```

Est−ce plus rapide que `soundex3a.py` ou `soundex3b.py` ? Non, en fait c'est la méthode la plus lente jusqu'à présent :

```
C:\samples\soundex\stage3>python soundex3c.py
Woo             W000 14.1662554878
Pilgrim         P426 16.0397885765
Flingjingwaller F452 22.1789341942
```

Nous n'avons fais aucun progrès du tout, à part essayer et rejeter plusieurs techniques "astucieuses". Le code le plus rapide que nous avons vu jusque là est celui de la méthode originelle, la plus évidente (`soundex2c.py`). Parfois, l'astuce ne paie pas.

Exemple 18.5. Meilleur résultat jusqu'à maintenant : `soundex/stage2/soundex2c.py`

```
import string, re

allChar = string.uppercase + string.lowercase
charToSoundex = string.maketrans(allChar, "91239129922455912623919292" * 2)
isOnlyChars = re.compile('^[A−Za−z]+$').search

def soundex(source):
    if not isOnlyChars(source):
        return "0000"
    digits = source[0].upper() + source[1:].translate(charToSoundex)
    digits2 = digits[0]
    for d in digits[1:]:
        if digits2[−1] != d:
            digits2 += d
    digits3 = re.sub('9', '', digits2)
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]

if __name__ == '__main__':
    from timeit import Timer
    names = ('Woo', 'Pilgrim', 'Flingjingwaller')
    for name in names:
        statement = "soundex('%s')" % name
        t = Timer(statement, "from __main__ import soundex")
        print name.ljust(15), soundex(name), min(t.repeat())
```

18.6. Optimisation des manipulations de chaînes

L'étape finale de l'algorithme Soundex est de compléter les résultats courts par des zéros et de tronquer les résultats long. Quelle est la meilleure manière de faire cela ?

Voici ce que nous avons pour l'instant, tiré de `soundex/stage2/soundex2c.py` :

```
    digits3 = re.sub('9', '', digits2)
```

```
    while len(digits3) < 4:
        digits3 += "0"
    return digits3[:4]
```

Voici les résultats pour `soundex2c.py` :

```
C:\samples\soundex\stage2>python soundex2c.py
Woo             W000 12.6070768771
Pilgrim         P426 14.4033353401
Flingjingwaller F452 19.7774882003
```

La première chose à essayer est de remplacer l'expression régulière par une boucle. Voici le code de `soundex/stage4/soundex4a.py` :

```
    digits3 = ''
    for d in digits2:
        if d != '9':
            digits3 += d
```

Est−ce que `soundex4a.py` est plus rapide ? Oui, il l'est :

```
C:\samples\soundex\stage4>python soundex4a.py
Woo             W000 6.62865531792
Pilgrim         P426 9.02247576158
Flingjingwaller F452 13.6328416042
```

Mais il y a quelque chose qui cloche, une boucle pour enlever des caractères d'une chaîne ? Nous pouvons utiliser une simple méthode de chaîne pour ça. Voici `soundex/stage4/soundex4b.py` :

```
    digits3 = digits2.replace('9', '')
```

Est−ce que `soundex4b.py` est plus rapide ? C'est une bonne question, ça dépend de l'entrée :

```
C:\samples\soundex\stage4>python soundex4b.py
Woo             W000 6.75477414029
Pilgrim         P426 7.56652144337
Flingjingwaller F452 10.8727729362
```

La méthode de chaîne dans `soundex4b.py` est plus rapide que la boucle pour la plupart des noms, mais elle est en fait un peu plus lente que `soundex4a.py` dans les cas triviaux (pour des noms très courts). Les optimisations de performances ne sont pas toujours uniformes, des réglages qui rendent un cas plus rapide peuvent rendre d'autres cas plus lent. Dans le cas présent, la majorité des cas bénéficiera de la modification, donc nous allons la conserver, mais rappelons nous de ce principe.

Pour finir, examinons les deux dernières étapes de l'algorithme : compléter les résultats courts avec des zéros et tronquer les résultats long à quatre caractères. Le code dans `soundex4b.py` fait cela, mais il est horriblement inefficace. Regardons `soundex/stage4/soundex4c.py` pour voir pourquoi :

```
    digits3 += '000'
    return digits3[:4]
```

Pourquoi utiliser une boucle `while` pour compléter le résultat ? Nous savons à l'avance que nous allons tronquer le résultat à quatre caractères et que nous avons au moins un caractère (la première lettre, qui est passée sans modification de la variable `source` d'origine). Cela signifie que nous pouvons simplement ajouter trois zéros à la sortie, puis la tronquer. Il ne faut pas se laisser enfermer par la formulation précise du problème, envisager le problème sous un angle un peu différent peut amener à une solution plus simple.

Quels gains de vitesse avons nous obtenus dans `soundex4c.py` en supprimant la boucle `while` ? Ils sont assez significatifs :

```
C:\samples\soundex\stage4>python soundex4c.py
Woo             W000 4.89129791636
Pilgrim         P426 7.30642134685
Flingjingwaller F452 10.689832367
```

Pour finir, il y a encore quelque chose que nous pouvons faire pour rendre ces trois lignes de code plus rapide : nous pouvons les combiner en une seule ligne. Regardons `soundex/stage4/soundex4d.py` :

```
    return (digits2.replace('9', '') + '000')[:4]
```

Mettre tout ce code sur une seule ligne dans `soundex4d.py` est à peine plus rapide que dans `soundex4c.py` :

```
C:\samples\soundex\stage4>python soundex4d.py
Woo             W000 4.93624105857
Pilgrim         P426 7.19747593619
Flingjingwaller F452 10.5490700634
```

C'est aussi bien moins lisible, pour des gains minimes. Est−ce que ça en vaut la peine ? Il vaudrait mieux avoir de bons commentaires, les performances ne sont pas tout. Il faut toujours équilibrer l'optimisation des performances et la lisibilité (et donc la maintenabilité) d'un programme.

18.7. Résumé

Ce chapitre a illustré plusieurs aspects importants des réglages de performances en Python et en général.

- Si vous avez le choix entre une expression régulière et une boucle, choisissez l'expression régulière. Le moteur d'expressions régulières est écrit en C et est exécuté en mode natif par votre ordinateur, les boucles sont écrites en Python et sont exécutées par l'interpréteur Python.
- Si vous devez choisir entre une expression régulière et une méthode de chaîne, choisissez la méthode de chaîne. Les deux sont écrites en C, il faut donc choisir le plus simple.
- Les recherches dans les dictionnaires sont rapides, mais les fonctions spécialisées comme `string.maketrans` et les méthodes de chaînes comme `isalpha()` sont plus rapide. Si Python a une méthode spécialisée pour remplir une tâche, utilisez−la.
- Ne soyez pas trop astucieux. Parfois l'algorithme le plus évident est aussi le plus rapide.
- N'en faites pas trop. Les performances ne sont pas tout.

J'insiste sur ce dernier point. Au cours de ce chapitre, nous avons rendu la fonction trois fois plus rapide et gagné 20 secondes sur 1 million d'appel de fonction. C'est bien. Mais réfléchissez, pendant l'exécution de ce million d'appels de fonction, combien de secondes sont passées par notre application à attendre une connexion de base de données, la fin d'une écriture disque ou une entrée de l'utilisateur ? Ne passez pas trop de temps à optimiser une algorithme, ou vous raterez des améliorations dans d'autres parties de votre programme. Il faut développer un instinct permettant de savoir le genre de code que Python exécute rapidement, corriger les erreurs grossières si vous les trouvez et laisser le reste tranquille.

Annexe A. Pour en savoir plus

Chapitre 1. Installation de Python

Chapitre 2. Votre premier programme Python

- 2.3. Documentation des fonctions
    - La PEP 257 (http://www.python.org/peps/pep−0257.html) définit les conventions pour les `doc string`.
    - Le *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) explique la manière d'écrire de bonnes `doc string`.
    - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite des conventions d'espacement dans les `doc string` (http://www.python.org/doc/current/tut/node6.html#SECTION00675000000000000000).
- 2.4.2. Qu'est−ce qu'un objet ?
    - La *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique précisémment ce qu'implique de dire que tout est objet en Python (http://www.python.org/doc/current/ref/objects.html), puisque certains pédants aiment discuter longuement de ce genre de choses.
    - eff−bot (http://www.effbot.org/guides/) propose un résumé des objets Python (http://www.effbot.org/guides/python−objects.htm).
- 2.5. Indentation du code
    - La *Python Reference Manual* (http://www.python.org/doc/current/ref/) discute des aspects multiplate−formes de l'indentation et présente diverses erreurs d'indentation (http://www.python.org/doc/current/ref/indentation.html).
    - Le *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discute du bon usage de l'indentation.
- 2.6. Test des modules
    - La *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique les détails techniques de l'importation de modules (http://www.python.org/doc/current/ref/import.html).

Chapitre 3. Types prédéfinis

- 3.1.3. Enlever des éléments d'un dictionnaire
    - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) explique comment utiliser les dictionnaires pour modéliser les matrices creuses (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
    - La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) a de nombreux exemples de code ayant recours aux dictionnaires (http://www.faqts.com/knowledge−base/index.phtml/fid/541).
    - Le Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) explique comment trier les valeurs d'un dictionnaire par leurs clés (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52306).
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes des dictionnaires (http://www.python.org/doc/current/lib/typesmapping.html).
- 3.2.5. Utilisation des opérateurs de listes

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) explique les listes et expose le sujet important du passage de listes comme arguments de fonction (http://www.ibiblio.org/obp/thinkCSpy/chap08.htm).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) montre comment utiliser des listes comme des piles ou des files (http://www.python.org/doc/current/tut/node7.html#SECTION007110000000000000000).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions courantes à propos des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/534) et fourni de nombreux exemples de code utilisant des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/540).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes des listes (http://www.python.org/doc/current/lib/typesseq−mutable.html).

• 3.3. Présentation des tuples

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) explique les tuples et montre comment concaténer des tuples (http://www.ibiblio.org/obp/thinkCSpy/chap10.htm).
- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) vous apprendra à trier un tuple (http://www.faqts.com/knowledge−base/view.phtml/aid/4553/fid/587).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) explique comment définir un tuple avec un seul élément (http://www.python.org/doc/current/tut/node7.html#SECTION007300000000000000000).

• 3.4.2. Assignation simultanée de plusieurs valeurs

- Le *Python Reference Manual* (http://www.python.org/doc/current/ref/) présente des exemples des cas où vous pouvez omettre le marqueur de continuation (http://www.python.org/doc/current/ref/implicit−joining.html) et où vous devez l'utiliser (http://www.python.org/doc/current/ref/explicit−joining.html).
- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) montre comment utiliser l'assignation multiple pour échanger les valeurs de deux variables (http://www.ibiblio.org/obp/thinkCSpy/chap09.htm).

• 3.5. Formatage de chaînes

- La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume tous les caractères spéciaux utilisés pour le formatage de chaînes (http://www.python.org/doc/current/lib/typesseq−strings.html).
- *Effective AWK Programming* (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Top) explique tous les caractères de formatage (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Control+Letters) et des technique de formatage avancées comme le règlage de la largeur ou de la précision et le remplissage avec des zéros (http://www−gnats.gnu.org:8080/cgi−bin/info2www?(gawk)Format+Modifiers).

• 3.6. Mutation de listes

- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d'une autre manière de transformer des listes avec la fonction prédéfinie `map` (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
- Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) montre comment emboîter des mutations de listes (http://www.python.org/doc/current/tut/node7.html#SECTION007140000000000000000).

• 3.7. Jointure de listes et découpage de chaînes

- La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions courantes à propose des chaînes

(http://www.faqts.com/knowledge−base/index.phtml/fid/480) et dispose de nombreux exemples de code utilisant des chaînes (http://www.faqts.com/knowledge−base/index.phtml/fid/539).

- La *Python Library Reference* (http://www.python.org/doc/current/lib/) récapitule toutes les méthodes de chaînes (http://www.python.org/doc/current/lib/string−methods.html).
- La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `string` (http://www.python.org/doc/current/lib/module−string.html).
- *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) explique pourquoi `join` est une méthode de chaînes (http://www.python.org/cgi−bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) et non une méthode de liste.

Chapitre 4. Le pouvoir de l introspection

- 4.2. Arguments optionnels et nommés
  - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de manière précise de quand et comment les arguments par défaut sont évalués (http://www.python.org/doc/current/tut/node6.html#SECTION00671000000000000000), ce qui est important lorsque la valeur par défaut est une liste ou une expression ayant un effet de bord.
- 4.3.3. Fonctions prédéfinies
  - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente toutes les fonctions prédéfinies (http://www.python.org/doc/current/lib/built−in−funcs.html) et toutes les exceptions prédéfinies (http://www.python.org/doc/current/lib/module−exceptions.html).
- 4.5. Filtrage de listes
  - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d une autre manière de filtrer les listes en utilisant la fonction prédéfinie `filter` (http://www.python.org/doc/current/tut/node7.html#SECTION00713000000000000000).
- 4.6.1. Utilisation de l'astuce and−or
  - Le Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) traite des alternatives à l astuce `and−or` (http://www.activestate.com/ASPN/Python/Cookbook/Recipe/52310).
- 4.7.1. Les fonctions lambda dans le monde réel
  - La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) traite de l utilisation de `lambda` pour faire des appels de fonction indirects (http://www.faqts.com/knowledge−base/view.phtml/aid/6081/fid/241).
  - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) montre comment accéder à des variables extérieures de l intérieur d une fonction `lambda` (http://www.python.org/doc/current/tut/node6.html#SECTION00674000000000000000). (La PEP 227 (http://python.sourceforge.net/peps/pep−0227.html) explique comment cela va changer dans les futures versions de Python.)
  - La *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) a des exemples de code obscur monoligne utilisant `lambda` (http://www.python.org/cgi−bin/faqw.py?query=4.15&querytype=simple&casefold=yes&req=search).

Chapitre 5. Les objets et l'orienté objet

- 5.2. Importation de modules avec from module import
  - eff−bot (http://www.effbot.org/guides/) a d'autres choses à ajouter à propos de `import` *`module`* et

`from module import` (http://www.effbot.org/guides/import-confusion.htm).
    - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de techniques d'import avancées, y compris `from module import *` (http://www.python.org/doc/current/tut/node8.html#SECTION008410000000000000000).
- 5.3.2. Quand utiliser self et __init__
    - *Learning to Program* (http://www.freenetpages.co.uk/hp/alan.gauld/) a une introduction en douceur aux classes (http://www.freenetpages.co.uk/hp/alan.gauld/tutclass.htm).
    - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) montre comment utiliser des classes pour modéliser des types composés (http://www.ibiblio.org/obp/thinkCSpy/chap12.htm).
    - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) a un aperçu en profondeur des classes, des espaces de noms et de l'héritage (http://www.python.org/doc/current/tut/node11.html).
    - La Python Knowledge Base (http://www.faqts.com/knowledge-base/index.phtml/fid/199/) répond à des questions courantes à propos des classes (http://www.faqts.com/knowledge-base/index.phtml/fid/242).
- 5.4.1. Ramasse-miettes
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume les attributs prédéfinis comme `__class__` (http://www.python.org/doc/current/lib/specialattrs.html).
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `gc` module (http://www.python.org/doc/current/lib/module-gc.html), qui vous donne un contrôle de bas niveau sur le ramasse-miettes de Python.
- 5.5. UserDict : une classe enveloppe
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `UserDict` (http://www.python.org/doc/current/lib/module-UserDict.html) et le module `copy` (http://www.python.org/doc/current/lib/module-copy.html).
- 5.7. Méthodes spéciales avancées
    - La *Python Reference Manual* (http://www.python.org/doc/current/ref/) documente toutes les méthodes spéciales de classe (http://www.python.org/doc/current/ref/specialnames.html).
- 5.9. Fonctions privées
    - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite du fonctionnement des variables privées (http://www.python.org/doc/current/tut/node11.html#SECTION001160000000000000000).

Chapitre 6. Traitement des exceptions et utilisation de fichiers

- 6.1.1. Utilisation d'exceptions pour d'autres cas que la gestion d'erreur
    - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de la définition et du déclenchement de vos propres exceptions et de la gestion de plusieurs exceptions à la fois (http://www.python.org/doc/current/tut/node10.html#SECTION001040000000000000000).
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les exceptions prédéfinies (http://www.python.org/doc/current/lib/module-exceptions.html).
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module getpass (http://www.python.org/doc/current/lib/module-getpass.html).
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `traceback` (http://www.python.org/doc/current/lib/module-traceback.html), qui fournit un accès de bas niveau aux attributs d'une exception après qu'elle ait été déclenchée.

- La *Python Reference Manual* (http://www.python.org/doc/current/ref/) traite du fonctionnement interne du bloc `try...except` (http://www.python.org/doc/current/ref/try.html).

- 6.2.4. Ecriture dans un fichier

  - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de la lecture et de l'écriture de fichiers, y compris comment lire un fichier ligne par ligne dans une liste (http://www.python.org/doc/current/tut/node9.html#SECTION009210000000000000000).
  - eff−bot (http://www.effbot.org/guides/) traite de l'efficience et de la performance de différentes manières de lire un fichier (http://www.effbot.org/guides/readline−performance.htm).
  - La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions courantes à propose des fichiers (http://www.faqts.com/knowledge−base/index.phtml/fid/552).
  - La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes de l'objet−fichier (http://www.python.org/doc/current/lib/bltin−file−objects.html).

- 6.4. Utilisation de sys.modules

  - Le *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de quand et comment les arguments par défaut sont évalués (http://www.python.org/doc/current/tut/node6.html#SECTION006710000000000000000).
  - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `sys` (http://www.python.org/doc/current/lib/module−sys.html).

- 6.5. Travailler avec des répertoires

  - La Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions sur le module `os` (http://www.faqts.com/knowledge−base/index.phtml/fid/240).
  - La *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `os` (http://www.python.org/doc/current/lib/module−os.html) et le module `os.path` (http://www.python.org/doc/current/lib/module−os.path.html).

Chapitre 7. Expressions régulières

- 7.6. Etude de cas : reconnaissance de numéros de téléphone

  - La Regular Expression HOWTO (http://py−howto.sourceforge.net/regex/regex.html) explique les expressions régulières et leur usage en Python.
  - La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume le module `re` (http://www.python.org/doc/current/lib/module−re.html).

Chapitre 8. Traitement du HTML

- 8.4. Présentation de BaseHTMLProcessor.py

  - Le W3C (http://www.w3.org/) traite des références de caractères et d entités (http://www.w3.org/TR/REC−html40/charset.html#entities).
  - La *Python Library Reference* (http://www.python.org/doc/current/lib/) confirme vos soupçons selon lesquels le module (http://www.python.org/doc/current/lib/module−htmlentitydefs.html) est exactement ce que son nom laisse deviner.

- 8.9. Assembler les pièces

  - Vous croyiez que je plaisantais quand je parlais de traitement côté serveur. C est ce que je pensais aussi, jusqu à ce que je trouve ce "traducteur" en ligne (http://rinkworks.com/dialect/). Malheureusement, le code source n a pas l air d être disponible.

Chapitre 9. Traitement de données XML

- 9.4. Le standard Unicode

    - Unicode.org (http://www.unicode.org/) est le site officiel du standard Unicode et propose une brève introduction technique (http://www.unicode.org/standard/principles.html).
    - Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) contient beaucoup plus d'exemples sur la façon d'utiliser les fonctions Unicode de Python, y compris la manière de forcer Python à contraindre l'Unicode en ASCII, même s'il regimbe.
    - PEP 263 (http://www.python.org/peps/pep−0263.html) approfondit la manière de définir un jeu d'encodage dans vos fichiers `.py`.

Chapitre 10. Des scripts et des flots de données (streams)

Chapitre 11. Services Web HTTP

- 11.1. Plonger

    - Paul Prescod pense que les services Web HTTP sont le futur de l'Internet (http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html).

Chapitre 12. Services Web SOAP

- 12.1. Plonger

    - http://www.xmethods.net/ est un répertoire de services Web SOAP en accès public.
    - La spécification de SOAP (http://www.w3.org/TR/soap/) est étonnamment lisible, si vous aimez ce genre de choses.
- 12.8. Recherche d'erreurs dans les services Web SOAP

    - New developments for SOAPpy (http://www−106.ibm.com/developerworks/webservices/library/ws−pyth17.html) explique pas à pas une tentative de connexion à un service SOAP qui ne fonctionne pas comme il est dit.

Chapitre 13. Tests unitaires

- 13.1. Introduction au chiffres romains

    - Ce site (http://www.wilkiecollins.demon.co.uk/roman/front.htm) a plus d information sur les nombres romains, y compris une histoire (http://www.wilkiecollins.demon.co.uk/roman/intro.htm) fascinante de la manière dont les Romains et d autres civilisations les utilisaient vraiment (pour faire court, à l aveuglette et sans cohérence).
- 13.3. Présentation de romantest.py

    - Le site Web de PyUnit (http://pyunit.sourceforge.net/) présente un traitement en profondeur de l usage du module `unittest` (http://pyunit.sourceforge.net/pyunit.html), y compris des fonctionnalités avancées non couvertes par ce chapitre.
    - La FAQ PyUnit (http://pyunit.sourceforge.net/pyunit.html) explique pourquoi les cas de test sont stockés séparément (http://pyunit.sourceforge.net/pyunit.html#WHERE) du code qu ils testent.
    - La *Python Library Reference* (http://www.python.org/doc/current/lib/) résume le module `unittest` (http://www.python.org/doc/current/lib/module−unittest.html).
    - ExtremeProgramming.org (http://www.extremeprogramming.org/) explique pourquoi vous devriez

écrire des tests unitaires (http://www.extremeprogramming.org/rules/unittests.html).

- Le Portland Pattern Repository (http://www.c2.com/cgi/wiki) propose une discussion en cours sur les tests unitaires (http://www.c2.com/cgi/wiki?UnitTests), y compris une définition standard (http://www.c2.com/cgi/wiki?StandardDefinitionOfUnitTest), pourquoi vous devriez écrire les tests unitaires en premier (http://www.c2.com/cgi/wiki?CodeUnitTestFirst) et de nombreuses études de cas (http://www.c2.com/cgi/wiki?UnitTestTrial) en profondeur.

Chapitre 14. Ecriture des tests en premier

Chapitre 15. Refactorisation

- 15.5. Résumé

  - XProgramming.com (http://www.xprogramming.com/) a des liens pour télécharger des *frameworks* de tests unitaires (http://www.xprogramming.com/software.htm) pour de nombreux langages.

Chapitre 16. Programmation fonctionnelle

Chapitre 17. Fonctions dynamiques

- 17.7. plural.py, étape 6

  - PEP 255 (http://www.python.org/peps/pep−0255.html) définit les générateurs.
  - Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/) a de nombreux autres exemples de générateurs (http://www.google.com/search?q=generators+cookbook+site:aspn.activestate.com).

Chapitre 18. Ajustements des performances

- 18.1. Plonger

  - Soundexing and Genealogy (http://www.avotaynu.com/soundex.html) présente une chronologie de l'évolution de Soundex et de ses variantes régionales.

Annexe B. Survol en cinq minutes

Chapitre 1. Installation de Python

- 1.1. Quel Python vous faut−il ?

  La première chose à faire avec Python est de l'installer. Mais est−ce nécéssaire ?
- 1.2. Python sous Windows

  Sous Windows, il y a plusieurs possibilités pour installer Python.
- 1.3. Python sous Mac OS X

  Sous Mac OS X, vous avez deux possibilités, garder la version préinstallée ou installer une nouvelle version. Vous préfèrerez sans doute cette dernière solution.
- 1.4. Python sous Mac OS 9

  Mac OS 9 n'est fournit avec aucune version de Python, mais l'installation en est très simple.
- 1.5. Python sous RedHat Linux

  Téléchargez le dernier RPM Python en allant sur http://www.python.org/ftp/python/2.3.3/ et en sélectionnant le numéro de version le plus haut, puis en sélectionnant le sous−répertoire `rpms/` de cette version. Téléchargez ensuite le RPM ayant le plus haut numéro de version. Vous pouvez l'installer avec la commande **rpm**, comme ci−dessous :
- 1.6. Python sous Debian GNU/Linux

  Si vous avez la chance d'utiliser Debian GNU/Linux, vous pouvez installer Python à l'aide de la commande **apt**.
- 1.7. Installation de Python à partir du source

  Si vous préférez installer à partir du source, vous pouvez télécharger le code source de Python à partir de http://www.python.org/ftp/python/2.3.3/. Sélectionnez le numéro de version le plus haut de la liste, téléchargez le fichier `.tgz` et faites la séquence habituelle **`configure`**, **`make`**, **`make install`**.
- 1.8. L'interface interactive

  Maintenant que Python est installé, nous allons voir en quoi consiste cette interface interactive que vous avez lancé.
- 1.9. Résumé

  Vous devez maintenant avoir une version de Python installée et fonctionnelle.

Chapitre 2. Votre premier programme Python

- 2.1. Plonger

  Voici un programme Python complet et fonctionnel.
- 2.2. Déclaration de fonctions

  Python dispose de fonctions comme la plupart des autre langages, mais il n'a pas de fichiers d'en−tête séparés comme C++ ou de sections `interface`/`implementation` comme Pascal. Lorsque vous avez besoin d'une fonction, vous n'avez qu'à la déclarer et l'écrire.
- 2.3. Documentation des fonctions

Le programme `apihelper.py` et sa sortie devraient maintenant être parfaitement clairs.

Chapitre 5. Les objets et l'orienté objet

Voila pour ce qui est des chicanes techniques des objets. Vous verrez une application réelle des méthodes de classe spéciales au Chapitre 12, qui utilise `getattr` pour créer un mandataire d'un service web distant.

Chapitre 6. Traitement des exceptions et utilisation de fichiers

- 6.1. Traitement des exceptions

  Comme beaucoup de langages orientés objet, Python gère les exception à l'aide de blocs `try...except`.
- 6.2. Les objets−fichier

  Python a une fonction prédéfinie, `open`, pour ouvrir un fichier sur le disque. `open` retourne un objet−fichier qui possède des méthodes et des attributs pour obtenir des informations et manipuler le fichier ouvert.
- 6.3. Itérations avec des boucles for

  Comme la plupart des langages, Python a des boucles `for`. La seule raison pour laquelle vous ne les avez pas vues jusqu à maintenant est que Python sait faire tellement d autre choses que vous n en avez pas besoin aussi souvent.
- 6.4. Utilisation de sys.modules

  Les modules, comme tout le reste en Python, sont des objets. Une fois qu'il a été importé, vous pouvez toujours obtenir une référence à un module à travers le dictionnaire global `sys.modules`.
- 6.5. Travailler avec des répertoires

  Le module `os.path` a de nombreuses fonctions pour manipuler les chemins de fichiers et de répertoires. Ici nous voulons gérer les chemins et lister le contenu d'un répertoire.
- 6.6. Assembler les pièces

  A nouveau, tous les dominos sont en place. Nous avons vu comment chaque ligne de code fonctionne. Maintenant prenons un peut de recul pour voir comment tout cela s assemble.
- 6.7. Résumé

  Le programme `fileinfo.py`, introduit au Chapitre 5; devrait maintenant être parfaitement clair.

Chapitre 7. Expressions régulières

- 7.1. Plonger

  Si ce que vous essayez de faire peut être accompli avec les fonctions de chaînes, utilisez−les. Elles sont rapides et faciles à comprendre et il y a beaucoup d'avantages à un code rapide, simple et lisible. Mais si vous vous rendez compte que vous utilisez un grand nombre de fonctions de chaînes différentes avec des instruction `if` pour les cas particulier, ou si vous les associez à des fonctions `split` et `join` et à des *list comprehension* de manière complexe et illisible, vous devez vous tourner vers les expressions régulières.
- 7.2. Exemple : adresses postales

  Cette série d exemples est inspirée d un problème réel que j ai eu au cours de mon travail, l extraction et la standardisation d adresses postales exportées d un ancien système avant de

appelant une méthode chaque fois qu'un élément est rencontré. (Si vous lisez Chapitre 8, *Traitement du HTML*, cela devrait vous paraître familier, parce que c'est la façon dont le module `sgmllib` fonctionne.) L'autre est appelée DOM ("Document Object Model"), et fonctionne en lisant le document XML dans son entier et en en créant une représentation interne au moyen de classes Python natives reliées dans une structure arborescente. Python dispose de modules standards pour chacun de ces deux traitements, mais ce chapitre ne concernera que l'utilisation du DOM.

- 9.2. Les paquetages

  Analyser un document XML est pour l'heure chose très simple : cela tient sur une ligne de code. Cependant, avant que vous n'abordiez cette ligne de code, une digression s'impose pour parler des paquetages.
- 9.3. Analyser un document XML

  Comme je le disais, analyser un document XML est chose très simple qui tient en une ligne de code. A vous de décider de la suite.
- 9.4. Le standard Unicode

  Le standard Unicode est un mécanisme pour représenter les caractères de tous les différents langages à travers le monde. Quand Python analyse un document XML, toutes les données sont stockées en mémoire au format Unicode.
- 9.5. Rechercher des éléments

  Parcourir des documents XML en s'arrêtant à chaque noeud peut être fastidieux. Si vous recherchez un noeud particulier, enfoui au plus profond de votre document XML, il existe un raccourci pour le retrouver rapidement : `getElementsByTagName`.
- 9.6. Accéder aux attributs d'un élément

  Les éléments XML peuvent avoir un ou plusieurs attributs et il est très facile d'y accéder une fois le document XML analysé.
- 9.7. Transition

  Voilà, c'est tout pour ce qui concerne XML en tant que tel. Le chapitre suivant reprend les mêmes programmes donnés en exemple, mais en mettant l'accent sur certains aspects qui rendent plus souple leur maniement : l'utilisation des flots de données (*streams*) pour le traitement des données en entrée, l'utilisation de `getattr` pour la sélection de méthode et l'utilisation des drapeaux de ligne de commande pour permettre aux utilisateurs de paramétrer le programme sans intervenir dans le code.

Chapitre 10. Des scripts et des flots de données (streams)

- 10.1. Extraire les sources de données en entrée

  L'une des grandes forces de Python repose sur son principe de liaison dynamique, dont un puissant usage est le *pseudo objet−fichier* (*file−like objecti*).
- 10.2. Entrée, sortie et erreur standard

  Les utilisateurs d'UNIX sont déjà familiers avec les concepts d'entrée standard, de sortie standard et d'erreur standard. Cette section s'adresse aux autres.
- 10.3. Mettre en cache la consultation de noeuds

  `kgp.py` recourt à différentes astuces qui peuvent ou non se révéler également utiles dans

votre traitement XML. La première tire avantage de la structure logique des documents en entrée pour construire un cache de noeuds.

- 10.4. Trouver les descendants directs d'un noeud

  Une autre technique bien utile lorsqu'il s'agit d'analyser un document XML consiste à retrouver tous les descendants directs d'un élément particulier. Par exemple, dans les fichiers de grammaire, un élément `ref` peut contenir plusieurs éléments `p`, qui à leur tour peuvent contenir un certain nombre de choses, y compris d'autres éléments `p`. Mais vous ne voulez retrouver que les éléments `p` qui sont les enfants d'un élément `ref` et non les éléments `p` qui sont les enfants d'un autre élément `p`.
- 10.5. Créer des gestionnaires distincts pour chaque type de noeud

  Une troisième astuce bien utile au traitement XML implique la séparation de votre code en fonctions logiques, sur la base des types de noeud et des noms d'élément. Les documents XML analysés sont constitués de divers types de noeud, chacun représenté par un objet Python. La racine d'un document est elle−même représentée par un objet `Document`. L'objet `Document` contient alors un ou plusieurs objets `Element` (pour les balises XML courantes), dont chacun peut contenir d'autres objets `Element`, `Text` (pour les fragments de texte), ou `Comment` (pour les commentaires imbriqués). Python facilite l'écriture d'un sélecteur pour séparer la logique de chaque type de noeud.
- 10.6. Manipuler les arguments de la ligne de commande

  Python supporte complètement la création de programmes qui peuvent être lancés en ligne de commande, à l'aide d'arguments et de drapeaux longs ou cours pour spécifier diverses options. Cela n'est nullement spécifique à XML, mais comme ce script fait grand usage du traitement en ligne de commande, il est très à propos d'y faire ici mention.
- 10.7. Assembler les pièces

  Vous avez déjà parcouru un long chemin. Portez votre regard en arrière et voyez comment rassembler toutes ces étapes.
- 10.8. Résumé

  Python est accompagné de puissantes bibliothèques pour analyser et manipuler des documents XML. Le module `minidom` prend un fichier XML et l'analyse en objets Python, fournissant un accès aléatoire à des éléments arbitraires. Ce chapitre montre encore comment Python peut servir à créer un "véritable" script autonome exécutable en ligne de commande, pourvu de drapeaux et d'arguments de ligne de commande, d'une gestion d'erreur et même de la capacité de récupérer en entrée la redirection du résultat d'un programme antérieur.

Chapitre 11. Services Web HTTP

- 11.1. Plonger

  Nous avons vu le traitement du HTML et le traitement du XML et, au cours de ces chapitres, comment télécharger une page Web et comment analyser du XML à partir d'une URL. Nous allons maintenant plonger dans le sujet plus général des services Web HTTP.
- 11.2. Obtenir des données par HTTP : la mauvaise méthode

  Imaginons que nous souhaitons télécharger une ressource par HTTP, par exemple un fil Atom. Seulement, nous ne voulons pas le télécharger une seule fois, nous voulons le télécharger toutes les heures, pour obtenir les dernières nouvelles sur le site qui fournit le fil. Nous allons le faire de la manière la plus simple, puis nous verrons comment faire mieux.

- 11.3. Fonctionnalités de HTTP

  Il y a cinq fonctionnalités importantes de HTTP que nous devons supporter dans notre programme.
- 11.4. débogage de services Web HTTP

  Pour commencer, nous allons activer les fonctionnalités de débogage de la bibliothèque HTTP de Python pour voir tout ce qui est échangé. Cela nous servira tout au long de ce chapitre, au fur et à mesure que nous rajouterons des fonctionnalités.
- 11.5. Changer la chaîne User−Agent

  La première chose à faire pour améliorer notre client de services Web HTTP est de faire en sorte qu'il s'identifie correctement avec une chaîne `User−Agent`. Pour cela, nous devons aller plus loin qu'`urllib` et plonger dans `urllib2`.
- 11.6. Prise en charge de Last−Modified et ETag

  Maintenant que nous savons comment ajouter des en−têtes HTTP à nos requêtes de services Web, voyons comment prendre en charge les en−têtes `Last−Modified` et `ETag`.
- 11.7. Prise en charge des redirections.

  La prise en charge des redirections temporaires et permanentes se fait avec un autre type de gestionnaire d'URL spécialisé.
- 11.8. Prise en charge des données compressées.

  La dernière fonctionnalité importante du protocole HTTP que nous voulons supporter est la compression. Beaucoup de services Web ont la capacité d'envoyer les données compressées, ce qui qui peut réduire le volume de données envoyées de 60 % ou plus. C'est particulièrement vrai des services Web XML puisque les données XML se compressent très bien.
- 11.9. Assembler les pièces

  Nous avons vu toutes les pièces nécessaires à la construction d'un client de services Web intelligent. Maintenat, voyons comment tout cela s'assemble.
- 11.10. Résumé

  `openanything.py` et ses fonctions devraient être tout à fait clairs maintenant.

Chapitre 12. Services Web SOAP

- 12.1. Plonger

  Vous utilisez Google, n'est−ce pas ? N'avez−vous jamais souhaité accéder aux résultats de recherches Google par la programmation ? Maintenant, vous pouvez le faire, voici un programme Python qui fait des recherches sur Google.
- 12.2. Installation des bibliothèques SOAP

  Contrairement au reste de ce livre, ce chapitre utilise des bibliothèques qui ne sont pas distribuées avec Python.
- 12.3. Premiers pas avec SOAP

  Le coeur de SOAP est la l'appel de fonction distant. Il existe un certain nombre de serveurs publics SOAP qui fournissent des fonctions simples à titre de démonstration.

- 12.4. Débogage de services Web SOAP

  Les bibliothèques SOAP fournissent une manière simple de voir ce qu'il se passe dans les coulisses.
- 12.5. Présentation de WSDL

  Les appels de méthodes locaux sont délégués à la classe `SOAPProxy` qui les converti de manière transparente en appels de méthodes SOAP distants. Comme nous l'avons vu, c'est un gros travail et `SOAPProxy` le fait rapidement et de manière transparente. Mais ce que cette classe ne fait pas est de fournir un mode d'introspection de méthodes.
- 12.6. Introspection de services Web SOAP avec WSDL

  Comme beaucoup de choses dans le domaine des services Web, WSDL a un longue et tortueuse histoire, pleine de controverses politiques et d'intrigue. Je n'aborderais pas du tout cette histoire, je la trouve ennuyeuse à pleurer. Il y a eu des normes concurrentes pour remplir ces fonctions, mais WSDL a gagné, donc apprenons à l'utiliser.
- 12.7. Recherche Google

  Revenons au code d'exemple que nous avons vu au début du chapitre, qui effectue quelque chose de plus intéressant et de plus utile qu'obtenir la température.
- 12.8. Recherche d'erreurs dans les services Web SOAP

  Bien sûr, le monde des services Web SOAP n'est pas différent du reste. Parfois ça ne marche pas.
- 12.9. Résumé

  Les services Web SOAP sont très complexes. La spécification est très ambitieuse et tente de répondre à de nombreux cas d'utilisation différents des services Web. Ce chapitre a abordé quelques uns des cas d'utilisation les plus simples.

Chapitre 13. Tests unitaires

- 13.1. Introduction au chiffres romains

  Dans les chapitres précédents, nous avons "plongé" en regardant immédiatement du code et en essayant de le comprendre le plus vite possible. Maintenant que vous connaissez un peu plus de Python, nous allons prendre un peu de recul et regarder ce qui se passe *avant* que le code soit écrit.
- 13.2. Présentation de romantest.py

  Maintenant que nous avons défini entièrement le comportement que nous attendons de nos fonctions de conversion, nous allons faire quelque chose d un peu inattendu : nous allons écrire une suite de tests qui évalue ces fonctions et s assure qu elle se comporte comme nous voulons qu elles le fassent. Vous avez bien lu, nous allons écrire du code pour tester du code que nous n avons pas encore écrit.
- 13.3. Présentation de romantest.py

  Voici la suite de tests complète de nos fonctions de conversion de chiffres romains, qui n ont pas encore été écrites mais le seront dans `roman.py`. La manière dont tout ça fonctionne ensemble n est pas immédiatement évidente, aucune de ces classes ou méthodes ne se référencent entre elles. Il y a de bonnes raisons à cela, comme nous le verrons bientôt.
- 13.4. Tester la réussite

n avez pas encore écrit.

- 15.2. Gestion des changements de spécification

  Malgré vos meilleurs efforts pour plaquer vos clients au sol et leur extirper une définition de leurs besoins grâce à la menace, les spécifications vont changer. La plupart des clients ne savent pas ce qu ils veulent jusqu à ce qu ils le voient et même ceux qui le savent ne savent pas vraiment comment l exprimer. Et même ceux qui savent l exprimer voudront plus à la version suivante de toute manière. Préparez−vous donc à mettre à jour vos cas de test à mesure que vos spécifications changent.
- 15.3. Refactorisation

  Le meilleur avec des tests unitaires exhaustifs, ce n est pas le sentiment que vous avez quand tous vos cas de test finissent par passer, ni même le sentiment que vous avez quand quelqu un vous reproche d avoir endommagé leur code et que vous pouvez véritablement *prouver* que vous ne l avez pas fait. Le meilleur, c est que les tests unitaires vous permettent la refactorisation continue de votre code.
- 15.4. Postscriptum

  Un lecteur astucieux a lu la section précédente et l a amené au niveau supérieur. Le point le plus compliqué (et pesant le plus sur les performances) du programme tel qu il est écrit actuellement est l expression régulière, qui est nécessaire puisque nous n avons pas d autre moyen de subdiviser un nombre romain. Mais il n y a que 5000 nombres romains, pourquoi ne pas construire une table de référence une fois, puis simplement la lire ? Cette idée est encore meilleure quand on réalise qu il n y a pas besoin d utiliser les expressions régulière du tout. Au fur et à mesure que l on construit la table de référence pour convertir les entiers en nombres romains, on peut construire la table de référence inverse pour convertir les nombres romains en entiers.
- 15.5. Résumé

  Les tests unitaires forment un concept puissant qui, s il est implémenté correctement, peut à la fois réduire les coûts de maintenance et augmenter la flexibilité d un projet à long terme. Il faut aussi comprendre que les tests unitaires ne sont pas une panacée, une baguette magique ou une balle d argent. Ecrire de bons cas de test est difficile et les tenir à jour demande de la discipline (surtout quand les clients réclament à hauts cris la correction de bogues critiques). Les tests unitaires ne sont pas destinés à remplacer d autres formes de tests comme les tests fonctionnels, les tests d intégration et les tests utilisateurs. Mais ils sont réalisables et ils marchent et une fois que vous les aurez vu marcher, vous vous demanderez comment vous avez pu vous en passer.

Chapitre 16. Programmation fonctionnelle

- 16.1. Plonger

  Au Chapitre 13, *Tests unitaires*, vous avez appris la philosophie des tests unitaires. Au Chapitre 14, *Ecriture des tests en premier*, vous avez suivi pas à pas l'implémentation de tests unitaires en Python. Au Chapitre 15, *Refactorisation*, vous avez vu comment les tests unitaires facilitent la refactorisation à grand échelle. Ce chapitre va poursuivre le développement de ces programmes, mais cette fois en mettant l'accent sur des techniques avancées de Python plutôt que sur les test unitaires proprement dit.
- 16.2. Trouver le chemin

  Lorsque vous exécutez des scripts Python depuis la ligne de commande, il est parfois utile de

Annexe C. Trucs et astuces

Chapitre 1. Installation de Python

Chapitre 2. Votre premier programme Python

- 2.1. Plonger

  Dans l'IDE ActivePython sous Windows, vous pouvez exécuter le programme Python que vous êtes en train d'éditer par File−>Run... (**Ctrl−R**). La sortie est affichée dans la fenêtre interactive.

  Dans l'IDE Python sous Mac OS, vous pouvez exécuter un module avec Python−>Run window... (**Cmd−R**) mais il y a une option importante que vous devez activer préalablement. Ouvrez le module dans l'IDE, ouvrez le menu des options des modules en cliquant le triangle noir dans le coin supérieur droit de la fenêtre et assurez−vous que "Run as __main__" est coché. Ce réglage est sauvegardé avec le module, vous n'avez donc à faire cette manipulation qu'une fois par module.

  Sur les systèmes compatibles UNIX (y compris Mac OS X), vous pouvez exécuter un programme Python depuis la ligne de commande : **`python odbchelper.py`**
- 2.2. Déclaration de fonctions

  En Visual Basic, les fonctions (qui retournent une valeur) débutent avec `function` et les sous−routines (qui ne retourne aucune valeur) débutent avec `sub`. Il n'y a pas de sous−routines en Python. Tout est fonction, toute fonction retourne un valeur (même si c'est `None`) et toute fonction débute avec `def`.

  En Java, C++ et autres langage à typage statique, vous devez spécifier les types de données de la valeur de retour d'une fonction ainsi que de chaque paramètre. En Python, vous ne spécifiez jamais de manière explicite le type de quoi que ce soit. En se basant sur la valeur que vous lui assignez, Python gère les types de données en interne.
- 2.3. Documentation des fonctions

  Les triples guillemets sont aussi un moyen simple de définir une chaîne contenant à la fois des guillemets simples et doubles, comme `qq/.../` en Perl.

  Beaucoup d'IDE Python utilisent les `doc string` pour fournir une documentation contextuelle, ainsi lorsque vous tapez le nom d'une fonction, sa `doc string` apparaît dans une bulle d'aide. Cela peut être incroyablement utile, mais cette utilité est liée à la qualité de votre `doc string`.
- 2.4. Tout est objet

  `import` en Python est similaire à `require` en Perl. Une fois que vous importez un module Python, vous accédez à ses fonctions avec *`module.function`*. Une fois que vous incluez un module Perl, vous accédez à ses fonctions avec *`module::function`*.
- 2.5. Indentation du code

  Python utilise le retour chariot pour séparer les instructions, deux points et l'indentation pour séparer les blocs de code. C++ et Java utilisent des points−virgules pour séparer les instructions et des accolades pour séparer les blocs de code.
- 2.6. Test des modules

  A l'instar de C, Python utilise `==` pour la comparaison et `=` pour l'assignement. Mais au contraire de C, Python ne permet pas les assignations dans le corps d'une instruction afin d'éviter qu'une valeur soit accidentellement assignée alors que vous pensiez effectuer une simple comparaison.

  Avec MacPython, il y a une étape supplémentaire pour que l'astuce `if __name__` fonctionne. Ouvrez le menu des options des modules en cliquant le triangle noir dans le coin supérieur droit de la fenêtre et assurez−vous que Run as __main__ est coché.

Chapitre 3. Types prédéfinis

- 3.1. Présentation des dictionnaires

  En Python, un dictionnaire est comme une table de hachage en Perl. En Perl, les variables qui stockent des tables de hachage débutent toujours par le caractère %. En Python vous pouvez nommer votre variable comme bon vous semble et Python se chargera de la gestion du typage.

  Un dictionnaire Python est similaire à une instance de la classe `Hashtable` en Java.

  Un dictionnaire Python est similaire à une instance de l'objet `Scripting.Dictionnary` en Visual Basic.
- 3.1.2. Modification des dictionnaires

  Les dictionnaires ne sont liés à aucun concept d'ordonnancement des éléments. Il est incorrect de dire que les élément sont "dans le désordre", il ne sont tout simplement pas ordonnés. C'est une distinction importante qui vous ennuiera lorsque vous souhaiterez accéder aux éléments d'un dictionnaire d'une façon spécifique et reproductible (par exemple par ordre alphabétique des clés). C'est possible, mais cette fonctionalite n'est pas integrée au dictionnaire.
- 3.2. Présentation des listes

  Une liste en Python est comme un tableau Perl. En Perl, les variables qui stockent des tableaux débutent toujours par le caractère @, en Python vous pouvez nommer votre variable comme bon vous semble et Python se chargera de la gestion du typage.

  Une liste Python est bien plus qu'un tableau en Java (même s'il peut être utilisé comme tel si vous n'attendez vraiment rien de mieux de la vie). Une meilleure analogie serait la classe `ArrayList`, qui peut contenir n'importe quels objets et qui croît dynamiquement au fur et à mesure que de nouveaux éléments y sont ajoutés.
- 3.2.3. Recherche dans une liste

  Avant la version 2.2.1, Python n'avait pas de type booléen. Pour compenser cela, Python acceptait pratiquement n'importe quoi dans un contexte requérant un booléen (comme une instruction `if`), en fonction des règles suivantes :
  - `0` est faux, tous les autres nombres sont vrai.
  - Une chaîne vide (`""`) est faux, toutes les autres chaînes sont vrai.
  - Une liste vide (`[]`) est faux, toutes les autres listes sont vrai.
  - Un tuple vide (`()`) est faux, tous les autres tuples sont vrai.
  - Un dictionnaire vide (`{}`) est faux, tous les autres dictionnaires sont vrai.

  Ces règles sont toujours valides en Python 2.3.3 et au−delà, mais vous pouvez maintenant utiliser un véritable booléen, qui a pour valeur `True` ou `False`. Notez la majuscule, ces valeurs comme tout le reste en Python, sont sensibles à la casse.
- 3.3. Présentation des tuples

  Les tuples peuvent être convertis en listes et vice−versa. La fonction prédéfinie `tuple` prends une liste et retourne un tuple contenant les mêmes éléments et la fonction `list` prends un tuple et retourne une liste. En fait, `tuple` gèle une liste et `list` dégèle un tuple.
- 3.4. Définitions de variables

  Lorsqu'une commande est étalée sur plusieurs lignes avec le marqueur de continuation de ligne ("\"), les lignes suivantes peuvent être indentées de n'importe qu'elle manière, les règles d'indentation strictes habituellement utilisées en Python ne s'appliquent pas. Si votre IDE Python indente automatiquement les lignes continuées, vous devriez accepter ses réglages par défauts sauf raison impérative.
- 3.5. Formatage de chaînes

  Le formatage de chaînes en Python utilise la même syntaxe que la fonction C `sprintf`.
- 3.7. Jointure de listes et découpage de chaînes

  La méthode `join` ne fonctionne qu'avec des listes de chaînes; elle n'applique pas la conversion de types. La `join`ture d'une liste comprenant au moins un élément non−chaîne déclenchera une exception.

une_chaîne.split(*délimiteur*, 1) est une technique utile pour chercher une sous−chaîne dans une chaîne et utiliser tout ce qui précède cette sous−chaîne (le premier élément de la liste retournée) et tout ce qui la suit (le second élément de la liste retournée).

Chapitre 4. Le pouvoir de l introspection

- 4.2. Arguments optionnels et nommés

  La seule chose que vous avez à faire pour appeler une fonction est de spécifier une valeur (d une manière ou d une autre) pour chaque argument obligatoire, la manière et l ordre dans lequel vous le faites ne dépendent que de vous.
- 4.3.3. Fonctions prédéfinies

  Python est fourni avec d excellent manuels de référence que vous devriez parcourir de manière exhaustive pour apprendre tous les modules que Python offre. Mais alors que dans la plupart des langages vous auriez à vous référer constamment aux manuels (ou aux pages man, ou pire, à MSDN) pour vous rappeler l usage de ces modules, Python est en grande partie auto−documenté.
- 4.7. Utiliser des fonctions lambda

  Les fonctions `lambda` sont une question de style. Les utiliser n est jamais une nécessité, partout où vous pouvez les utiliser, vous pouvez utiliser une fonction ordinaire. Je les utilise là où je veux incorporer du code spécifique et non réutilisable sans encombrer mon code de multiples fonctions d une seule ligne.
- 4.8. Assembler les pièces

  En SQL, vous devez utiliser `IS NULLmethod` au lieu de `= NULL` pour la comparaison d une valeur nulle. En Python, vous pouvez utiliser aussi bien `== None` que `is None`, mais `is None` est plus rapide.

Chapitre 5. Les objets et l'orienté objet

- 5.2. Importation de modules avec from module import

  `from` *`module`* `import *` en Python est comme `use` *`module`* in Perl. `import` *`module`* en Python est comme `require` *`module`* en Perl.

  `from` *`module`* `import *` en Python est comme `import` *`module`*`.*` en Java. `import` *`module`* en Python est comme `import` *`module`* en Java.

  Utilisez `from module import *` avec modération, il rend plus difficile de déterminer l'origine d'une fonction ou d'un attribut, ce qui rend le débogage et la refactorisation plus difficiles.
- 5.3. Définition de classes

  L'instruction `pass` de Python est comme une paire d'accolades vide (`{}`) en Java ou C.

  En Python, l'ancêtre d'une classe est simplement indiqué entre parenthèses immédiatement après le nom de la classe. Il n'y a pas de mot clé spécifique comme `extends` en Java.
- 5.3.1. Initialisation et écriture de classes

  Par convention, le premier argument d'une méthode de classe (la référence à l'instance en cours) est appelé `self`. Cet argument remplit le rôle du mot réservé `this` en C++ ou Java, mais `self` n'est pas un mot réservé de Python, seulement une convention de nommage. Cependant, veuillez ne pas l'appeler autre chose que `self`, c'est une très forte convention.
- 5.3.2. Quand utiliser self et __init__

  Les méthodes `__init__` sont optionnelles, mais quand vous en définissez une, vous devez vous rappeler d'appeler explicitement la méthode `__init__` de l'ancêtre de la classe. C'est une règle plus générale : quand un descendant veut étendre le comportement d'un ancêtre, la méthode du descendant doit appeler la méthode de l'ancêtre explicitement au moment approprié, avec les arguments appropriés.
- 5.4. Instantiation de classes

En Python, vous appelez simplement une classe comme si c'était une fonction pour créer une nouvelle instance de la classe. Il n'y a pas d'opérateur `new` explicite comme pour C++ ou Java.

- 5.5. UserDict : une classe enveloppe

Dans l'IDE ActivePython sous Windows, vous pouvez ouvrir rapidement n'importe quel module dans votre chemin de bibliothèques avec File−>Locate... (**Ctrl−L**).

Java et Powerbuilder supportent la surcharge de fonction par liste d'arguments : une classe peut avoir différentes méthodes avec le même nom mais avec un nombre différent d'arguments ou des arguments de type différent. D'autres langages (notamment PL/SQL) supportent même la surcharge de fonction par nom d'argument : une classe peut avoir différentes méthodes avec le même nom et le même nombre d'arguments du même type mais avec des noms d'arguments différents. Python ne supporte ni l'une ni l'autre, il n'a tout simplement aucune forme de surcharge de fonction. Les méthodes sont définies uniquement par leur nom et il ne peut y avoir qu'une méthode par classe avec le même nom. Donc si une classe descendante a une méthode `__init__`, elle redéfinit *toujours* la méthode `__init__` de la classe ancêtre, même si la descendante la définit avec une liste d'arguments différente. Et la même règle s'applique pour toutes les autres méthodes.

Guido, l'auteur originel de Python, explique la redéfinition de méthode de cette manière : "Les classes dérivées peuvent redéfinir les méthodes de leur classes de base. Puisque les méthodes n'ont pas de privilèges spéciaux lorsqu'elles appellent d'autres méthodes du même objet, une méthode d'une classe de base qui appelle une autre méthode définie dans cette même classe de base peut en fait se retrouver à appeler une méthode d'une classe dérivée qui la redéfini (pour les programmeurs C++ cela veut dire qu'en Python toutes les méthodes sont virtuelles)." Si cela n'a pas de sens pour vous (personellement, je m'y perd complètement) vous pouvez ignorer la question. Je me suis juste dit que je ferais circuler l'information.

Assignez toujours une valeur initiale à toutes les données attributs d'une instance dans la méthode `__init__`. Cela vous épargera des heures de débogage plus tard, à la poursuite d'exceptions `AttributeError` pour cause de référence à des attributs non−initialisés (et donc non−existants).

Dans les versions de Python antérieures à la 2.2, vous ne pouviez pas directement dériver les types de données prédéfinis comme les chaînes, les listes et les dictionnaires. Pour compenser cela, Python est fourni avec des classes enveloppes qui reproduisent le comportement de ces types de données prédéfinis : `UserString`, `UserList` et `UserDict`. En utilisant un mélange de méthodes ordinaires et spéciales, la classe `UserDict` fait une excellente imitation d'un dictionnaire, mais c'est juste une classe comme les autres, vous pouvez donc la dériver pour créer des classes personalisées semblables à un dictionnaire comme `FileInfo`. En Python 2.2 et suivant, vous pourriez récrire l'exemple de ce chapitre de manière à ce que `FileInfo` hérite directement de `dict` au lieu de `UserDict`. Cependant, vous devriez quand même lire l'explication du fonctionnement de `UserDict` au cas où vous auriez besoin d'implémenter ce genre d'objet enveloppe ou au cas où vous auriez à travailler avec une version de Python antérieure à la 2.2.

- 5.6.1. Lire et écrire des éléments

Lorsque vous accédez à des données attributs dans une classe, vous devez qualifier le nom de l'attribut : `self.`*`attribute`*. Lorsque vous appelez d'autres méthodes dans une classe, vous devez qualifier le nom de la méthode : `self.`*`method`*.

- 5.7. Méthodes spéciales avancées

En Java, vous déterminez si deux variables de chaînes référencent la même zone mémoire à l'aide de `str1 == str2`. On appelle cela *identité* des objets et la syntaxe Python en est `str1 is str2`. Pour comparer des valeurs de chaînes en Java, vous utiliseriez `str1.equals(str2)`, en Python, vous utiliseriez `str1 == str2`. Les programmeurs Java qui ont appris que le monde était rendu meilleur par le fait que `==` en Java fasse une comparaison par identité plutôt que par valeur peuvent avoir des difficultés à s'adapter au fait que Python est dépourvu d'un tel piège.

Alors que les autres langages orientés objet ne vous laissent définir que le modèle physique d'un objet ("cet objet a une méthode `GetLength`"), les méthodes spéciales de Python comme `__len__` vous permettent de

définir le modèle logique d'un objet ("cet objet a une longueur").

- 5.8. Attributs de classe

  En Java, les variables statiques (appelées attributs de classe en Python) aussi bien que les variables d'instance (appelées données attributs en Python) sont définies immédiatement après la définition de la classe (avec le mot−clé `static` pour les premières). En Python, seuls les attributs de classe peuvent être définis à cet endroit, les données attributs sont définies dans la méthode `__init__`.

  Il n'y a pas de constantes en Python. Tout est modifiable en faisant un effort. C'est en accord avec un des principes essentiels de Python : un mauvais comportement doit être découragé mais pas interdit. Si vous voulez vraiment changer la valeur de `None`, vous pouvez le faire, mais ne venez pas vous plaindre que votre code est impossible à déboguer.

- 5.9. Fonctions privées

  En Python, toutes les méthodes spéciales (comme `__setitem__`) et les attributs prédéfinis (comme `__doc__`) suivent une convention standard : il commencent et se terminent par deux caractères de soulignement. Ne nommez pas vos propres méthodes et attributs de cette manière, cela n'apporterait que de la confusion pour vous et les autres.

Chapitre 6. Traitement des exceptions et utilisation de fichiers

- 6.1. Traitement des exceptions

  Python utilise `try...except` pour gérer les exceptions et `raise` pour les générer. Java et C++ utilisent `try...catch` pour gérer les exceptions et `throw` pour les générer.

- 6.5. Travailler avec des répertoires

  A chaque fois que c est possible, vous devriez utiliser les fonction de `os` et `os.path` pour les manipulations de fichier, de répertoire et de chemin. Ces modules enveloppent des modules spécifiques aux plateformes, les fonctions comme `os.path.split` marchent donc sous UNIX, Windows, Mac OS et toute autre plateforme supportée par Python.

Chapitre 7. Expressions régulières

- 7.4. Utilisation de la syntaxe {n,m}

  Il n y a aucun moyen de déterminer par un programme que deux expressions régulières sont équivalentes. Le mieux que vous puissiez faire est d écrire de nombreux cas de test pour vérifier que leur comportements sont identiques pour les entrées pertinentes. Nous discuterons plus en détail l écriture de cas de tests plus loin dans le livre.

Chapitre 8. Traitement du HTML

- 8.2. Présentation de sgmllib.py

  Python 2.0 avait un bogue qui empêchait `SGMLParser` de reconnaître les déclarations (`handle_decl` n était jamais appelé), ce qui veut dire que les `DOCTYPE`s étaient ignorés silencieusement. Ce bogue est corrigé dans Python 2.1.

  Dans l IDE ActivePython sous Windows, vous pouvez spécifier des arguments de ligne de commande dans la boîte de dialogue "Run script". Séparez les différents arguments par des espaces.

- 8.4. Présentation de BaseHTMLProcessor.py

  La spécification HTML exige que tous les éléments non−HTML (comme le JavaScript côté client) soient compris dans des commentaires HTML, mais toutes les pages web ne le font pas (et les navigateurs web récents ne l exigent pas). `BaseHTMLProcessor`, lui, l exige, si le script n est correctement encadré dans un commentaire, il sera analysé comme s il était du code HTML. Par exemple, si le script contient des

signes inférieurs à ou égal, `SGMLParser` peut considérer à tort qu il a trouvé des balises et des attributs. `SGMLParser` convertit toujours les noms de balises et d attributs en minuscules, ce qui peut empêcher la bonne exécution du script et `BaseHTMLProcessor` entoure toujours les valeurs d attributs entre guillemets (même si le document HTML n en utilisait pas ou utilisait des guillemets simples), ce qui empêchera certainement l exécution du script. Protégez toujours vos script côté client par des commentaires HTML.

- 8.5. locals et globals

  Python 2.2 a introduit une modification légère mais importante qui affecte l ordre de recherche dans les espaces de noms : les portées imbriquées. Dans les versions précédentes de Python, lorsque vous référenciez une variable dans une fonction imbriquée ou une fonction `lambda`, Python recherchait la variable dans l espace de noms de la fonction (imbriquée ou `lambda`) en cours, puis dans l espace de noms du module. Python 2.2 recherche la variable dans l espace de noms de la fonction (imbriquée ou `lambda`) en cours, *puis dans l espace de noms de la fonction parente*, puis dans l espace de noms du module. Python 2.1 peut adopter l'un ou l'autre de ces comportements, par défaut il fonctionne comme Python 2.0, mais vous pouvez ajouter la ligne de code suivante au début de vos modules pour les faire fonctionner comme avec Python 2.2 :

  ```
  from __future__ import nested_scopes
  ```

  A l aide des fonctions `locals` et `globals`, vous pouvez obtenir la valeur d une variable quelconque dynamiquement, en fournissant le nom de la variable sous forme de chaîne. C est une fonctionnalité semblable à celle de la fonction `getattr`, qui vous permet d accéder à une fonction quelconque dynamiquement en fournissant le nom de la fonction sous la forme d une chaîne.

- 8.6. Formatage de chaînes à l aide d un dictionnaire

  L utilisation du formatage de chaîne à l aide d un dictionnaire avec `locals` est une manière pratique de rendre des expressions de formatage complexes plus lisibles, mais elle a un prix. Il y a une petite baisse de performance due à l appel de `locals`, puisque `locals` effectue une copie de l espace de noms local.

Chapitre 9. Traitement de données XML

- 9.2. Les paquetages

  Un paquetage est un répertoire pourvu du fichier `__init__.py`. Le fichier `__init__.py` définit les attributs et les méthodes du paquetage. Il n'est cependant pas tenu de définir quoi que ce soit; ce peut simplement être un fichier vide, mais il se doit d'être présent. Et si `__init__.py` n'existe pas, le répertoire reste un répertoire, pas un paquetage et ne peut ni être importé ni contenir de modules ou de paquetages imbriqués.

- 9.6. Accéder aux attributs d'un élément

  Cette section peut paraître un peu confuse dans le mesure où des terminologies se recouvrent. Les éléments d'un document XML ont des attributs, mais les objets Python ont aussi des attributs. Lorsque vous analysez un document XML, vous obtenez un paquet d'objets Python qui représentent l'ensemble des parties du document XML et certains de ces objets Python représentent les attributs des éléments XML. Mais les objets (Python) qui représentent les attributs (XML) possèdent également des attributs (Python), qui sont utilisés pour accéder à diverses parties de l'attribut (XML) que l'objet représente. Je vous avais bien dit que tout cela prêtait à confusion. Je suis ouvert à toute suggestion qui permettrait de les distinguer plus clairement.

  A l'instar d'un dictionnaire, les attributs d'un élément XML ne sont pas ordonnés. Il se peut que les attributs *apparaissent* dans un certain ordre dans le document XML original et qu'ils *apparaissent* dans un certain ordre quand le document XML est analysé en objets Python, mais ces ordonnancements sont arbitraires et n'ont aucune signification particulière. Vous devriez toujours accéder aux attributs individuels par leur nom, comme les clés d'un dictionnaire.

Chapitre 10. Des scripts et des flots de données (streams)

Chapitre 11. Services Web HTTP

- 11.6. Prise en charge de Last−Modified et ETag

  Dans ces exemples, le serveur HTTP supporte à la fois les en−têtes `Last-Modified` et `ETag`, mais ce n'est pas le cas de tous les serveurs. Pour vos clients de services Web, vous devez prévoir de supporter les deux et programmer de manière défensive au cas ou un serveur ne supporterais que l'un des deux, ou aucun.

Chapitre 12. Services Web SOAP

Chapitre 13. Tests unitaires

- 13.2. Présentation de romantest.py

  `unittest` est inclus dans Python 2.1 et versions ultérieures. Les utilisateurs de Python 2.0 peuvent le télécharger depuis `pyunit.sourceforge.net` (http://pyunit.sourceforge.net/).

Chapitre 14. Ecriture des tests en premier

- 14.3. roman.py, étape 3

  La chose la plus importante que des tests unitaires complets vous disent est quand vous arrêter d écrire du code. Quand tous les tests unitaires d une fonction passent, arrêtez d écrire le code de la fonction. Quand tous les tests d un module passent, arrêtez d écrire le code du module.
- 14.5. roman.py, étape 5

  Quand tous vos tests passent, arrêtez d écrire du code.

Chapitre 15. Refactorisation

- 15.3. Refactorisation

  A chaque fois que vous allez utiliser une expression régulière plus d une fois, il vaut mieux la compiler pour obtenir un objet motif et appeler ses méthodes directement.

Chapitre 16. Programmation fonctionnelle

- 16.2. Trouver le chemin

  Les chemins et noms de fichier que vous passez à `os.path.abspath` n'ont pas besoin d'exister sur le disque.

  `os.path.abspath` ne construit pas seulement des chemins complets, il les normalise. Cela signifie que si vous êtes dans le répertoire `/usr/`, `os.path.abspath('bin/../local/bin')` retournera `/usr/local/bin`. Il normalise le chemin en le rendant aussi simple que possible. Si vous voulez normaliser un chemin de cette manière sans le transformer en chemin complet, utilisez `os.path.normpath`.

  Comme les autres fonctions des modules `os` et `os.path`, `os.path.abspath` est multiplate−forme. Les résultats que vous obtiendrez seront légèrement ifférents si vous utilisez Windows (qui utilise le le *backslash* comme séparateur de chemin) ou Mac OS (qui utilise les deux points), mais le script fonctionnera. C'est là le rôle du module `os`.

Chapitre 17. Fonctions dynamiques

Chapitre 18. Ajustements des performances

- 18.2. Utilisation du module timeit

  Vous pouvez utiliser le module `timeit` en ligne de commande pour tester un programme Python existant sans en modifier le code. Voir http://docs.python.org/lib/node396.html pour la documentation des paramètres de ligne de commande.

  Le module `timeit` ne fonctionne que si vous savez déjà quelle partie de votre code optimiser. Si vous avez un programme Python plus grand et que vous ne savez pas où se trouve le problème de performances, allez voir le module `hotshot`. (http://docs.python.org/lib/module−hotshot.html)

Annexe D. Liste des exemples

Annexe E. Historique des révisions

Historique des versions	
Version 5.4	2004−05−20
• Added Section 12.1, «Plonger». • Added Section 12.2, «Installation des bibliothèques SOAP». • Added Section 12.3, «Premiers pas avec SOAP». • Added Section 12.4, «Débogage de services Web SOAP». • Added Section 12.5, «Présentation de WSDL». • Added Section 12.6, «Introspection de services Web SOAP avec WSDL». • Added Section 12.7, «Recherche Google». • Added Section 12.8, «Recherche d'erreurs dans les services Web SOAP». • Added Section 12.9, «Résumé». • Incorporated technical reviewer revisions in Chapitre 16, *Programmation fonctionnelle* and Chapitre 18, *Ajustements des performances*.	
Version 5.3	2004−05−12
• Added `isalpha()` example to Section 18.3, «Optimisation d'expressions régulières». Thanks, Paul. • Incorporated copyediting revisions into Chapitre 5, *Les objets et l'orienté objet* and Chapitre 6, *Traitement des exceptions et utilisation de fichiers*. • Fixed URL of Section 9.7, «Transition».	
Version 5.2	2004−05−09
• Fixed URL of Section 14.1, «roman.py, étape 1». • Added Section 18.1, «Plonger». • Added Section 18.2, «Utilisation du module timeit». • Added Section 18.3, «Optimisation d'expressions régulières». • Added Section 18.4, «Optimisation de la lecture d'un dictionnaire». • Added Section 18.5, «Optimisation des opérations sur les listes». • Added Section 18.6, «Optimisation des manipulations de chaînes». • Added Section 18.7, «Résumé».	
Version 5.1	2004−05−05
• Clarified Exemple 7.7, «Rechercher les dizaines» and Exemple 7.8, «Validation des chiffres romains avec {n,m}». • Clarified Exemple 7.10, «Trouver des numéros». • Fixed typo in Exemple 11.6, «Test de Last−Modified». Thanks, Jesir. • Fixed typo in Exemple 3.11, «Différence entre extend et append». Thanks, Daniel. • Incorporated technical reviewer revisions. • Incorporated copy editor revisions in Chapitre 1, *Installation de Python*, Chapitre 2, *Votre premier programme Python*, Chapitre 3, *Types prédéfinis*, and Chapitre 4, *Le pouvoir de l introspection*.	
Version 5.0	2004−04−16
• Added Section 11.1, «Plonger». • Added Section 11.2, «Obtenir des données par HTTP : la mauvaise méthode». • Added Section 11.3, «Fonctionnalités de HTTP». • Added Section 11.4, «débogage de services Web HTTP». • Added Section 11.5, «Changer la chaîne User−Agent».	

- Added Section 11.6, «Prise en charge de Last−Modified et ETag».
- Added Section 11.7, «Prise en charge des redirections.».
- Added Section 11.8, «Prise en charge des données compressées.».
- Added Section 11.9, «Assembler les pièces».
- Added Section 11.10, «Résumé».
- Added Exemple 3.11, «Différence entre extend et append».
- Changed descriptions of how to download Python throughout Chapitre 1, *Installation de Python* to be more generic and less version−specific.
- Changed references of "module" to "program" in Section 2.1, «Plonger» and Section 2.4, «Tout est objet» since we haven't explained modules yet.
- Added explicit instructions in Section 2.4, «Tout est objet» for the reader to open their Python IDE and follow along with the examples.
- Changed all examples and descriptions that referred to truth values `1` and `0` to refer to `True` and `False`.
- Updated Exemple 3.22, «Formatage de chaîne et concaténation» to show new Python 2.3 `TypeError` message.
- Fixed typo in Exemple 17.19, «Utilisation des générateurs à la place de la récursion».
- Fixed typo in Section 7.7, «Résumé».
- Fixed typo in Exemple 17.9, «plural4.py».

Version 4.9	2004−03−25

- Finished Section 16.7, «Assembler les pièces».
- Added Section 16.8, «Résumé».
- Split unit testing introduction into two chapters, Chapitre 13, *Tests unitaires* and Chapitre 14, *Ecriture des tests en premier*.
- Fixed typo in Exemple 17.12, «plural4.py, suite et fin».
- Fixed typo in Exemple 17.18, «Présentation des générateurs».

Version 4.8	2004−03−25

- Finished Section 17.7, «plural.py, étape 6».
- Finished Section 17.8, «Résumé».
- Fixed broken links in Annexe A, *Pour en savoir plus*, Annexe B, *Survol en cinq minutes*, Annexe C, *Trucs et astuces*, Annexe D, *Liste des exemples*.

Version 4.7	2004−03−21

- Added Section 17.1, «Plonger».
- Added Section 17.2, «plural.py, étape 1».
- Added Section 17.3, «plural.py, étape 2».
- Added Section 17.4, «plural.py, étape 3».
- Added Section 17.5, «plural.py, étape 4».
- Added Section 17.6, «plural.py, étape 5».
- Added Section 17.7, «plural.py, étape 6» (unfinished).
- Added Section 17.8, «Résumé» (unfinished).

Version 4.6	2004−03−14

- Finished Section 7.4, «Utilisation de la syntaxe {n,m}».
- Finished Section 7.5, «Expressions régulières détaillées».
- Finished Section 7.6, «Etude de cas : reconnaissance de numéros de téléphone».
- Expanded Section 7.7, «Résumé».

Version 4.5	2004−03−07

- Added Section 7.1, «Plonger».
- Added Section 7.4, «Utilisation de la syntaxe {n,m}» (incomplete).
- Added Section 7.5, «Expressions régulières détaillées» (incomplete).
- Added Section 7.6, «Etude de cas : reconnaissance de numéros de téléphone» (incomplete).
- Added Section 7.7, «Résumé».
- Moved Section 7.2, «Exemple : adresses postales» and Section 7.3, «Exemple : chiffres romains» to regular expressions chapter.
- Added Exemple 6.20, «Liste du contenu d'un répertoire avec glob».
- Added Exemple 6.7, «Ecriture dans un fichier».
- Added Exemple 5.11, «Dériver une classe directement du type prédéfini dict».
- Added Exemple 10.11, «Afficher un message sur stderr».
- Added Exemple 4.12, «Création d'un sélecteur avec getattr» and Exemple 4.13, «Valeurs par défaut de getattr».
- Added Exemple 2.6, «Instructions if».
- Added Exemple 3.23, «Formatage de nombres».
- Split Chapitre 5, *Les objets et l'orienté objet* into 2 chapters: Chapitre 5, *Les objets et l'orienté objet* and Chapitre 6, *Traitement des exceptions et utilisation de fichiers*.
- Split Chapitre 9, *Traitement de données XML* into 2 chapters: Chapitre 9, *Traitement de données XML* and Chapitre 10, *Des scripts et des flots de données (streams)*.
- Split Chapitre 13, *Tests unitaires* into 2 chapters: Chapitre 13, *Tests unitaires* and Chapitre 15, *Refactorisation*.
- Renamed `help` to `info` in Chapitre 4, *Le pouvoir de l introspection*.
- Fixed incorrect back−reference in Section 8.5, «locals et globals».
- Fixed broken example links in Section 8.1, «Plonger».
- Fixed missing line in example in Section 9.1, «Plonger».
- Fixed typo in Section 8.2, «Présentation de sgmllib.py».

Version 4.4	2003−10−08

- Added Section 1.1, «Quel Python vous faut−il ?».
- Added Section 1.2, «Python sous Windows».
- Added Section 1.3, «Python sous Mac OS X».
- Added Section 1.4, «Python sous Mac OS 9».
- Added Section 1.5, «Python sous RedHat Linux».
- Added Section 1.6, «Python sous Debian GNU/Linux».
- Added Section 1.7, «Installation de Python à partir du source».
- Added Section 1.9, «Résumé».
- Removed preface.
- Fixed typo in Exemple 3.27, «Sortie de odbchelper.py».
- Added link to PEP 257 in Section 2.3, «Documentation des fonctions».
- Fixed link to *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) in Section 3.4.2, «Assignation simultanée de plusieurs valeurs».
- Added note about implied assert in Section 3.3, «Présentation des tuples».

Version 4.3	2003−09−28

- Added Section 16.6, «Importation dynamique de modules».
- Added Section 16.7, «Assembler les pièces» (incomplete).
- Fixed links in Annexe F, *A propos de ce livre*.

Version 4.2.1	2003−09−17

- Fixed links on index page.
- Fixed syntax highlighting.

Version 4.2	2003−09−12

- Fixed typos in Section 16.4, «La mutation de liste revisitée», Section 16.3, «Le filtrage de liste revisité», Section 7.2, «Exemple : adresses postales», and Section 10.6, «Manipuler les arguments de la ligne de commande». Thanks, Doug.
- Fixed external link in Section 5.3, «Définition de classes». Thanks, Harry.
- Changed wording at the end of Section 4.5, «Filtrage de listes». Thanks, Paul.
- Added sentence in Section 13.5, «Tester l échec» to make it clearer that we're passing a function to `assertRaises`, not a function name as a string. Thanks, Stephen.
- Fixed typo in Section 8.8, «Présentation de dialect.py». Thanks, Wellie.
- Fixed links to dialectized examples.
- Fixed external link to the history of Roman numerals. Thanks to many concerned Roman numeral fans around the world.

Version 4.1	2002−07−28

- Added Section 10.3, «Mettre en cache la consultation de noeuds».
- Added Section 10.4, «Trouver les descendants directs d'un noeud».
- Added Section 10.5, «Créer des gestionnaires distincts pour chaque type de noeud».
- Added Section 10.6, «Manipuler les arguments de la ligne de commande».
- Added Section 10.7, «Assembler les pièces».
- Added Section 10.8, «Résumé».
- Fixed typo in Section 6.5, «Travailler avec des répertoires». It's `os.getcwd()`, not `os.path.getcwd()`. Thanks, Abhishek.
- Fixed typo in Section 3.7, «Jointure de listes et découpage de chaînes». When evaluated (instead of printed), the Python IDE will display single quotes around the output.
- Changed `str` example in Section 4.8, «Assembler les pièces» to use a user−defined function, since Python 2.2 obsoleted the old example by defining a `doc string` for the built−in dictionary methods. Thanks Eric.
- Fixed typo in Section 9.4, «Le standard Unicode», "anyway" to "anywhere". Thanks Frank.
- Fixed typo in Section 13.6, «Tester la cohérence», doubled word "accept". Thanks Ralph.
- Fixed typo in Section 15.3, «Refactorisation», `C?C?C?` matches 0 to 3 `C` characters, not 4. Thanks Ralph.
- Clarified and expanded explanation of implied slice indices in Exemple 3.9, «Raccourci pour le découpage». Thanks Petr.
- Added historical note in Section 5.5, «UserDict : une classe enveloppe» now that Python 2.2 supports subclassing built−in datatypes directly.
- Added explanation of `update` dictionary method in Exemple 5.9, «Definition de la classe UserDict». Thanks Petr.
- Clarified Python's lack of overloading in Section 5.5, «UserDict : une classe enveloppe». Thanks Petr.
- Fixed typo in Exemple 8.8, «Présentation de BaseHTMLProcessor». HTML comments end with two dashes and a bracket, not one. Thanks Petr.
- Changed tense of note about nested scopes in Section 8.5, «locals et globals» now that Python 2.2 is out. Thanks Petr.
- Fixed typo in Exemple 8.14, «Formatage à l aide d un dictionnaire dans BaseHTMLProcessor.py»; a space should have been a non−breaking space. Thanks Petr.
- Added title to note on derived classes in Section 5.5, «UserDict : une classe enveloppe». Thanks Petr.
- Added title to note on downloading `unittest` in Section 15.3, «Refactorisation». Thanks Petr.
- Fixed typesetting problem in Exemple 16.6, «Exécuter les scripts dans le répertoire en cours»; tabs should have been spaces, and the line numbers were misaligned. Thanks Petr.
- Fixed capitalization typo in the tip on truth values in Section 3.2, «Présentation des listes». It's `True` and `False`, not `true` and `false`. Thanks to everyone who pointed this out.
- Changed section titles of Section 3.1, «Présentation des dictionnaires», Section 3.2, «Présentation des listes», and Section 3.3, «Présentation des tuples». "Dictionaries 101" was a cute way of saying that this section was

an beginner's introduction to dictionaries. American colleges tend to use this numbering scheme to indicate introductory courses with no prerequisites, but apparently this is a distinctly American tradition, and it was unnecessarily confusing my international readers. In my defense, when I initially wrote these sections a year and a half ago, it never occurred to me that I would have international readers.
- Upgraded to version 1.52 of the DocBook XSL stylesheets.
- Upgraded to version 6.52 of processeur XSLT SAXON de Michael Kay.
- Various accessibility−related stylesheet tweaks.
- Somewhere between this revision and the last one, she said yes. The wedding will be next spring.

Version 4.0−2	2002−04−26

- Fixed typo in Exemple 4.15, «Présentation de and».
- Fixed typo in Exemple 2.4, «Chemin de recherche d'import».
- Fixed Windows help file (missing table of contents due to base stylesheet changes).

Version 4.0	2002−04−19

- Expanded Section 2.4, «Tout est objet» to include more about import search paths.
- Fixed typo in Exemple 3.7, «Indices de liste négatifs». Thanks to Brian for the correction.
- Rewrote the tip on truth values in Section 3.2, «Présentation des listes», now that Python has a separate boolean datatype.
- Fixed typo in Section 5.2, «Importation de modules avec from module import» when comparing syntax to Java. Thanks to Rick for the correction.
- Added note in Section 5.5, «UserDict : une classe enveloppe» about derived classes always overriding ancestor classes.
- Fixed typo in Exemple 5.18, «Modification des attributs de classe». Thanks to Kevin for the correction.
- Added note in Section 6.1, «Traitement des exceptions» that you can define and raise your own exceptions. Thanks to Rony for the suggestion.
- Fixed typo in Exemple 8.17, «Traitement de balises spécifiques». Thanks for Rick for the correction.
- Added note in Exemple 8.18, «SGMLParser» about what the return codes mean. Thanks to Howard for the suggestion.
- Added `str` function when creating `StringIO` instance in Exemple 10.6, «openAnything». Thanks to Ganesan for the idea.
- Added link in Section 13.3, «Présentation de romantest.py» to explanation of why test cases belong in a separate file.
- Changed Section 16.2, «Trouver le chemin» to use `os.path.dirname` instead of `os.path.split`. Thanks to Marc for the idea.
- Added code samples (`piglatin.py`, `parsephone.py`, and `plural.py`) for the upcoming regular expressions chapter.
- Updated and expanded list of Python distributions on home page.

Version 3.9	2002−01−01

- Added Section 9.4, «Le standard Unicode».
- Added Section 9.5, «Rechercher des éléments».
- Added Section 9.6, «Accéder aux attributs d'un élément».
- Added Section 10.1, «Extraire les sources de données en entrée».
- Added Section 10.2, «Entrée, sortie et erreur standard».
- Added simple counter `for` loop examples (good usage and bad usage) in Section 6.3, «Itérations avec des boucles for». Thanks to Kevin for the idea.
- Fixed typo in Exemple 3.25, «Les fonctions keys, values et items» (two elements of `params.values()` were reversed).
- Fixed mistake in Section 4.3, «Utilisation de type, str, dir et autres fonction prédéfinies» with regards to the name of the `__builtin__` module. Thanks to Denis for the correction.

- Added additional example in Section 16.2, «Trouver le chemin» to show how to run unit tests in the current working directory, instead of the directory where `regression.py` is located.
- Modified explanation of how to derive a negative list index from a positive list index in Exemple 3.7, «Indices de liste négatifs». Thanks to Renauld for the suggestion.
- Updated links on home page for downloading latest version of Python.
- Added link on home page to Bruce Eckel's preliminary draft of *Thinking in Python* (http://www.mindview.net/Books/TIPython), a marvelous (and advanced) book on design patterns and how to implement them in Python.

Version 3.8	2001−11−18

- Added Section 16.2, «Trouver le chemin».
- Added Section 16.3, «Le filtrage de liste revisité».
- Added Section 16.4, «La mutation de liste revisitée».
- Added Section 16.5, «Programmation centrée sur les données».
- Expanded sample output in Section 16.1, «Plonger».
- Finished Section 9.3, «Analyser un document XML».

Version 3.7	2001−09−30

- Added Section 9.2, «Les paquetages».
- Added Section 9.3, «Analyser un document XML».
- Cleaned up introductory paragraph in Section 9.1, «Plonger». Thanks to Matt for this suggestion.
- Added Java tip in Section 5.2, «Importation de modules avec from module import». Thanks to Ori for this suggestion.
- Fixed mistake in Section 4.8, «Assembler les pièces» where I implied that you could not use `is None` to compare to a null value in Python. In fact, you can, and it's faster than `== None`. Thanks to Ori pointing this out.
- Clarified in Section 3.2, «Présentation des listes» where I said that `li = li + other` was equivalent to `li.extend(other)`. The result is the same, but `extend` is faster because it doesn't create a new list. Thanks to Denis pointing this out.
- Fixed mistake in Section 3.2, «Présentation des listes» where I said that `li += other` was equivalent to `li = li + other`. In fact, it's equivalent to `li.extend(other)`, since it doesn't create a new list. Thanks to Denis pointing this out.
- Fixed typographical laziness in Chapitre 2, *Votre premier programme Python*; when I was writing it, I had not yet standardized on putting string literals in single quotes within the text. They were set off by typography, but this is lost in some renditions of the book (like plain text), making it difficult to read. Thanks to Denis for this suggestion.
- Fixed mistake in Section 2.2, «Déclaration de fonctions» where I said that statically typed languages always use explicit variable + datatype declarations to enforce static typing. Most do, but there are some statically typed languages where the compiler figures out what type the variable is based on usage within the code. Thanks to Tony for pointing this out.
- Added link to Spanish translation (http://diveintopython.org/es/).

Version 3.6.4	2001−09−06

- Added code in `BaseHTMLProcessor` to handle non−HTML entity references, and added a note about it in Section 8.4, «Présentation de BaseHTMLProcessor.py».
- Modified Exemple 8.11, «Présentation de globals» to include `htmlentitydefs` in the output.

Version 3.6.3	2001−09−04

- Fixed typo in Section 9.1, «Plonger».
- Added link to Korean translation (http://diveintopython.org/kr/html/index.htm).

Version 3.6.2	2001−08−31

- Fixed typo in Section 13.6, «Tester la cohérence» (the last requirement was listed twice).

Version 3.6	2001−08−31

- Finished Chapitre 8, *Traitement du HTML* with Section 8.9, «Assembler les pièces» and Section 8.10, «Résumé».
- Added Section 15.4, «Postscriptum».
- Started Chapitre 9, *Traitement de données XML* with Section 9.1, «Plonger».
- Started Chapitre 16, *Programmation fonctionnelle* with Section 16.1, «Plonger».
- Fixed long−standing bug in colorizing script that improperly colorized the examples in Chapitre 8, *Traitement du HTML*.
- Added link to French translation (http://diveintopython.org/fr/toc.html). They did the right thing and translated the source XML, so they can re−use all my build scripts and make their work available in six different formats.
- Upgraded to version 1.43 of the DocBook XSL stylesheets.
- Upgraded to version 6.43 of processeur XSLT SAXON de Michael Kay.
- Massive stylesheet changes, moving away from a table−based layout and towards more appropriate use of cascading style sheets. Unfortunately, CSS has as many compatibility problems as anything else, so there are still some tables used in the header and footer. The resulting HTML version looks worse in Netscape 4, but better in modern browsers, including Netscape 6, Mozilla, Internet Explorer 5, Opera 5, Konqueror, and iCab. And it's still completely readable in Lynx. I love Lynx. It was my first web browser. You never forget your first.
- Moved to Ant (http://jakarta.apache.org/ant/) to have better control over the build process, which is especially important now that I'm juggling six output formats and two languages.
- Consolidated the available downloadable archives; previously, I had different files for each platform, because the .zip files that Python's `zipfile` module creates are non−standard and can't be opened by Aladdin Expander on Mac OS. But the .zip files that Ant creates are completely standard and cross−platform. Go Ant!
- Now hosting the complete XML source, XSL stylesheets, and associated scripts and libraries on SourceForge. There's also CVS access for the really adventurous.
- Re−licensed the example code under the new−and−improved GPL−compatible Python 2.1.1 license (http://www.python.org/2.1.1/license.html). Thanks, Guido; people really do care, and it really does matter.

Version 3.5	2001−06−26

- Added explanation of strong/weak/static/dynamic datatypes in Section 2.2, «Déclaration de fonctions».
- Added case−sensitivity example in Section 3.1, «Présentation des dictionnaires».
- Use `os.path.normcase` in Chapitre 5, *Les objets et l'orienté objet* to compensate for inferior operating systems whose files aren't case−sensitive.
- Fixed indentation problems in code samples in PDF version.

Version 3.4	2001−05−31

- Added Section 14.5, «roman.py, étape 5».
- Added Section 15.1, «Gestion des bogues».
- Added Section 15.2, «Gestion des changements de spécification».
- Added Section 15.3, «Refactorisation».
- Added Section 15.5, «Résumé».
- Fixed yet another stylesheet bug that was dropping nested `</span>` tags.

Version 3.3	2001−05−24

- Added Section 13.2, «Présentation de romantest.py».

- Added Section 13.3, «Présentation de romantest.py».
- Added Section 13.4, «Tester la réussite».
- Added Section 13.5, «Tester l échec».
- Added Section 13.6, «Tester la cohérence».
- Added Section 14.1, «roman.py, étape 1».
- Added Section 14.2, «roman.py, étape 2».
- Added Section 14.3, «roman.py, étape 3».
- Added Section 14.4, «roman.py, étape 4».
- Tweaked stylesheets in an endless quest for complete Netscape/Mozilla compatibility.

Version 3.2	2001−05−03

- Added Section 8.8, «Présentation de dialect.py».
- Added Section 7.2, «Exemple : adresses postales».
- Fixed bug in `handle_decl` method that would produce incorrect declarations (adding a space where it couldn't be).
- Fixed bug in CSS (introduced in 2.9) where body background color was missing.

Version 3.1	2001−04−18

- Added code in `BaseHTMLProcessor.py` to handle declarations, now that Python 2.1 supports them.
- Added note about nested scopes in Section 8.5, «locals et globals».
- Fixed obscure bug in Exemple 8.1, «BaseHTMLProcessor.py» where attribute values with character entities would not be properly escaped.
- Now recommending (but not requiring) Python 2.1, due to its support of declarations in `sgmllib.py`.
- Updated download links on the home page (http://diveintopython.org/) to point to Python 2.1, where available.
- Moved to versioned filenames, to help people who redistribute the book.

Version 3.0	2001−04−16

- Fixed minor bug in code listing in Chapitre 8, *Traitement du HTML*.
- Added link to Chinese translation on home page (http://diveintopython.org/).

Version 2.9	2001−04−13

- Added Section 8.5, «locals et globals».
- Added Section 8.6, «Formatage de chaînes à l aide d un dictionnaire».
- Tightened code in Chapitre 8, *Traitement du HTML*, specifically `ChefDialectizer`, to use fewer and simpler regular expressions.
- Fixed a stylesheet bug that was inserting blank pages between chapters in the PDF version.
- Fixed a script bug that was stripping the `DOCTYPE` from the home page (http://diveintopython.org/).
- Added link to Python Cookbook (http://www.activestate.com/ASPN/Python/Cookbook/), and added a few links to individual recipes in Annexe A, *Pour en savoir plus*.
- Switched to Google (http://www.google.com/services/free.html) for searching on `http://diveintopython.org/`.
- Upgraded to version 1.36 of the DocBook XSL stylesheets, which was much more difficult than it sounds. There may still be lingering bugs.

Version 2.8	2001−03−26

- Added Section 8.3, «Extraction de données de documents HTML».
- Added Section 8.4, «Présentation de BaseHTMLProcessor.py».
- Added Section 8.7, «Mettre les valeurs d attributs entre guillemets».

- Tightened up code in Chapitre 4, *Le pouvoir de l introspection*, using the built−in function `callable` instead of manually checking types.
- Moved Section 5.2, «Importation de modules avec from module import» from Chapitre 4, *Le pouvoir de l introspection* to Chapitre 5, *Les objets et l'orienté objet*.
- Fixed typo in code example in Section 5.1, «Plonger» (added colon).
- Added several additional downloadable example scripts.
- Added Windows Help output format.

Version 2.7	2001−03−16

- Added Section 8.2, «Présentation de sgmllib.py».
- Tightened up code in Chapitre 8, *Traitement du HTML*.
- Changed code in Chapitre 2, *Votre premier programme Python* to use `items` method instead of `keys`.
- Moved Section 3.4.2, «Assignation simultanée de plusieurs valeurs» section to Chapitre 2, *Votre premier programme Python*.
- Edited note about `join` string method, and provided a link to the new entry in *The Whole Python FAQ* (http://www.python.org/doc/FAQ.html) that explains why `join` is a string method (http://www.python.org/cgi−bin/faqw.py?query=4.96&querytype=simple&casefold=yes&req=search) instead of a list method.
- Rewrote Section 4.6, «Particularités de and et or» to emphasize the fundamental nature of `and` and `or` and de−emphasize the `and−or` trick.
- Reorganized language comparisons into `note`s.

Version 2.6	2001−02−28

- The PDF and Word versions now have colorized examples, an improved table of contents, and properly indented `tip`s and `note`s.
- The Word version is now in native Word format, compatible with Word 97.
- The PDF and text versions now have fewer problems with improperly converted special characters (like trademark symbols and curly quotes).
- Added link to download Word version for UNIX, in case some twisted soul wants to import it into StarOffice or something.
- Fixed several `note`s which were missing titles.
- Fixed stylesheets to work around bug in Internet Explorer 5 for Mac OS which caused colorized words in the examples to be displayed in the wrong font. (Hello?!? Microsoft? Which part of `<pre>` don't you understand?)
- Fixed archive corruption in Mac OS downloads.
- In first section of each chapter, added link to download examples. (My access logs show that people skim or skip the two pages where they could have downloaded them (the home page (http://diveintopython.org/) and preface), then scramble to find a download link once they actually start reading.)
- Tightened the home page (http://diveintopython.org/) and preface even more, in the hopes that someday someone will read them.
- Soon I hope to get back to actually writing this book instead of debugging it.

Version 2.5	2001−02−23

- Added Section 6.4, «Utilisation de sys.modules».
- Added Section 6.5, «Travailler avec des répertoires».
- Moved Exemple 6.17, «Division de noms de chemins» from Section 3.4.2, «Assignation simultanée de plusieurs valeurs» to Section 6.5, «Travailler avec des répertoires».
- Added Section 6.6, «Assembler les pièces».
- Added Section 5.10, «Résumé».
- Added Section 8.1, «Plonger».
- Fixed program listing in Exemple 6.10, «Parcourir un dictionnaire» which was missing a colon.

Version 2.4.1	2001−02−12

- Changed newsgroup links to use "news:" protocol, now that `deja.com` is defunct.
- Added file sizes to download links.

Version 2.4	2001−02−12

- Added "further reading" links in most sections, and collated them in Annexe A, *Pour en savoir plus*.
- Added URLs in parentheses next to external links in text version.

Version 2.3	2001−02−09

- Rewrote some of the code in Chapitre 5, *Les objets et l'orienté objet* to use class attributes and a better example of multi−variable assignment.
- Reorganized Chapitre 5, *Les objets et l'orienté objet* to put the class sections first.
- Added Section 5.8, «Attributs de classe».
- Added Section 6.1, «Traitement des exceptions».
- Added Section 6.2, «Les objets−fichier».
- Merged the "review" section in Chapitre 5, *Les objets et l'orienté objet* into Section 5.1, «Plonger».
- Colorized all program listings and examples.
- Fixed important error in Section 2.2, «Déclaration de fonctions»: functions that do not explicitly return a value return `None`, so you *can* assign the return value of such a function to a variable without raising an exception.
- Added minor clarifications to Section 2.3, «Documentation des fonctions», Section 2.4, «Tout est objet», and Section 3.4, «Définitions de variables».

Version 2.2	2001−02−02

- Edited Section 4.4, «Obtenir des références objet avec getattr».
- Added titles to `xref` tags, so they can have their cute little tooltips too.
- Changed the look of the revision history page.
- Fixed problem I introduced yesterday in my HTML post−processing script that was causing invalid HTML character references and breaking some browsers.
- Upgraded to version 1.29 of the DocBook XSL stylesheets.

Version 2.1	2001−02−01

- Rewrote the example code of Chapitre 4, *Le pouvoir de l introspection* to use `getattr` instead of `exec` and `eval`, and rewrote explanatory text to match.
- Added example of list operators in Section 3.2, «Présentation des listes».
- Added links to relevant sections in the summary lists at the end of each chapter (Section 3.8, «Résumé» and Section 4.9, «Résumé»).

Version 2.0	2001−01−31

- Split Section 5.6, «Méthodes de classe spéciales» into three sections, Section 5.5, «UserDict : une classe enveloppe», Section 5.6, «Méthodes de classe spéciales», and Section 5.7, «Méthodes spéciales avancées».
- Changed notes on garbage collection to point out that Python 2.0 and later can handle circular references without additional coding.
- Fixed UNIX downloads to include all relevant files.

Version 1.9	2001−01−15

- Removed introduction to Chapitre 2, *Votre premier programme Python*.
- Removed introduction to Chapitre 4, *Le pouvoir de l introspection*.

- Removed introduction to Chapitre 5, *Les objets et l'orienté objet*.
- Edited text ruthlessly. I tend to ramble.

Version 1.8	2001−01−12

- Added more examples to Section 3.4.2, «Assignation simultanée de plusieurs valeurs».
- Added Section 5.3, «Définition de classes».
- Added Section 5.4, «Instantiation de classes».
- Added Section 5.6, «Méthodes de classe spéciales».
- More minor stylesheet tweaks, including adding titles to `link` tags, which, if your browser is cool enough, will display a description of the link target in a cute little tooltip.

Version 1.71	2001−01−03

- Made several modifications to stylesheets to improve browser compatibility.

Version 1.7	2001−01−02

- Added introduction to Chapitre 2, *Votre premier programme Python*.
- Added introduction to Chapitre 4, *Le pouvoir de l introspection*.
- Added review section to Chapitre 5, *Les objets et l'orienté objet* [later removed]
- Added Section 5.9, «Fonctions privées».
- Added Section 6.3, «Itérations avec des boucles for».
- Added Section 3.4.2, «Assignation simultanée de plusieurs valeurs».
- Wrote scripts to convert book to new output formats: one single HTML file, PDF, Microsoft Word 97, and plain text.
- Registered the `diveintopython.org` domain and moved the book there, along with links to download the book in all available output formats for offline reading.
- Modified the XSL stylesheets to change the header and footer navigation that displays on each page. The top of each page is branded with the domain name and book version, followed by a breadcrumb trail to jump back to the chapter table of contents, the main table of contents, or the site home page.

Version 1.6	2000−12−11

- Added Section 4.8, «Assembler les pièces».
- Finished Chapitre 4, *Le pouvoir de l introspection* with Section 4.9, «Résumé».
- Started Chapitre 5, *Les objets et l'orienté objet* with Section 5.1, «Plonger».

Version 1.5	2000−11−22

- Added Section 4.6, «Particularités de and et or».
- Added Section 4.7, «Utiliser des fonctions lambda».
- Added appendix that lists section abstracts.
- Added appendix that lists tips.
- Added appendix that lists examples.
- Added appendix that lists revision history.
- Expanded example of mapping lists in Section 3.6, «Mutation de listes».
- Encapsulated several more common phrases into entities.
- Upgraded to version 1.25 of the DocBook XSL stylesheets.

Version 1.4	2000−11−14

- Added Section 4.5, «Filtrage de listes».
- Added `dir` documentation to Section 4.3, «Utilisation de type, str, dir et autres fonction prédéfinies».
- Added `in` example in Section 3.3, «Présentation des tuples».

- Added additional note about `if __name__` trick under MacPython.
- Switched to processeur XSLT SAXON de Michael Kay.
- Upgraded to version 1.24 of the DocBook XSL stylesheets.
- Added db−html processing instructions with explicit filenames of each chapter and section, to allow deep links to content even if I add or re−arrange sections later.
- Made several common phrases into entities for easier reuse.
- Changed several `literal` tags to `constant`.

Version 1.3	2000−11−09

- Added section on dynamic code execution.
- Added links to relevant section/example wherever I refer to previously covered concepts.
- Expanded introduction of chapter 2 to explain what the function actually does.
- Explicitly placed example code under the GNU General Public License and added appendix to display license. [Note 8/16/2001: code has been re−licensed under GPL−compatible Python license]
- Changed links to licenses to use `xref` tags, now that I know how to use them.

Version 1.2	2000−11−06

- Added first four sections of chapter 2.
- Tightened up preface even more, and added link to Mac OS version of Python.
- Filled out examples in "Mapping lists" and "Joining strings" to show logical progression.
- Added output in chapter 1 summary.

Version 1.1	2000−10−31

- Finished chapter 1 with sections on mapping and joining, and a chapter summary.
- Toned down the preface, added links to introductions for non−programmers.
- Fixed several typos.

Version 1.0	2000−10−30

- Initial publication

Annexe F. A propos de ce livre

Ce livre a été écrit en XML DocBook (http://www.oasis−open.org/docbook/) à l aide d Emacs (http://www.gnu.org/software/emacs/) et converti en HTML à l aide du processeur XSLT SAXON de Michael Kay (http://saxon.sourceforge.net/) avec une version modifiée des feuilles de style XSL de Norman Walsh (http://www.nwalsh.com/xsl/). De là, il a été converti au format PDF à l aide de HTMLDoc (http://www.easysw.com/htmldoc/) et en texte simple à l aide de w3m (http://ei5nazha.yz.yamagata−u.ac.jp/~aito/w3m/eng/). Les listings de programmes et les exemples ont été colorisés à l aide d une version actualisée de `pyfontify.py` de Just van Rossum, qui est inclue dans les scripts d exemple.

Si vous souhaitez en savoir plus sur l usage de DocBook pour la rédaction technique, vous pouvez Télécharger les sources XML (download/diveintopython−xml−5.4.zip) et les scripts de construction (download/diveintopython−common−5.4.zip), qui comprennent les feuilles de styles XSL modifiées utilisées pour créer l ensemble des différents formats du livre. Vous devriez également lire le livre de référence, *DocBook: The Definitive Guide* (http://www.docbook.org/). Si vous comptez utiliser DocBook sérieusement, je vous conseille de vous abonner aux listes de diffusion de DocBook (http://lists.oasis−open.org/archives/).

Annexe G. GNU Free Documentation License

Version 1.1, March 2000

> Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats

suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the

previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/ (http://www.gnu.org/copyleft/).

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Annexe H. Python license

H.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL−compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non−profit modeled after the Apache Software Foundation. See http://www.python.org/psf/ for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

H.B. Terms and conditions for accessing or otherwise using Python

H.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.1.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.1.1.
4. PSF is making Python 2.1.1 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.2. BeOpen Python open source license agreement version 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non−exclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.3. CNRI open source GPL−compatible license agreement

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty−free, world−wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995−2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement

may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non−separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.4. CWI permissions statement and disclaimer