Table of Contents

Table of Contents

Table of Contents

Plongez au coeur de Python

11 February 2004

Les évolutions de cet ouvrage (et de sa traduction française) sont disponibles sur le site http://diveintopython.org/. Si vous le lisez ailleurs, il est possible que vous ne disposiez pas de la dernière version.

Preface

Cet ouvrage n est pas pour les débutants, les faiblards ou Pour Les Nuls. Il attend de vous beaucoup de choses.

- Vous connaissez au moins un véritable langage orienté objet, tel que Java, C++ ou Delphi.
- Vous connaissez au moins un langage de script, tel que Perl, Visual Basic ou JavaScript.
- Vous avez déjà installé Python. Voir la page d accueil (http://diveintopython.org/) pour des liens de téléchargement de Python pour votre système d exploitation favori. Python 2.0 ou une version plus récente est requis, Python 2.2.1 est recommandé. Lorsqu il existe des différences notables entre 2.0 et 2.2.1, elles sont mentionnées clairement dans le texte.
- Vous avez téléchargé les programmes d exemple (http://diveintopython.org/download/diveintopython−examples−4.1.zip) utilisés dans ce livre.

Si vous venez à peine de débuter en programmation, cela ne veut pas dire que vous ne pouvez pas apprendre Python. Python est un langage facile à apprendre, mais vous devrez sans doute l apprendre ailleurs. Je vous recommande chaudement *Learning to Program* (http://www.freenetpages.co.uk/hp/alan.gauld/) et *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/), Python.org (http://www.python.org/) a des liens vers d autres introductions à la programmation en Python (http://www.python.org/doc/Intros.html) pour les non−programmeurs.

Plongeons.

Chapter 1. Faire connaissance de Python

1.1. Plonger

Voici un programme Python complet et fonctionnel.

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

Example 1.1. `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−4.1.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 Python sous Windows, vous pouvez exécuter un module avec 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 module depuis la ligne de commande : **`python odbchelper.py`**

Example 1.2. Sortie de `odbchelper.py`

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

1.2. Déclaration de fonctions

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

Example 1.3. Declaration de la fonction `buildConnectionString`

```
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 donnée 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 retourner cette 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 sons 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.
Addendum. Un lecteur erudit 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 (nous verrons plus loin dans ce chapitre comment le faire).

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).

1.3. Documentation des fonctions

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

Example 1.4. Définition d une `doc string` pour la fonction `buildConnectionString`

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

    Returns string."""
```

Les triples guillemets délimitent une chaîne de caractère multi−lignes. Tout les caractères se trouvant entre les guillemets de début et de fin font partie d une même chaîne, y compris les retours ` la ligne et les autres guillemets. Vous pouvez les utiliser partout bien que vous les verrez le plus souvent utilisé pour définir des `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 est 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`s pour fournir une documentation contextuelle, ainsi lorsque vous tapez le nom d une fonction, sa `doc string` apparaît dans une bulle d aide. Ce peut être incroyablement utile, mais cette utilité est liée à la qualité de votre `doc string`.

Pour en savoir plus

- *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) explique la manière d écrire de bonnes `doc string`.
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite des conventions d espacements dans les `doc string`s (http://www.python.org/doc/current/tut/node6.html#SECTION00675000000000000000).

1.4. Tout est objet

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

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

Example 1.5. Accède; 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. Une fois que vous importez un module, vous pouvez référencer chacune de ces fonctions, classes ou attributs publiques. 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, ce 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__`.

L `import` de Python est similaire au `require` de 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`*.

Avant d aller plus loin, je veux mentionner rapidement le chemin de recherche de bibliothès. 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 a 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)

Example 1.6. Chemin de recherche pour `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 en inclus dansPython 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 dePython en le joignant à `sys.path`, alors 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 plus loin dans ce chapitre.)

En Python, tout est objet et presque tout dispose d attributs et de méthodes. [1]Toutes les fonctions ont un attribut intégré `__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.

C est tellement important que je vais le répéter au cas où vous l auriez raté les fois précédentes : *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

- *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique exactectement ce que signifie l affirmation 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).

1.5. 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.

Example 1.7. 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, alternatives `if`, boucles `for`, etc.) sont définis par leur indentation. L indentation démarre le bloc et la désinendation le termine. Il n y a pas de 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, qui n est pas indentée, est en dehors de la fonction.

Après quelques protestations initiales et diverses analogies sarcastiques avec Fortran, vous ferez rapidement la paix avec cela et vous commencerez à en percevoir les avantages. Un des bénéfices majeurs est que les programmes Python se ressemblent puisque l indentation est une obligation du langage et non une affaire de style. Cela rends plus aisée la compréhension du code Python écrit par d autres.

Python utilise le retour à la ligne pour séparer les instructions et deux points ainsi que l indentation pour séparer les blocs de code. C++ et Java utilisent le point virgule pour séparer les instructions et les accolades pour séparer les blocs de code.

Pour en savoir plus

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

1.6. 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 facilement tester vos modules lorsque vous les écrivez.

Example 1.8. 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 assignation. Mais, au contraire de C, Python ne supporte pas les assignations simultanées afin d éviter qu une valeur soit accidentellement assignée alors que vous pensiez effectuer une simple comparaison.

Mais pourquoi est−ce que l instruction `if` en question est une astuce ? Les modules sont des objets et tous les modules disposent de l attribut intégré `__name__`. Le `__name__` d un module dépend de la façon dont vous l utilisez. Si vous `import`ez 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__`.

Example 1.9. Le `__name__` d un module importé

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

Sachant cela, vous pouvez concevoir une suite de tests pour votre module au sein même de ce dernier en le 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 de 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

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

1.7. Présentation des dictionnaires

Une petite digression est de rigueur car vous devez absolument connaître les dictionnaires, les tuples et les listes (oh mon dieu!). Si vous être un hacker Perl, vous pouvez probablement passer en vitesse les points concernant les dictionnaires et les listes mais vous devriez quand même faire attention aux tuples.

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.

Example 1.10. 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 entourés 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.

Example 1.11. 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 va ennuiera le jour où vous penserez que vous ajoutez de nouvelles valeurs alors que vous étiez seulement en train de modifier encore et encore 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). Il y a des façons de le faire, seulement elle ne sont pas intégrées dans le dictionnaire.

Example 1.12. 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.

Example 1.13. 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.

Example 1.14. Les chaînes 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 à une clé de dictionnaire existante remplace simplement l ancienne valeur par une nouvelle.

❷ Ceci n est pas l assignation d une valeur à une clé existante car les chaînes en Python sont sensibles à la casse et donc `'key'` n est pas la meme chose que `'Key'`. Cela crée une nouvelle paire clé/valeur dans le dictionnaire, les deux clé peuvent être similaires pour vous mais pour Python elles sont complètement différente.

Pour en savoir plus

- *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).
- 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).
- 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).
- *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).

1.8. 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 `Vector`, qui peut contenir n importe quels objets et qui croît dynamiquement au fur et à mesure que de nouveaux éléments y sont ajoutés.

Example 1.15. 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 utilisé 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.

Example 1.16. 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 les indices négatifs vous prêtent à confusion, voyez−les comme suit : `li[n] == li[n - len(li)]`. Donc dans cette liste, `li[-3] == li[5 - 3] == li[2]`.

Example 1.17. 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.

Example 1.18. 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.

Example 1.19. 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 ce cas, la liste est composée de deux éléments.

Example 1.20. 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
0
```

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 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 `1` si la valeur a été trouvée et `0` 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 2.2.1 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.

Example 1.21. 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.

Example 1.22. 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 3.)

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

- *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) enseigne les listes et explique le sujet important du passage de listes comme arguments de fonction (http://www.ibiblio.org/obp/thinkCSpy/chap08.htm).
- *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).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions fréquentes à propos des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/534) et fourni de nombreuxexemples de code utilisant des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/540).
- *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).

1.9. Présentation des tuples

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

Example 1.23. 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.

Example 1.24. 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
1
```

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 élémént 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.
- 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. [2]
- 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 intégrée `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

- *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).
- 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).
- *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).

1.10. 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, `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.

Example 1.25. 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.

Lorsq 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.

Example 1.26. 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.

Pour en savoir plus

- *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).

1.11. Assignation simultanée de plusieurs valeurs

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

Example 1.27. 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 intégrée `range` avec l assignation multiples de variables pour assigner rapidement des valeurs consécutives.

Example 1.28. 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 intégrée `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 marrant qui affiche des calendrier 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 3.

Pour en savoir plus

- *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/chap10.htm).

1.12. 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 chaîne en Python utilise la même syntaxe que la fonction C `sprintf`.

Example 1.29. 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 effectuer une simple concaténation de chaînes, et vous auriez raison si le formatage de chaîne se bornait à la concaténation. Il n y est pas question uniquement de formatage mais également de conversion de types.

Example 1.30. 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 add type "int" to string
```

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.

Pour en savoir plus

- *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).

1.13. 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.

Example 1.31. 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. Il n y a pas de *race condition* (collision) ou d autres bizarreries à redouter. Python assemble la nouvelle liste en mémoire et assigne le résultat à la variable une fois la transformation terminée.

Example 1.32. *List comprehensions* dans `buildConnectionString`

```
["%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.

Example 1.33. `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`.

Example 1.34. *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

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d une autre manière de transformer des listes avec la fonction intégrée `map` (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
- *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).

1.14. Jointure de listes et découpage de chaînes

Vous avez une liste de paires clé−valeur sous la forme `clé=valeur` et vous voulez les assembler au sein d une même chaîne. Pour joindre une liste de chaînes en une seule, vous pouvez utiliser la méthode `join` d un objet chaîne.

Example 1.35. Jointure de liste dans `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 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.

Example 1.36. 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 `help` et imprimée par le bloc appelant, ce qui vous donne la sortie qui vous a tant émerveillé depuis que vous avez débuté la lecture de ce chapitre.

Note historique. 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`.

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`.

Example 1.37. 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` effectue l opération inverse de `join` en éclatant une chaîne en une liste. Notez que le séparateur (`;`) est complètement retiré; il n apparaît dans aucun des éléments de la liste retournée.

❷ `split` prends un second argument optionnel qui est le nombre de découpages souhaités. ("Ooooh, des arguments optionnels..." Vous apprendrez comment faire cela dans vos propres fonctions au cours du chapitre suivant.)

`une_chaîne.split(séparateur, 1)` est une technique pratique lorsque voulez rechercher une sous−chaîne dans une chaîne et traiter tous ce qui se situe avant (soit le premier élément de la liste) et après (le dernier élément de la liste) l occurrence de cette sous−chaîne.

Pour en savoir plus

- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions fréquentes 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).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes de chaînes (http://www.python.org/doc/current/lib/string−methods.html).
- *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.

1.15. Résumé

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

Example 1.38. `odbchelper.py`

```
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)
```

Example 1.39. 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 qui puissent être exécutés comme programmes autonomes, au moins pour les tester
- Importer des modules et appeler leurs fonctions
- Déclarer des fonctions et utiliser des `doc string`s, des variables locales avec 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

---

[1] 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 dans la suite de ce chapitre) et tous les objets ne sont pas dérivables (voir chapitre 3). Mais tout est objet dans le sens où il peut se voir assigné à une variable ou passé comme argument à une fonction (voir chapitre 2).

[2] 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.

Chapter 2. Le pouvoir de l introspection

2.1. Plonger

Ce chapitre traite d une des forces de Python : l introspection. Comme vous le savez, tout est objet dans Python, l introspection consiste en du code regardant les autres modules et fonctions en mémoires comme des objets, obtenant des informations de leur part et les manipulant. 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 dons nous ne connaissons même pas le nom à l avance.

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 Chapter 1, *Faire connaissance de 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.

Example 2.1. `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−4.1.zip) du livre.

```
def help(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 help.__doc__
```

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

❷ La fonction `help` 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 `help`.

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

La fonction `help` 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`s.

Example 2.2. Exemple d utilisation de `apihelper.py`

```
>>> from apihelper import help
```

```
>>> li = []
>>> help(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`s 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.

Example 2.3. Usage avancé de `apihelper.py`

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

    Returns string.
```

2.2. 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.

Example 2.4. `help`, une fonction avec deux arguments optionnels

```
def help(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. If `help` est appelé avec un seul argument, `spacing` prend pour valeur `10` et `collapse` la valeur `1`. Si `help` 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.

Example 2.5. Appels de `help` autorisés

```
help(odbchelper)                ❶
help(odbchelper, 12)            ❷
help(odbchelper, collapse=0)    ❸
```

```
help(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

- *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.

2.3. `type`, `str`, `dir` et autres fonctions intégrées

Python a un petit ensemble de fonctions intégrées 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 (hum, hum, Visual Basic).

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.

Example 2.6. 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
1
```

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 `help`, comme nous le verrons bientôt.

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

Example 2.7. Présentation de `str`

```
>>> str(1)          ❶
'1'
>>> 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 `help`, comme nous le verrons bientôt.

Au coeur de notre fonction `help`, 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.

Example 2.8. Présentation de `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 chaîne, pas les méthodes 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 intégrés, 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 étudié dans Chapter 1, *Faire connaissance de Python*.

Enfin, la fonction `callable` prend n importe quel objet et retourne `1` si l objet peut être appelé, sinon `0`. Les objets appelables sont les fonctions, les méthodes de classes ainsi que les classes elles−mêmes. (Nous verrons les classes au chapitre 3.)

Example 2.9. Présentation de `callable`

```
>>> import string
>>> string.punctuation                    ❶
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                           ❷
<function join at 00C55A7C>
>>> callable(string.punctuation)          ❸
0
>>> callable(string.join)                 ❹
1
>>> 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 appelable, 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.

`type`, `str`, `dir`, et toutes les autres fonctions intégrées de Python sont regroupées 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 intégrées (*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 intégrés de manière groupée en obtenant des informations sur le module `__builtin__`. Et devinez quoi, nous avons une fonction pour ça, elle s appelle `help`. Essayez vous−même et parcourez la liste maintenant, nous examinerons certaines des fonctions les plus importantes plus tard. (Certaines des classes d erreur intégrées, comme `AttributeError`, devraient avoir l air familier.)

Example 2.10. Attributs et fonctions intégrés

```
>>> from apihelper import help
>>> import __builtin__
>>> help(__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, que Dieu vous aide, MSDN) pour vous rappeler l usage de ces modules, Python est en grande partie auto−documenté.

Pour en savoir plus

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

2.4. 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`.

Example 2.11. 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 intégrée extrèmement utile qui retourne n importe quel attribut de n importe quel objet. Ici, k 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, voyez 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 ke 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.

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

Example 2.12. `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
1
>>> callable(getattr(object, method))               ❺
1
```

❶ Ceci retourne une référence à la fonction `buildConnectionString` du module `odbchelper`, que nous avons étudié dans Chapter 1, *Faire connaissance de 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 `help`. `object` est passé en argument à la fonction, `method` est une chaîne, le nom de la méthode ou 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).

2.5. 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*. Cela peut être associé à un mécanisme de filtrage par lequel certains éléments sont appliqués alors que d autres sont totalement ignorés.

Example 2.13. Syntaxe du filtrage de listes

```
[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 expresion 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.

Example 2.14. 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 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 quo 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.

Example 2.15. Filtrage d une liste dans `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, intégrées (comme la méthode `pop` d une liste) ou définie 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

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

2.6. 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.

Example 2.16. 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.[3] 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'`.

Example 2.17. 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.

Example 2.18. 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 !)

Example 2.19. 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` trick, 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`.

Example 2.20. 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

- 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).

2.7. 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 ou une fonction est nécéssaire.

Example 2.21. 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 est 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 expresison unique. Les fonctions `lambda` ne peuvent pas contenir de commandes, et elles ne peuvent contenir plus d une expression. N essayez pas de faire trop rentrer 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 nécéssité, 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 ligne.

Example 2.22. 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 1 ou 2 arguments, sans arguments elle utilise les espaces comme séparateur.

Example 2.23. `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 `help` pour replier les `doc string`s sur une seule ligne.

Mais que fait donc exactement cette fonction `help` avec ces fonctions `lambda`, ces `split`s et ces astuces `and-or` ?

Example 2.24. Assignation d une fonction à une variable

```
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

- 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).
- *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). (PEP 227 (http://python.sourceforge.net/peps/pep-0227.html) explique comment cela va changer dans les futures versions de Python.)
- *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).

2.8. 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.

Example 2.25. `apihelper.py` : le plat de résistance

```
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 et vers le début. Le

```
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`.

Example 2.26. 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 `help`, `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 intégrée 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`.

Example 2.27. Pourquoi utiliser `str` sur une `doc string` ?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__      ❶
>>> foo.__doc__ == None  ❷
1
>>> 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 NULL` 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.

Example 2.28. 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 `help` l utilise pour afficher sur deux colonnes et aligner les `doc string`s 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..

Example 2.29. 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 travailler toujours avec des listes.

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

Example 2.30. Le plat de résistance d'`apihelper.py`, revisité

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

2.9. Résumé

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

Example 2.31. `apihelper.py`

```
def help(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 help.__doc__
```

Example 2.32. Sortie de `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(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.

---

[3] 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 3.

Chapter 3. Un framework orienté objet

3.1. Plonger

Ce chapitre, et la plupart ceux qui le suivent, a trait à la programmation Python orientée objet. Vous vous rappelez que j ai dit que vous deviez connaître un langage orienté objet pour lire ce livre ? Et bien, je ne plaisantais pas.

Voici un programme Python complet et fonctionnel. Lisez les `doc string`s 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, c est à vous l expliquer que sert la suite du chapitre.

Example 3.1. `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−4.1.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.

Example 3.2. Sortie de `fileinfo.py`

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
```

3.2. Importation de modules avec `from module import`

Python fournit deux manières d importer les modules. Les deux sont utiles, et vous devez savoir quand utiliser laquelle. Vous avez déjà vu la première, `import module`, au chapitre 1. La deuxième manière accomplit la même action mais a des différences subtiles mais importante dans son fonctionnement.

Example 3.3. 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 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` en Perl; `import module` en Python est comme `require module` en Perl.

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

Example 3.4. `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 de 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 import 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 devez−vous 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.

Pour en savoir plus

- 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).
- *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).

3.3. Définition de classes

Python est entièrement orienté objet : vous pouvez définir vos propres classes, hériter de vos classes ou des classes intégrées 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 commencer 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.

Example 3.5. La classe Python la plus simple

```
class foo:  ❶
    pass    ❷ ❸
```

❶ Le nom de cette classe est foo et elle n hérite d aucune autre classe.

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__`.

Example 3.6. 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 classeclass `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.

Nous ne l étudierons pas en détail dans ce livre, mais 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, séparée par des virgules.

Example 3.7. 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 crée. 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 pauvent 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 noms. Cependant, veuillez ne pas l appeler autre chose que `self`, c est une très forte convention.

Example 3.8. 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.

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

- *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/chap11.htm).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) a une aperçu en profondeur des classes, des espaces de noms et de l héritage (http://www.python.org/doc/current/tut/node11.html).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond à des questions fréquentes à propos des classes (http://www.faqts.com/knowledge−base/index.phtml/fid/242).

3.4. 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 seral l objet nouvellement créé.

Example 3.9. 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
'base class for file info'
>>> 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 intégré, `__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 de 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.

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.

Example 3.10. 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 assigné 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 foirs la nouvelle instance de `FileInfo` class 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

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

3.5. `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 Python sous Windows, vous pouvez ouvrir rapidement n importe quel module dans votre chemin de bibliothèques avec File−>Locate... (**Ctrl−L**).

Note historique. Dans les versions de Python antérieures à la 2.2, vous ne pouviez pas directement dériver les types de données intégrés 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 intégrés : `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` works, au cas où vous aurriez besoin d implémenter ce genre d objet enveloppe, ou au cas où vous auriez à travailler avec une version de Python antèrieure à la 2.2.

Example 3.11. 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 initalisés 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 De plus, 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 suivant, il y a juste une instruction unique sdur 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. (C est 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 normal methodsancêtre, 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++ : 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).

Example 3.12. 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 dictionnaore, 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 nécessiterais 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 qu il soit très intéressant et vaille la peine que vous y jetiez un coup d Sil). 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 difficultés, les appels sont redirigés vers les méthodes de `self.data`.

Pour en savoir plus

- *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) (http://www.python.org/doc/current/lib/module−copy.html).

3.6. 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.

Example 3.13. 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 attendrais. Mais il y a un chaînon manquant : en coulisse, Python a converti 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.

Example 3.14. 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 appeleriez 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 est 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.

Example 3.15. 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`*.

Example 3.16. Définition du nom d un `MP3FileInfo`

```
>>> 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.

3.7. 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.

Example 3.17. 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 intégrée 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 **ENTER**, 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 intégrés 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 intégrée 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 the `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 intégré. Et c est vrai que tout serait plus simple (et la classe `UserDict` serait inutile) si on pouvait hériter d un type de données intégré 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 permettent 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 permettent 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

- *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).

3.8. 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.

Example 3.18. 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 est 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.[4]Vous pouvez également les modifier.

Example 3.19. 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 intégré 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.

3.9. Fonctions privées

Comme la plupart des langages, Python possède le concept de fonctions privées, qui ne peuvent être appelées de l extérieur de leur module, de méthodes de classe privées, qui ne peuvent être appelées de l extérieur de leur classe et d 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.

`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.

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.

En Python, toutes les méthodes spéciales (comme `__setitem__`) et les attributs intégrés (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 apporterais que de la confusion pour vous et les autres.

Python n a pas de notion de méthodes de classe protégées (accessibles uniquement par leur propre classe et les descendants de celle−ci). Les méthodes de classes sont ou privées (accessibles seulement par leur propre classe) ou publiques (accessible de partout).

Example 3.20. 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.[5]

Pour en savoir plus

- *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).

3.10. 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.
Si vous connaissez déjà tout sur les exceptions, vous pouvez survoler cette section. Si vous avez du jusque ici programmer dans des langages ne gérant pas les exceptions, ou si vous avez utilisé un langage digne de ce nom mais sans utiliser les exceptions, cette section est très importante.

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.[6]

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`.

Example 3.21. 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 intégrée `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.

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 library 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 intégrée `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.

Example 3.22. Support de fonctionnalités propre à une plateforme

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.

```
# 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

- *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).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les exceptions intégrées (http://www.python.org/doc/current/lib/module−exceptions.html).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module getpass (http://www.python.org/doc/current/lib/module−getpass.html).
- *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.
- *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).

3.11. Les objets fichier

Python a une fonction intégrée, `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.

Example 3.23. 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.

Example 3.24. 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                      2000htt
>>> 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.

Example 3.25. Fermeture d un fichier

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed ❶
0
>>> f.close() ❷
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed
1
>>> f.seek(0) ❸
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() ❹
```

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 `0`). Les fichiers ouverts consomment des ressources système et, en fonction du mode d ouverture, les autres programmes peuvent ne pas y avoir accès. Il est important de fermer les fichiers dès que vous avez fini de les utiliser.

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

❸ 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.

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

Example 3.26. 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 risqué 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 Voici 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`.

Pour en savoir plus

- *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 éfficience et de la performance de différentes manières de lire un fichier (http://www.effbot.org/guides/readline−performance.htm).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions fréquentes à propose des fichiers (http://www.faqts.com/knowledge−base/index.phtml/fid/552).
- *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).

3.12. 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 types de données liste aussi puissant que celui de Python, vous êtes donc amenés 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.

Example 3.27. Présentation de la boucle `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 succesivement 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*.

Example 3.28. 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 Faire un compteur "normal" (selon les critères de Visual Basic) pour la boucle `for` est également simple. Comme nous l avons vu dans Example 1.28, 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.

Example 3.29. Parcourir un dictionnaire

```
>>> 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.

Example 3.30. 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.

3.13. Complément sur les modules

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

Example 3.31. 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.

Example 3.32. 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`.

Example 3.33. 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__` intégré, 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.

Example 3.34. `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 epressions par défait 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

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite de quand et comment exactement les arguments par défaut sont évalués (http://www.python.org/doc/current/tut/node6.html#SECTION00671000000000000000).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module `sys` (http://www.python.org/doc/current/lib/module−sys.html).

3.14. Le module os

Le module `os` a de nombreuses fonctions utiles pour manipuler les fichiers et les processus. `os.path` a des fonctions pour manipuler les chemins de fichiers et de répertoires.

Example 3.35. 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 la 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, cela 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.

Example 3.36. 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..

Example 3.37. 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.

Example 3.38. 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)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
```

Ces deux lignes de code combinent tout ce que nous avons appris jusque ici à propos du module `os`, et même plus.

1. `os.listdir(directory)` retourne une liste de tous les fichiers et répertoires de `directory`.
2. 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.
3. 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.
4. 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`).
5. 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.

Pour en savoir plus

- 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).
- *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).

3.15. 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.

Example 3.39. `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 intéresantes (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 bient, 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.

3.16. Résumé

Le programme `fileinfo.py` devrait maintenant être parfaitement clair.

Example 3.40. `fileinfo.py`

```
"""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 :

- 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
- 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 indépendament de la plateforme
- Instancier des classes de type inconnu dynamiquement en traitant les classes comme des objets

---

[4] Il n y a pas de constantes en Python. Tout est modifiable en faisiant un effort. Cela est en accord acec 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] 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.

[6] Ou, comme des *marketroids* le diraient, votre programme ferait une opération illégale. Quelque chose du genre.

Chapter 4. Traitement du HTML

4.1. 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.

Example 4.1. `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−4.1.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)
```

Example 4.2. `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")
```

Example 4.3. Sortie de `dialect.py`

Ce script effectue la traduction de Section 1.8, 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.

4.2. 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 hierarchique 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 l illustrer. Vous pouvez exécuter `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 Python 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.

Example 4.4. Exemple de test de `sgmllib.py`

Voici un extrait de la table des matières de la version HTML de ce livre, `toc.html`.

```
<h1>
  <a name='c40a'></a>
  Dive Into Python
</h1>
<p class='pubdate'>
  28 Feb 2001
</p>
<p class='copyright'>
  Copyright copy 2000, 2001 by
  <a href='mailto:f8dy@diveintopython.org' title='send e-mail to the author'>
    Mark Pilgrim
  </a>
</p>
<p>
  <a name='c40ab2b4'></a>
  <b></b>
</p>
<p>
  This book lives at
```

```
  <a href='http://diveintopython.org/'>
    http://diveintopython.org/
  </a>
  .
  If you re reading it somewhere else, you may not have the latest version.
</p>
```

En l utilisant avec la suite de tests de `sgmllib.py`, on obtient la sortie suivante :

```
start tag: <h1>
start tag: <a name="c40a" >
end tag: </a>
data: 'Dive Into Python'
end tag: </h1>
start tag: <p class="pubdate" >
data: '28 Feb 2001'
end tag: </p>
start tag: <p class="copyright" >
data: 'Copyright '
*** unknown entity ref: &copy;
data: ' 2000, 2001 by '
start tag: <a href="mailto:f8dy@diveintopython.org" title="send e-mail to the author" >
data: 'Mark Pilgrim'
end tag: </a>
end tag: </p>
start tag: <p>
start tag: <a name="c40ab2b4" >
end tag: </a>
start tag: <b>
end tag: </b>
end tag: </p>
start tag: <p>
data: 'This book lives at '
start tag: <a href="http://diveintopython.org/" >
data: 'http://diveintopython.org/'
end tag: </a>
data: ".\012If you re reading it somewhere else, you may not have the lates"
data: 't version.\012'
end tag: </p>
```

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.

4.3. 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 obtenit 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.

Example 4.5. 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:f8dy@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>

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

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.

Example 4.6. 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-4.1.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`.

Example 4.7. 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.html
#download
toc.html
history.html
download/dip_pdf.zip
download/dip_pdf.tgz
download/dip_pdf.hqx
download/diveintopython.pdf
download/diveintopython.zip
download/diveintopython.tgz
download/diveintopython.hqx

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

❶ Appelez la méthode `feed`, définie dans `SGMLParser`, pour charger le code HTML dans l analyseur. [7] 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.

4.4. 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 utilse, 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 envoit 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`.

Example 4.8. 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.[8]

❷ Comme `BaseHTMLProcessor` ne définit aucune méthode pour des balises spécifiques (comme la méthode `start_a` de `URLLister`), `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 dans la prochaîne 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 empécher la bonne exêcution 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.

Example 4.9. `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 nécéssaire 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

- W3C (http://www.w3.org/) traite des références de caractères et d entités (http://www.w3.org/TR/REC−html40/charset.html#entities).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) confirme vos soupçons selon lesquels le module `htmlentitydefs` (http://www.python.org/doc/current/lib/module−htmlentitydefs.html) est exactement ce que son nom laisse deviner.

4.5. `locals` et `globals`

Python a deux fonctions intégrées permettant d accéder aux variables locales et globales sous forme de dictionnaire : `locals` et `globals`.

D abord, un mot des 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 intégré, 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 intégré − 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 un modification légère mais importante qui affecte l ordre de recherche dans les espaces de noms : les portées imbriquées. Dans les versions précédente de Python, lorsque vous référenciez une variable dans une fonction imbriquée ou une fonction `lambda`, Python recherche 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 les deux 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
```

Comme beaucoup de chose en Python, les espaces de noms sont accessibles directement durant l execution. L espace de noms local est accessible par la fonction intégrée `locals` et l espace de noms global (du module) est accessible par la fonction intégrée `globals`.

Example 4.10. 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 défini 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.

Example 4.11. Présentation de `globals`

Ajoutez le bloc suivant à `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 assignment multiple. Le seul élément nouveau ici est la fonction `globals`.

Maintenant, exécuter le programme de la ligne de commande nous donne la sortie suivante :

```
c:\docbook\dip\py>python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                                    ❶
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python21\lib\htmlentitydefs.py'> ❷
BaseHTMLProcessor = __main__.BaseHTMLProcessor                     ❸
__name__ = __main__                                                ❹
[...snip...]
```

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 intégré `__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.

Example 4.12. `locals` est en lecture seule, contrairement à `globals`

```
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 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`.

4.6. Formatage de chaînes à l aide d un dictionnaire

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.

Example 4.13. 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 occurence sera remplacée avec la même valeur.

Pourquoi voudriez−vous utiliser le formatage à l aide d un dictionnaire ? C 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`.

Example 4.14. 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 intégrée `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-->'`.

```
    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 `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 :

1. Supposez que `attrs` vaut `[('href', 'index.html'), ('title', 'Go to home page')]`.
2. Durant la première étape de la *list comprehension*, la clé (`key`) sera `'href'` et la valeur (`value`) `'index.html'`.
3. 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*.
4. Durant la seconde étape, `key` sera `'title'` et `value` `'Go to home page'`.

5. Le formatage de chaîne donnera `' title="Go to home page"'`.
6. 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 dûe à l appel de `locals`, puisque `locals` effectue une copie de l espace de noms local.

4.7. 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 attibuts sans guillemets et je veux les mettre entre guillemets. Comment faire ?"[9] (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.

Example 4.15. 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 utilison 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 optenons 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.

4.8. 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`.

Example 4.16. 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.

Example 4.17. `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`.

Example 4.18. 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 nécéssaires, la seule vrai solution est d utiliser les expressions régulières.

4.9. Introduction aux expressions régulières

Les expressions régulières sont un moyen puissant (et relativement 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.

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 et sont dans la casse correspondante. Les méthodes `replace` et `split` ont les mêmes restrictions. Utilisez les quand vous pouvez (elles sont rapides et faciles à comprendre) mais, pour tous les cas un peu plus complexes, vous devrez vous tourner vers les expressions régulières.

Example 4.19. Reconnaître la fin d une chaîne

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 exportée d un ancien système avant de les importer dans un nouveau système (vous voyez, je n invente rien, c est réellement utile).

```
>>> 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 abbrégé en `'RD.'`. Au premier abord, je pensait 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`.

Example 4.20. Reconnaître des mots entiers

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)      ❶
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)   ❷
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)   ❸
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)   ❹
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s)  ❺
```

```
'100 BROAD RD. APT 3'
```

❶ 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.

❷ 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, cela 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).

❸ 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).

❹ **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.

❺ 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.

Ce n est qu une tout petite partie de ce que les expressions régulières peuvent faire. Elles sont extrèmement puissante, des livres entiers leurs sont consacrés. Elles ne sont pas la solution idéale à n importe quel problème. Vous devriez en apprendre suffisament sur elles pour savoir quand elles sont appropriées et quand elles risquent de causer plus de problèmes que d en résoudre.

> Certaines personnes, lorsqu elles sont confrontées à un problème, se disent "Je sais, je vais utiliser les expressions régulières." Et là, elles ont maintenant deux problèmes.
>
> −−Jamie Zawinski, dans comp.lang.emacs

Pour en savoir plus

- La Regular Expression HOWTO (http://py−howto.sourceforge.net/regex/regex.html) présente les expressions régulières et explique comment les utiser 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).

4.10. Assembler les pièces

Il est temps d utiliser tout ce que nous avons appris. J espère que vous avez été attentifs.

Example 4.21. 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 source de l URL passée en paramètre.

Example 4.22. 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 designée par la châine (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.

Example 4.23. 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 fonciton `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 coyiez que je plaisantait 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/). Je ne sais pas du tout si il est implémenté en Python, mais la page d accueil de ma société est à hurler de rire en *Pig Latin*. Malheureusement, le code source n a pas l air d être disponible.

4.11. 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

- Utiliser des expressions régulières. Elles sont un outil important pour tous les programmeurs et vont remplir un plus grand rôle dans de prochains chapitres.

---

[7] 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.

[8] 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 $O(n^2)$.

[9] 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, 10/14/1998) Mais des questions sur le traitement du HTML apparaissent sous une forme ou l autre à peu près une fois par mois, et parmis ces questions celle−ci est fréquente.

Chapter 5. XML Processing

5.1. Diving in

This chapter is about XML processing in Python. It would be helpful if you already knew what an XML document looks like, that it s made up of structured tags to form a hierarchy of elements, and so on. If this doesn t make sense to you, go read an XML tutorial (http://directory.google.com/Top/Computers/Data_Formats/Markup_Languages/XML/Resources/FAQs,_Help,_and_Tutorials/ first, then come back.

Being a philosophy major is not required, although if you have ever had the misfortune of being subjected to the writings of Immanuel Kant, you will appreciate the example program a lot more than if you majored in something useful, like computer science.

There are two basic ways to work with XML. One is called SAX ("Simple API for XML"), and it works by reading the XML a little bit at a time and calling a method for each element it finds. (If you read Chapter 4, *Traitement du HTML*, this should sound familiar, because that s how the `sgmllib` module works.) The other is called DOM ("Document Object Model"), and it works by reading in the entire XML document at once and creating an internal representation of it using native Python classes linked in a tree structure. Python has standard modules for both kinds of parsing, but this chapter will only deal with using the DOM.

The following is a complete Python program which generates pseudo−random output based on a context−free grammar defined in an XML format. Don t worry yet if you don t understand what that means; we ll examine both the program s input and its output in more depth throughout the chapter.

Example 5.1. `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−4.1.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:])
```

Example 5.2. `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
    return StringIO.StringIO(str(source))
```

Run the program `kgp.py` by itself, and it will parse the default XML−based grammar, in `kant.xml`, and print several paragraphs worth of philosophy in the style of Immanuel Kant.

Example 5.3. Sample output of `kgp.py`

```
[f8dy@oliver 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...]
```

This is, of course, complete gibberish. Well, not complete gibberish. It is syntactically and grammatically correct (although very verbose −− Kant wasn t what you would call a get−to−the−point kind of guy). Some of it may actually be true (or at least the sort of thing that Kant would have agreed with), some of it is blatantly false, and most of it is simply incoherent. But all of it is in the style of Immanuel Kant.

Let me repeat that this is much, much funnier if you are now or have ever been a philosophy major.

The interesting thing about this program is that there is nothing Kant−specific about it. All the content in the previous example was derived from the grammar file, `kant.xml`. If we tell the program to use a different grammar file (which we can specify on the command line), the output will be completely different.

Example 5.4. Simpler output from `kgp.py`

```
[f8dy@oliver kgp]$ python kgp.py −g binary.xml
00101001
[f8dy@oliver kgp]$ python kgp.py −g binary.xml
10110100
```

We will take a closer look at the structure of the grammar file later in this chapter. For now, all you have to know is that the grammar file defines the structure of the output, and the `kgp.py` program reads through the grammar and makes random decisions about which words to plug in where.

5.2. Packages

Actually parsing an XML document is very simple: one line of code. However, before we get to that line of code, we need to take a short detour to talk about packages.

Example 5.5. Loading an XML document (a sneak peek)

```
>>> from xml.dom import minidom ❶
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

❶ This is a syntax we haven t seen before. It looks almost like the `from` *`module`* `import` we know and love, but the `"."` gives it away as something above and beyond a simple import. In fact, `xml` is what is known as a package, `dom` is a nested package within `xml`, and `minidom` is a module within `xml.dom`.

That sounds complicated, but it s really not. Looking at the actual implementation may help. Packages are little more than directories of modules; nested packages are subdirectories. The modules within a package (or a nested package) are still just `.py` files, like always, except that they re in a subdirectory instead of the main `lib/` directory of your Python installation.

Example 5.6. File layout of a package

```
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)
```

So when we say `from xml.dom import minidom`, Python figures out that that means "look in the `xml` directory for a `dom` directory, and look in *that* for the `minidom` module, and import it as `minidom`". But Python is even smarter than that; not only can you import entire modules contained within a package, you can selectively import

specific classes or functions from a module contained within a package. You can also import the package itself as a module. The syntax is all the same; Python figures out what you mean based on the file layout of the package, and automatically does the right thing.

Example 5.7. Packages are modules, too

```
>>> 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 Here we re importing a module (`minidom`) from a nested package (`xml.dom`). The result is that `minidom` is imported into our namespace, and in order to reference classes within the `minidom` module (like `Element`), we have to preface them with the module name.

❷ Here we are importing a class (`Element`) from a module (`minidom`) from a nested package (`xml.dom`). The result is that `Element` is imported directly into our namespace. Note that this does not interfere with the previous import; the `Element` class can now be referenced in two ways (but it s all still the same class).

❸ Here we are importing the `dom` package (a nested package of `xml`) as a module in and of itself. Any level of a package can be treated as a module, as we ll see in a moment. It can even have its own attributes and methods, just the modules we ve seen before.

❹ Here we are importing the root level `xml` package as a module.

So how can a package (which is just a directory on disk) be imported and treated as a module (which is always a file on disk)? The answer is the magical `__init__.py` file. You see, packages are not simply directories; they are directories with a specific file, `__init__.py`, inside. This file defines the attributes and methods of the package. For instance, `xml.dom` contains a `Node` class, which is defined in `xml/dom/__init__.py`. When you import a package as a module (like `dom` from `xml`), you re really importing its `__init__.py` file.

A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn t have to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn t exist, the directory is just a directory, not a package, and it can t be imported or contain modules or nested packages.

So why bother with packages? Well, they provide a way to logically group related modules. Instead of having an `xml` package with `sax` and `dom` packages inside, the authors could have chosen to put all the `sax` functionality in `xmlsax.py` and all the `dom` functionality in `xmldom.py`, or even put all of it in a single module. But that would have been unwieldy (as of this writing, the XML package has over 3000 lines of code) and difficult to manage (separate source files mean multiple people can work on different areas simultaneously).

If you ever find yourself writing a large subsystem in Python (or, more likely, when you realize that your small subsystem has grown into a large one), invest some time designing a good package architecture. It s one of the many things Python is good at, so take advantage of it.

5.3. Parsing XML

As I was saying, actually parsing an XML document is very simple: one line of code. Where you go from there is up to you.

Example 5.8. Loading an XML document (for real this time)

```
>>> 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 As we saw in the previous section, this imports the `minidom` module from the `xml.dom` package.

❷ Here is the one line of code that does all the work: `minidom.parse` takes one argument and returns a parsed representation of the XML document. The argument can be many things; in this case, it s simply a filename of an XML document on my local disk. (To follow along, you ll need to change the path to point to your downloaded examples directory.) But you can also pass a file object, or even a file−like object. We ll take advantage of this flexibility later in this chapter.

❸ The object returned from `minidom.parse` is a `Document` object, a descendant of the `Node` class. This `Document` object is the root level of a complex tree−like structure of interlocking Python objects that completely represent the XML document we passed to `minidom.parse`.

❹ `toxml` is a method of the `Node` class (and is therefore available on the `Document` object we got from `minidom.parse`). `toxml` prints out the XML that this `Node` represents. For the `Document` node, this prints out the entire XML document.

Now that we have an XML document in memory, we can start traversing through it.

Example 5.9. Getting child nodes

```
>>> 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 Every `Node` has a `childNodes` attribute, which is a list of the `Node` objects. A `Document` always has only one child node, the root element of the XML document (in this case, the `grammar` element).

❷ To get the first (and in this case, the only) child node, just use regular list syntax. Remember, there is nothing special going on here; this is just a regular Python list of regular Python objects.

❸

Since getting the first child node of a node is a useful and common activity, the `Node` class has a `firstChild` attribute, which is synonymous with `childNodes[0]`. (There is also a `lastChild` attribute, which is synonymous with `childNodes[-1]`.)

Example 5.10. `toxml` works on any node

```
>>> 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 Since the `toxml` method is defined in the `Node` class, it is available on any XML node, not just the `Document` element.

Example 5.11. Child nodes can be text

```
>>> 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 Looking at the XML in `binary.xml`, you might think that the `grammar` has only two child nodes, the two `ref` elements. But you re missing something: the carriage returns! After the `'<grammar>'` and before the first `'<ref>'` is a carriage return, and this text counts as a child node of the `grammar` element. Similarly, there is a carriage return after each `'</ref>'`; these also count as child nodes. So `grammar.childNodes` is actually a list of 5 objects: 3 `Text` objects and 2 `Element` objects.

❷ The first child is a `Text` object representing the carriage return after the `'<grammar>'` tag and before the first `'<ref>'` tag.

❸ The second child is an `Element` object representing the first `ref` element.

❹ The fourth child is an `Element` object representing the second `ref` element.

❺

The last child is a `Text` object representing the carriage return after the `'</ref>'` end tag and before the `'</grammar>'` end tag.

Example 5.12. Drilling down all the way to text

```
>>> 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 As we saw in the previous example, the first `ref` element is `grammarNode.childNodes[1]`, since childNodes[0] is a `Text` node for the carriage return.

❷ The `ref` element has its own set of child nodes, one for the carriage return, a separate one for the spaces, one for the `p` element, and so forth.

❸ You can even use the `toxml` method here, deeply nested within the document.

❹ The `p` element has only one child node (you can t tell that from this example, but look at `pNode.childNodes` if you don t believe me), and it is a `Text` node for the single character `'0'`.

❺ The `.data` attribute of a `Text` node gives you the actual string that the text node represents. But what is that `'u'` in front of the string? The answer to that deserves its own section.

5.4. Unicode

Unicode is a system to represent characters from all the world s different languages. When Python parses an XML document, all data is stored in memory as unicode.

We ll get to all that in a minute, but first, some background.

Historical note. Before unicode, there were separate character encoding systems for each language, each using the same numbers (0−255) to represent that language s characters. Some languages (like Russian) had multiple conflicting standards about how to represent the same characters; other languages (like Japanese) had so many characters that they required multiple character sets. Exchanging documents between systems was difficult because there was no way for a computer to tell for certain which character encoding scheme the document author had used; the computer only saw numbers, and the numbers could mean different things. Then think about trying to store these documents in the same place (like in the same database table); you would need to store the character encoding alongside each piece of text, and make sure to pass it around whenever you passed the text around. Then think about multilingual documents, with characters from multiple languages in the same document. (They typically used escape codes to switch modes; poof, we re in Russian koi8−r mode, so character 241 means this; poof, now we re in Mac Greek mode, so character 241 means something else. And so on.) These are the problems which unicode was designed

to solve.

To solve these problems, unicode represents each character as a 2−byte number, from 0 to 65535.[10] Each 2−byte number represents a unique character used in at least one of the world s languages. (Characters that are used in multiple languages have the same numeric code.) There is exactly 1 number per character, and exactly 1 character per number. Unicode data is never ambiguous.

Of course, there is still the matter of all these legacy encoding systems. 7−bit ASCII, for instance, which stores English characters as numbers ranging from 0 to 127. (65 is capital "A", 97 is lowercase "a", and so forth.) English has a very simple alphabet, so it can be completely expressed in 7−bit ASCII. Western European languages like French, Spanish, and German all use an encoding system called ISO−8859−1 (also called "latin−1"), which uses the 7−bit ASCII characters for the numbers 0 through 127, but then extends into the 128−255 range for characters like n−with−a−tilde−over−it (241), and u−with−two−dots−over−it (252). And unicode uses the same characters as 7−bit ASCII for 0 through 127, and the same characters as ISO−8859−1 for 128 through 255, and then extends from there into characters for other languages with the remaining numbers, 256 through 65535.

When dealing with unicode data, you may at some point need to convert the data back into one of these other legacy encoding systems. For instance, to integrate with some other computer system which expects its data in a specific 1−byte encoding scheme, or to print it to a non−unicode−aware terminal or printer. Or to store it in an XML document which explicitly specifies the encoding scheme.

And on that note, let s get back to Python.

Python has had unicode support throughout the language since version 2.0.[11] The XML package uses unicode to store all parsed XML data, but you can use unicode anywhere.

Example 5.13. Introducing unicode

```
>>> s = u'Dive in'          ❶
>>> s
u'Dive in'
>>> print s                 ❷
Dive in
```

❶ To create a unicode string instead of a regular ASCII string, add the letter "u" before the string. Note that this particular string doesn t have any non−ASCII characters. That s fine; unicode is a superset of ASCII (a very large superset at that), so any regular ASCII string can also be stored as unicode.

❷ When printing a string, Python will attempt to convert it to your default encoding, which is usually ASCII. (More on this in a minute.) Since this unicode string is made up of characters that are also ASCII characters, printing it has the same result as printing a normal ASCII string; the conversion is seamless, and if you didn t know that s was a unicode string, you d never notice the difference.

Example 5.14. Storing non−ASCII characters

```
>>> 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

The real advantage of unicode, of course, is its ability to store non−ASCII characters, like the Spanish "ñ" (n with a tilde over it). The unicode character code for the tilde−n is `0xf1` in hexadecimal (241 in decimal), which you can type like this: `\xf1`.

❷ Remember I said that the `print` function attempts to convert a unicode string to ASCII so it can print it? Well, that s not going to work here, because our unicode string contains non−ASCII characters, so Python raises a `UnicodeError` error.

❸ Here s where the conversion−from−unicode−to−other−encoding−schemes comes in. `s` is a unicode string, but `print` can only print a regular string. To solve this problem, we call the `encode` method, available on every unicode string, to convert the unicode string to a regular string in the given encoding scheme, which we pass as a parameter. In this case, we re using `latin−1` (also known as `iso−8859−1`), which includes the tilde−n (whereas the default ASCII encoding scheme did not, since it only includes characters numbered 0 through 127).

Remember I said Python usually converted unicode to ASCII whenever it needed to make a regular string out of a unicode string? Well, this default encoding scheme is an option which you can customize.

Example 5.15. `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` is a special script; Python will try to import it on startup, so any code in it will be run automatically. As the comment mentions, it can go anywhere (as long as `import` can find it), but it usually goes in the `site−packages` directory within your Python `lib` directory.

❷ `setdefaultencoding` function sets, well, the default encoding. This is the encoding scheme that Python will try to use whenever it needs to auto−coerce a unicode string into a regular string.

Example 5.16. Effects of setting the default encoding

```
>>> import sys
>>> sys.getdefaultencoding() ❶
'iso−8859−1'
>>> s = u'La Pe\xf1a'
>>> print s                  ❷
La Peña
```

❶ This example assumes that you have made the changes listed in the previous example to your `sitecustomize.py` file, and restarted Python. If your default encoding still says `'ascii'`, you didn t set up your `sitecustomize.py` properly, or you didn t restart Python. The default encoding can only be changed during Python startup; you can t change it later. (Due to some wacky programming tricks that I won t get into right now, you can t even call `sys.setdefaultencoding` after Python has started up. Dig into `site.py` and search for "`setdefaultencoding`" to find out how.)

❷ Now that the default encoding scheme includes all the characters we use in our string, Python has no problem auto−coercing the string and printing it.

Now, what about XML? Well, every XML document is in a specific encoding. Again, ISO−8859−1 is a popular encoding for data in Western European languages. KOI8−R is popular for Russian texts. The encoding, if specified, is in the header of the XML document.

Example 5.17. `russiansample.xml`

```
<?xml version="1.0" encoding="koi8-r"?>       ❶
<preface>
<title> @548A;>285</title>                    ❷
</preface>
```

❶ This is a sample extract from a real Russian XML document; it s part of the translation of the Preface of this book. Note the encoding, `koi8-r`, specified in the header.

❷ These are Cyrillic characters which, as far as I know, spell the Russian word for "Preface". If you open this file in a regular text editor, the characters will most likely like gibberish, because they re encoded using the `koi8-r` encoding scheme, but they re being displayed in `iso-8859-1`.

Example 5.18. Parsing `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 I'm assuming here that you saved the previous example as `russiansample.xml` in the current directory. I am also, for the sake of completeness, assuming that you ve changed your default encoding back to `'ascii'` by removing your `sitecustomize.py` file, or at least commenting out the `setdefaultencoding` line.

❷ Note that the text data of the `title` tag (now in the `title` variable, thanks to that long concatenation of Python functions which I hastily skipped over and, annoyingly, won t explain until the next section) −− the text data inside the XML document s `title` element is stored in unicode.

❸ Printing the title is not possible, because this unicode string contains non−ASCII characters, so Python can t convert it to ASCII because that doesn t make sense.

❹ We can, however, explicitly convert it to `koi8-r`, in which case we get a (regular, not unicode) string of single−byte characters (`f0`, `d2`, `c5`, and so forth) that are the `koi8-r`−encoded versions of the characters in the original unicode string.

❺ Printing the `koi8-r`−encoded string will probably show gibberish on your screen, because your Python IDE is interpreting those characters as `iso-8859-1`, not `koi8-r`. But at least they do print. (And, if you look carefully, it s the same gibberish that you saw when you opened the original XML document in a non−unicode−aware text editor. Python converted it from `koi8-r` into unicode when it parsed the XML document, and we ve just converted it back.)

To sum up, unicode itself is a bit intimidating if you ve never seen it before, but unicode data is really very easy to handle in Python. If your XML documents are all 7−bit ASCII (like the examples in this chapter), you will literally never think about unicode. Python will convert the ASCII data in the XML documents into unicode while parsing, and auto−coerce it back to ASCII whenever necessary, and you ll never even notice. But if you need to deal with dat in other languages, Python is ready.

Further reading

- Unicode.org (http://www.unicode.org/) is the home page of the unicode standard, including a brief technical introduction (http://www.unicode.org/standard/principles.html).
- Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) has some more examples of how to use Python s unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn t really want to.
- Unicode Proposal (http://www.lemburg.com/files/python/unicode−proposal.txt) is the original technical specification for Python s unicode functionality. For advanced unicode hackers only.

5.5. Searching for elements

Traversing XML documents by stepping through each node can be tedious. If you re looking for something in particular, buried deep within your XML document, there is a shortcut you can use to find it quickly: `getElementsByTagName`.

For this section, we ll be using the `binary.xml` grammar file, which looks like this:

Example 5.19. `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>
```

It has two `ref`s, `'bit'` and `'byte'`. A `bit` is either a `'0'` or `'1'`, and a `byte` is 8 `bit`s.

Example 5.20. Introducing `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` takes one argument, the name of the element you wish to find. It returns a list of `Element` objects, corresponding to the XML elements that have that name. In this case, we find two `ref` elements.

Example 5.21. Every element is searchable

```
>>> 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 Continuing from the previous example, the first object in our `reflist` is the `'bit' ref` element.

❷ We can use the same `getElementsByTagName` method on this `Element` to find all the `<p>` elements within the `'bit' ref` element.

❸ Just as before, the `getElementsByTagName` method returns a list of all the elements it found. In this case, we have two, one for each bit.

Example 5.22. Searching is actually recursive

```
>>> 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 Note carefully the difference between this and the previous example. Previously, we were searching for `p` elements within `firstref`, but here we are searching for `p` elements within `xmldoc`, the root−level object that represents the entire XML document. This *does* find the `p` elements nested within the `ref` elements within the root `grammar` element.

❷ The first two `p` elements are within the first `ref` (the `'bit' ref`).

❸ The last `p` element is the one within the second `ref` (the `'byte' ref`).

5.6. Accessing element attributes

XML elements can have one or more attributes, and it is incredibly simple to access them once you have parsed an XML document.

For this section, we ll be using the `binary.xml` grammar file that we saw in the previous section.

This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When we parse an XML document, we get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was

confusing. I am open to suggestions on how to distinguish these more clearly.

Example 5.23. Accessing element attributes

```
>>> 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 Each `Element` object has an attribute called `attributes`, which is a `NamedNodeMap` object. This sounds scary, but it s not, because a `NamedNodeMap` is an object that acts like a dictionary, so you already know how to use it.

❷ Treating the `NamedNodeMap` as a dictionary, we can get a list of the names of the attributes of this element by using `attributes.keys()`. This element has only one attribute, `'id'`.

❸ Attribute names, like all other text in an XML document, are stored in unicode.

❹ Again treating the `NamedNodeMap` as a dictionary, we can get a list of the values of the attributes by using `attributes.values()`. The values are themselves objects, of type `Attr`. We ll see how to get useful information out of this object in the next example.

❺ Still treating the `NamedNodeMap` as a dictionary, we can access an individual attribute by name, using normal dictionary syntax. (Readers who have been paying extra−close attention will already know how the `NamedNodeMap` class accomplishes this neat trick: by defining a `__getitem__` special method. Other readers can take comfort in the fact that they don t need to understand how it works in order to use it effectively.)

Example 5.24. Accessing individual attributes

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name  ❶
u'id'
>>> a.value ❷
u'bit'
```

❶ The `Attr` object completely represents a single XML attribute of a single XML element. The name of the attribute (the same name as we used to find this object in the `bitref.attributes` `NamedNodeMap` pseudo−dictionary) is stored in `a.name`.

❷ The actual text value of this XML attribute is stored in `a.value`.

Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

5.7. Abstracting input sources

One of Python s greatest strengths is its dynamic binding, and one powerful use of dynamic binding is the *file−like object*.

Many functions which require an input source could simply take a filename, go open the file for reading, read it, and close it when they re done. But they don t. Instead, they take a *file−like object*.

In the simplest case, a *file−like object* is any object with a `read` method with an optional `size` parameter, which returns a string. When called with no `size` parameter, it reads everything there is to read from the input source and returns all the data as a single string. When called with a `size` parameter, it reads that much from the input source and returns that much data; when called again, it picks up where it left off and returns the next chunk of data.

This is how reading from real files works; the difference is that we re not limiting ourselves to real files. The input source could be anything: a file on disk, a web page, even a hard−coded string. As long as we pass a file−like object to the function, and the function simply calls the object s `read` method, the function can handle any kind of input source without specific code to handle each kind.

In case you were wondering how this relates to XML processing, `minidom.parse` is one such function which can take a file−like object.

Example 5.25. Parsing XML from a file

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')     ❶
>>> xmldoc = minidom.parse(fsock) ❷
>>> fsock.close()                 ❸
>>> print xmldoc
<?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 First, we open the file on disk. This gives us a file object.

❷ We pass the file object to `minidom.parse`, which calls the `read` method of `fsock` and reads the XML document from the file on disk.

❸ Be sure to call the `close` method of the file object after we re done with it. `minidom.parse` will not do this for you.

Well, that all seems like a colossal waste of time. After all, we ve already seen that `minidom.parse` can simply take the filename and do all the opening and closing nonsense automatically. And it s true that if you know you re just going to be parsing a local file, you can pass the filename and `minidom.parse` is smart enough to Do The Right Thing(tm). But notice how similar −− and easy −− it is to parse an XML document straight from the Internet.

Example 5.26. Parsing XML from a 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 As we saw in the previous chapter, `urlopen` takes a web page URL and returns a file-like object. Most importantly, this object has a `read` method which returns the HTML source of the web page.

❷ Now we pass the file-like object to `minidom.parse`, which obediently calls the `read` method of the object and parses the XML data that the `read` method returns. The fact that this XML data is now coming straight from a web page is completely irrelevant. `minidom.parse` doesn t know about web pages, and it doesn t care about web pages; it just knows about file-like objects.

❸ As soon as you re done with it, be sure to close the file-like object that `urlopen` gives you.

❹ By the way, this URL is real, and it really is XML. It s an XML representation of the current headlines on Slashdot (http://slashdot.org/), a technical news and gossip site.

Example 5.27. Parsing XML from a string (the easy but inflexible way)

```
>>> 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` has a method, `parseString`, which takes an entire XML document as a string and parses it. You can use this instead of `minidom.parse` if you know you already have your entire XML document in a string.

OK, so we can use the `minidom.parse` function for parsing both local files and remote URLs, but for parsing strings, we use... a different function. That means that if we want to be able to take input from a file, a URL, or a string, we ll need special logic to check whether it s a string, and call the `parseString` function instead. How unsatisfying.

If there were a way to turn a string into a file-like object, then we could simply pass this object to `minidom.parse`. And in fact, there is a module specifically designed for doing just that: `StringIO`.

Example 5.28. Introducing `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 The `StringIO` module contains a single class, also called `StringIO`, which allows you to turn a string into a file−like object. The `StringIO` class takes the string as a parameter when creating an instance.

❷ Now we have a file−like object, and we can do all sorts of file−like things with it. Like `read`, which returns the original string.

❸ Calling `read` again returns an empty string. This is how real file objects work too; once you read the entire file, you can t read any more without explicitly seeking to the beginning of the file. The `StringIO` object works the same way.

❹ You can explicitly seek to the beginning of the string, just like seeking through a file, by using the `seek` method of the `StringIO` object.

❺ You can also read the string in chunks, by passing a `size` parameter to the `read` method.

❻ At any time, `read` will return the rest of the string that you haven t read yet. All of this is exactly how file objects work; hence the term *file−like object*.

Example 5.29. Parsing XML from a string (the file−like object way)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) ❶
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

❶ Now we can pass the file−like object (really a `StringIO`) to `minidom.parse`, which will call the object s `read` method and happily parse away, never knowing that its input came from a hard−coded string.

So now we know how to use a single function, `minidom.parse`, to parse an XML document stored on a web page, in a local file, or in a hard−coded string. For a web page, we use `urlopen` to get a file−like object; for a local file, we use `open`; and for a string, we use `StringIO`. Now let s take it one step further and generalize *these* differences as well.

Example 5.30. `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 The `openAnything` function takes a single parameter, `source`, and returns a file−like object. `source` is a string of some sort; it can either be a URL (like `'http://slashdot.org/slashdot.rdf'`), a full or partial pathname to a local file (like `'binary.xml'`), or a string that contains actual XML data to be parsed.

❷ First, we see if `source` is a URL. We do this through brute force: we try to open it as a URL and silently ignore errors caused by trying to open something which is not a URL. This is actually elegant in the sense that, if `urllib` ever supports new types of URLs in the future, we will also support them without recoding.

❸ If `urllib` yelled at us and told us that `source` wasn t a valid URL, we assume it s a path to a file on disk and try to open it. Again, we don t do anything fancy to check whether `source` is a valid filename or not (the rules for valid filenames vary wildly between different platforms anyway, so we d probably get them wrong anyway). Instead, we just blindly open the file, and silently trap any errors.

❹ By this point, we have to assume that `source` is a string that has hard−coded data in it (since nothing else worked), so we use `StringIO` to create a file−like object out of it and return that. (In fact, since we re using the `str` function, `source` doesn t even need to be a string; it could be any object, and we ll use its string representation, as defined by its `__str__` special method.)

Now we can use this `openAnything` function in conjunction with `minidom.parse` to make a function that takes a `source` that refers to an XML document somehow (either as a URL, or a local filename, or a hard−coded XML document in a string) and parses it.

Example 5.31. Using `openAnything` in `kgp.py`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

5.8. Standard input, output, and error

UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated `stdout` and `stderr`) are pipes that are built into every UNIX system. When you `print` something, it goes to the `stdout` pipe; when your program crashes and prints out debugging information (like a traceback in Python), it goes to the `stderr` pipe. Both of these pipes are ordinarily just connected to the terminal window where you are working, so when a program prints, you see the output, and when a program crashes, you see the debugging information. (If you re working on a system with a window−based Python IDE, `stdout` and `stderr` default to your "Interactive Window".)

Example 5.32. Introducing `stdout` and `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 As we saw in Example 3.28, Compteurs simples , we can use Python s built−in `range` function to build simple counter loops that repeat something a set number of times.

❷ `stdout` is a file−like object; calling its `write` function will print out whatever string you give it. In fact, this is what the `print` function really does; it adds a carriage return to the end of the string you re printing, and calls `sys.stdout.write`.

❸ In the simplest case, `stdout` and `stderr` send their output to the same place: the Python IDE (if you re in one), or the terminal (if you re running Python from the command line). Like `stdout`, `stderr` does not add carriage returns for you; if you want them, add them yourself.

`stdout` and `stderr` are both file−like objects, like the ones we discussed in Section 5.7, Abstracting input sources , but they are both write−only. They have no `read` method, only `write`. Still, they are file−like objects, and you can assign any other file− or file−like object to them to redirect their output.

Example 5.33. Redirecting output

```
[f8dy@oliver kgp]$ python stdout.py
Dive in
[f8dy@oliver kgp]$ cat out.log
This message will be logged instead of displayed
```

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−4.1.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 This will print to the IDE "Interactive Window" (or the terminal, if running the script from the command line).

❷ Always save `stdout` before redirecting it, so you can set it back to normal later.

❸ Open a new file for writing.

❹ Redirect all further output to the new file we just opened.

❺ This will be "printed" to the log file only; it will not be visible in the IDE window or on the screen.

❻ Set `stdout` back to the way it was before we mucked with it.

❼ Close the log file.

Redirecting `stderr` works exactly the same way, using `sys.stderr` instead of `sys.stdout`.

Example 5.34. Redirecting error information

```
[f8dy@oliver kgp]$ python stderr.py
[f8dy@oliver 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−4.1.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 Open the log file where we want to store debugging information.

❷ Redirect standard error by assigning the file object of our newly−opened log file to `stderr`.

❸ Raise an exception. Note from the screen output that this does *not* print anything on screen. All the normal traceback information has been written to `error.log`.

❹ Also note that we re not explicitly closing our log file, nor are we setting `stderr` back to its original value. This is fine, since once the program crashes (due to our exception), Python will clean up and close the file for us, and it doesn t make any difference that `stderr` is never restored, since, as I mentioned, the program crashes and Python ends. Restoring the original is more important for `stdout`, if you expect to go do other stuff within the same script afterwards.

Standard input, on the other hand, is a read−only file object, and it represents the data flowing into the program from some previous program. This will likely not make much sense to classic Mac OS users, or even Windows users unless you were ever fluent on the MS−DOS command line. The way it works is that you can construct a chain of commands in a single line, so that one program s output becomes the input for the next program in the chain. The first program simply outputs to standard output (without doing any special redirecting itself, just doing normal `print` statements or whatever), and the next program reads from standard input, and the operating system takes care of connecting one program s output to the next program s input.

Example 5.35. Chaining commands

```
[f8dy@oliver kgp]$ python kgp.py −g binary.xml        ❶
01100111
[f8dy@oliver 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>
[f8dy@oliver kgp]$ cat binary.xml | python kgp.py -g - ❸ ❹
10110001
```

❶ As we saw in Section 5.1, Diving in , this will print a string of eight random bits, `0` or `1`.

❷ This simply prints out the entire contents of `binary.xml`. (Windows users should use `type` instead of `cat`.)

❸ This prints the contents of `binary.xml`, but the "`|`" character, called the "pipe" character, means that the contents will not be printed to the screen. Instead, they will become the standard input of the next command, which in this case calls our Python script.

❹ Instead of specifying a module (like `binary.xml`), we specify "`-`", which causes our script to load the grammar from standard input instead of from a specific file on disk. (More on how this happens in the next example.) So the effect is the same as the first syntax, where we specified the grammar filename directly, but think of the expansion possibilities here. Instead of simply doing `cat binary.xml`, we could run a script that dynamically generates the grammar, then we can pipe it into our script. It could come from anywhere: a database, or some grammar-generating meta-script, or whatever. The point is that we don t have to change our `kgp.py` script at all to incorporate any of this functionality. All we have to do is be able to take grammar files from standard input, and we can separate all the other logic into another program.

So how does our script "know" to read from standard input when the grammar file is "`-`"? It s not magic; it s just code.

Example 5.36. Reading from standard input in `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 This is the `openAnything` function from `toolbox.py`, which we previously examined in Section 5.7, Abstracting input sources . All we ve done is add three lines of code at the beginning of the function to check if the source is "`-`"; if so, we return `sys.stdin`. Really, that s it! Remember, `stdin` is a file-like object with a `read` method, so the rest of our code (in `kgp.py`, where we call `openAnything`) doesn t change a bit.

5.9. Caching node lookups

`kgp.py` employs several tricks which may or may not be useful to you in your XML processing. The first one takes advantage of the consistent structure of the input documents to build a cache of nodes.

A grammar file defines a series of `ref` elements. Each `ref` contains one or more `p` elements, which can contain lots of different things, including `xref`s. Whenever we encounter an `xref`, we look for a corresponding `ref` element with the same `id` attribute, and choose one of the `ref` element s children and parse it. (We ll see how this random choice is made in the next section.)

This is how we build up our grammar: define `ref` elements for the smallest pieces, then define `ref` elements which "include" the first `ref` elements by using `xref`, and so forth. Then we parse the "largest" reference and follow each

xref, and eventually output real text. The text we output depends on the (random) decisions we make each time we fill in an xref, so the output is different each time.

This is all very flexible, but there is one downside: performance. When we find an xref and need to find the corresponding ref element, we have a problem. The xref has an id attribute, and we want to find the ref element that has that same id attribute, but there is no easy way to do that. The slow way to do it would be to get the entire list of ref elements each time, then manually loop through and look at each id attribute. The fast way is to do that once and build a cache, in the form of a dictionary.

Example 5.37. `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 Start by creating an empty dictionary, self.refs.

❷ As we saw in Section 5.5, Searching for elements , getElementsByTagName returns a list of all the elements of a particular name. We easily can get a list of all the ref elements, then simply loop through that list.

❸ As we saw in Section 5.6, Accessing element attributes , we can access individual attributes of an element by name, using standard dictionary syntax. So the keys of our self.refs dictionary will be the values of the id attribute of each ref element.

❹ The values of our self.refs dictionary will be the ref elements themselves. As we saw in Section 5.3, Parsing XML , each element, each node, each comment, each piece of text in a parsed XML document is an object.

Once we build this cache, whenever we come across an xref and need to find the ref element with the same id attribute, we can simply look it up in self.refs.

Example 5.38. Using our `ref` element cache

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

We ll explore the randomChildElement function in the next section.

5.10. Finding direct children of a node

Another useful techique when parsing XML documents is finding all the direct child elements of a particular element. For instance, in our grammar files, a ref element can have several p elements, each of which can contain many things, including other p elements. We want to find just the p elements that are children of the ref, not p elements that are children of other p elements.

You might think we could simply use getElementsByTagName for this, but we can t. getElementsByTagName searches recursively and returns a single list for all the elements it finds. Since p elements can contain other p elements, we can t use getElementsByTagName, because it would return nested p elements that we don t want. To find only direct child elements, we ll need to do it ourselves.

Example 5.39. Finding direct child elements

```
    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 As we saw in Example 5.9, Getting child nodes , the `childNodes` attribute returns a list of all the child nodes of an element.

❷ However, as we saw in Example 5.11, Child nodes can be text , the list returned by `childNodes` contains all different types of nodes, including text nodes. That s not what we re looking for here. We only want the children that are elements.

❸ Each node has a `nodeType` attribute, which can be `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE`, or any number of other values. The complete list of possible values is in the `__init__.py` file in the `xml.dom` package. (See Section 5.2, Packages for more on packages.) But we re just interested in nodes that are elements, so we can filter the list to only include those nodes whose `nodeType` is `ELEMENT_NODE`.

❹ Once we have a list of actual elements, choosing a random one is easy. Python comes with a module called `random` which includes several useful functions. The `random.choice` function takes a list of any number of items and returns a random item. In this case, the list contains `p` elements, so `chosen` is now a `p` element selected at random from the children of the `ref` element we were given.

5.11. Creating separate handlers by node type

The third useful XML processing tip involves separating your code into logical functions, based on node types and element names. Parsed XML documents are made up of various types of nodes, each represented by a Python object. The root level of the document itself is represented by a `Document` object. The `Document` then contains one or more `Element` objects (for actual XML tags), each of which may contain other `Element` objects, `Text` objects (for bits of text), or `Comment` objects (for embedded comments). Python makes it easy to write a dispatcher to separate the logic for each node type.

Example 5.40. Class names of parsed XML objects

```
>>> 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 Assume for a moment that `kant.xml` is in the current directory.

❷ As we saw in Section 5.2, Packages , the object returned by parsing an XML document is a `Document` object, as defined in the `minidom.py` in the `xml.dom` package. As we saw in Section 3.4, Instantiation de classes , `__class__` is built−in attribute of every Python object.

❸ Furthermore, `__name__` is a built−in attribute of every Python class, and it is a string. This string is not mysterious; it s the same as the class name you type when you define a class yourself. (See Section 3.3, Définition de classes .)

Fine, so now we can get the class name of any particular XML node (since each XML node is represented as a Python object). How can we use this to our advantage to separate the logic of parsing each node type? The answer is `getattr`, which we first saw in Section 2.4, Obtenir des références objet avec getattr .

Example 5.41. `parse`, a generic XML node dispatcher

```
def parse(self, node):
    parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) ❶ ❷
    parseMethod(node) ❸
```

❶ First off, notice that we re constructing a larger string based on the class name of the node we were passed (in the `node` argument). So if we re passed a `Document` node, we re constructing the string `'parse_Document'`, and so forth.

❷ Now we can treat that string as a function name, and get a reference to the function itself using `getattr`

❸ Finally, we can call that function and pass the node itself as an argument. The next example shows the definitions of each of these functions.

Example 5.42. Functions called by the `parse` dispatcher

```
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 `parse_Document` is only ever called once, since there is only one `Document` node in an XML document, and only one `Document` object in the parsed XML representation. It simply turns around and parses the root element of the grammar file.

❷ `parse_Text` is called on nodes that represent bits of text. The function itself does some special processing to handle automatic capitalization of the first word of a sentence, but otherwise simply appends the represented text to a list.

❸ `parse_Comment` is just a `pass`, since we don t care about embedded comments in our grammar files. Note, however, that we still need to define the function and explicitly make it do nothing. If the function did not exist, our generic `parse` function would fail as soon as it stumbled on a comment, because it would try to find the non−existent `parse_Comment` function. Defining a separate function for every node type, even ones we don t use, allows the generic `parse` function to stay simple and dumb.

❹ The `parse_Element` method is actually itself a dispatcher, based on the name of the element s tag. The basic idea is the same: take what distinguishes elements from each other (their tag names) and dispatch to a separate function for each of them. We construct a string like `'do_xref'` (for an `<xref>` tag), find a function of that name, and call it. And so forth for each of the other tag names that might be found in the course of parsing a grammar file (`<p>` tags, `<choice>` tags).

In this example, the dispatch functions `parse` and `parse_Element` simply find other methods in the same class. If your processing is very complex (or you have many different tag names), you could break up your code into separate modules, and use dynamic importing to import each module and call whatever functions you needed. Dynamic importing will be discussed in Chapter 7, *Data−Centric Programming*.

5.12. Handling command line arguments

Python fully supports creating programs that can be run on the command line, complete with command−line arguments and either short− or long−style flags to specify various options. None of this is XML−specific, but this script makes good use of command−line processing, so it seemed like a good time to mention it.

It s difficult to talk about command line processing without understanding how command line arguments are exposed to your Python program, so let s write a simple program to see them.

Example 5.43. Introducing `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−4.1.zip) du livre.

```
#argecho.py
import sys

for arg in sys.argv: ❶
    print arg
```

❶ Each command line argument passed to the program will be in `sys.argv`, which is just a list. Here we are printing each argument on a separate line.

Example 5.44. The contents of `sys.argv`

```
[f8dy@oliver py]$ python argecho.py              ❶
argecho.py
[f8dy@oliver py]$ python argecho.py abc def      ❷
argecho.py
abc
def
[f8dy@oliver py]$ python argecho.py −−help       ❸
argecho.py
−−help
[f8dy@oliver py]$ python argecho.py −m kant.xml ❹
argecho.py
−m
kant.xml
```

❶ The first thing to know about `sys.argv` is that it contains the name of the script we re calling. We will actually use this knowledge to our advantage later, in Chapter 7, *Data−Centric Programming*. Don t worry about it for now.

❷ Command line arguments are separated by spaces, and each shows up as a separate element in the `sys.argv` list.

❸ Command line flags, like `−−help`, also show up as their own element in the `sys.argv` list.

❹ To make things even more interesting, some command line flags themselves take arguments. For instance, here we have a flag (`−m`) which takes an argument (`kant.xml`). Both the flag itself and the flag s argument are simply sequential elements in the `sys.argv` list. No attempt is made to associate one with the other; all you get is a list.

So as we can see, we certainly have all the information passed on the command line, but then again, it doesn t look like it s going to be all that easy to actually use it. For simple programs that only take a single argument and have no flags, you can simply use `sys.argv[1]` to access the argument. There s no shame in this; I do it all the time. For more complex programs, you need the `getopt` module.

Example 5.45. Introducing `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 First off, look at the bottom of the example and notice that we re calling the `main` function with `sys.argv[1:]`. Remember, `sys.argv[0]` is the name of the script that we re running; we don t care about that for command line processing, so we chop it off and pass the rest of the list.

❷ This is where all the interesting processing happens. The `getopt` function of the `getopt` takes three parameters: the argument list (which we got from `sys.argv[1:]`), a string containing all the possible single−character command line flags that this program accepts, and a list of longer command line flags that are equivalent to the single−character versions. This is quite confusing at first glance, and is explained in more detail below.

❸ If anything goes wrong trying to parse these command line flags, `getopt` will raise an exception, which we catch. We told `getopt` all the flags we understand, so this probably means that the end user passed some command line flag that we don t understand.

❹ As is standard practice in the UNIX world, when our script is passed flags it doesn t understand, we print out a summary of proper usage and exit gracefully. Note that I haven t shown the `usage` function here. We would still need to code that somewhere and have it print out the appropriate summary; it s not automatic.

So what are all those parameters we pass to the `getopt` function? Well, the first one is simply the raw list of command line flags and arguments (not including the first element, the script name, which we already chopped off before calling our `main` function). The second is the list of short command line flags that our script accepts.

`"hg:d"`

−h
: print usage summary

−g ...
: use specified grammar file or URL

−d
: show debugging information while parsing

The first and third flags are simply standalone flags; you specify them or you don t, and they do things (print help) or change state (turn on debugging). However, the second flag (`−g`) *must* be followed by an argument, which is the name of the grammar file to read from. In fact it can be a filename or a web address, and we don t know which yet (we ll figure it out later), but we know it has to be *something*. So we tell `getopt` this by putting a colon after the `g` in that second parameter to the `getopt` function.

To further complicate things, our script accepts either short flags (like `−h`) or long flags (like `−−help`), and we want them to do the same thing. This is what the third parameter to `getopt` is for, to specify a list of the long flags that correspond to the short flags we specified in the second parameter.

```
["help", "grammar="]
```

--help
: print usage summary

--grammar ...
: use specified grammar file or URL

Three things of note here:

1. All long flags are preceded by two dashes on the command line, but we don t include those dashes when calling `getopt`. They are understood.
2. The `--grammar` flag must always be followed by an additional argument, just like the `-g` flag. This is notated by an equals sign, `"grammar="`.
3. The list of long flags is shorter than the list of short flags, because the `-d` flag does not have a corresponding long version. This is fine; only `-d` will turn on debugging. But the order of short and long flags needs to be the same, so you ll need to specify all the short flags that *do* have corresponding long flags first, then all the rest of the short flags.

Confused yet? Let s look at the actual code and see if it makes sense in context.

Example 5.46. Handling command-line arguments in `kgp.py`

```
def main(argv):                                  ❶
    grammar = "kant.xml"
    try:
    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 The `grammar` variable will keep track of the grammar file we re using. We initialize it here in case it s not specified on the command line (using either the `-g` or the `--grammar` flag).

❷ The `opts` variable that we get back from `getopt` contains a list of tuples, flag and argument. If the flag doesn t take an argument, then `arg` will simply be `None`. This makes it easier to loop through the flags.

❸ `getopt` validates that the command line flags are acceptable, but it doesn t do any sort of conversion between short and long flags. If you specify the `-h` flag, `opt` will contain `"-h"`; if you specify the `--help` flag, `opt` will contain `"--help"`. So we need to check for both.

❹ Remember, the `-d` flag didn t have a corresponding long flag, so we only need to check for the short form. If we find it, we set a global variable that we ll refer to later to print out debugging information. (I used this during the development of the script. What, you thought all these examples worked on the first try?)

❺

If we find a grammar file, either with a `-g` flag or a `--grammar` flag, we save the argument that followed it (stored in `arg`) into our `grammar` variable, overwriting the default that we initialized at the top of the `main` function.

❻ That s it. We ve looped through and dealt with all the command line flags. That means that anything left must be command line arguments. These come back from the `getopt` function in the `args` variable. In this case, we re treating them as source material for our parser. If there are no command line arguments specified, `args` will be an empty list, and `source` will end up as the empty string.

5.13. Putting it all together

We ve covered a lot of ground. Let s step back and see how all the pieces fit together.

To start with, this is a script that takes its arguments on the command line, using the `getopt` module.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

We create a new instance of the `KantGenerator` class, and pass it the grammar file and source that may or may not have been specified on the command line.

```
    k = KantGenerator(grammar, source)
```

The `KantGenerator` instance automatically loads the grammar, which is an XML file. We use our custom `openAnything` function to open the file (which could be stored in a local file or a remote web server), then use the built−in `minidom` parsing functions to parse the XML into a tree of Python objects.

```
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
```

Oh, and along the way, we take advantage of our knowledge of the structure of the XML document to set up a little cache of references, which are just elements in the XML document.

```
    def loadGrammar(self, grammar):
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

If we specified some source material on the command line, we use that; otherwise we rip through the grammar looking for the "top−level" reference (that isn t referenced by anything else) and use that as a starting point.

```
    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)
```

Now we rip through our source material. The source material is also XML, and we parse it one node at a time. To

keep our code separated and more maintainable, we use separate handlers for each node type.

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

We bounce through the grammar, parsing all the children of each `p` element,

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

replacing `choice` elements with a random child,

```
    def do_choice(self, node):
        self.parse(self.randomChildElement(node))
```

and replacing `xref` elements with a random child of the corresponding `ref` element, which we previously cached.

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Eventually, we parse our way down to plain text,

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

which we print out.

```
def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

5.14. Summary

Python comes with powerful libraries for parsing and manipulating XML documents. The `minidom` takes an XML file and parses it into Python objects, providing for random access to arbitrary elements. Furthermore, this chapter shows how Python can be used to create a "real" standalone command−line script, complete with command−line flags, command−line arguments, error handling, even the ability to take input from the piped result of a previous program.

Before moving on to the next chapter, you should be comfortable doing all of these things:

- Parsing XML documents using `minidom`, searching through the parsed document, and accessing arbitrary element attributes and element children
- Organizing complex libraries into packages
- Converting unicode strings to different character encodings
- Chaining programs with standard input and output
- Defining command−line flags and validating them with `getopt`

[10] This, sadly, is *still* an oversimplification. Unicode now has been extended to handle ancient Chinese, Korean, and Japanese texts, which had so many different characters that the 2−byte unicode system could not represent them all. But Python doesn t currently support that out of the box, and I don t know if there is a project afoot to add it. You ve reached the limits of my expertise, sorry.

[11] Actually, Python has had unicode support since version 1.6, but version 1.6 was a contractual obligation release that nobody likes to talk about, a bastard stepchild of a hippie youth best left forgotten. Even the official Python documentation claims that unicode was "new in version 2.0". It s a lie, but, like the lies of presidents who say they inhaled but didn t enjoy it, we choose to believe it because we remember our own misspent youths a bit too vividly.

Chapter 6. Tests unitaires

6.1. Plonger

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 qu il se passe *avant* que le code soit écrit.

Dans ce chapitre, nous allons écrire un ensemble de fonction utilitaires pour convertir vers et depuis des nombres romains. Dans les nombres romains, il y a sept caractères qui sont répétés et combinés de différentes manières pour représenter des nombres.

1. `I` = `1`
2. `V` = `5`
3. `X` = `10`
4. `L` = `50`
5. `C` = `100`
6. `D` = `500`
7. `M` = `1000`

Il y a des règles générales pour construire des nombres romains :

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 nombres romains sont toujours écrits du plus haut au plus bas et lus de gauche à droite, donc l ordre des caractères est très important. `DC` est `600`, `CD` est un nombre complètement différent (`400`, "`100` de moins que `500`"). `CI` est `101`, `IC` n est pas même un nombre romain valide (car on ne peut soustraire `1` directement de `100`, on devrait l écrire `XCIX`, "`10` de moins que `100`, puis `1` de moins que `10`").

Ces règles amènent à un certain nombre d observations intéressantes :

1. Il n y a qu une seule façon correcte de représenter une valeur en nombres romains.
2. L inverse est aussi vrai : si une chaîne de caractères est un nombre romain 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 nombres 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 nombres romains vont de `1` à `3999`.)
4. Il n est pas possible de représenter `0` en nombres romains. (Etonnamment, les anciens romains n avaient pas de notion du `0` comme nombre. 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 nombres romains.

6. Il n est pas possible de représenter de valeurs décimales ou des fractions en nombres romains.

Sachant tout cela, qu attendriez−vous d un ensemble de fonctions pour convertir vers et depuis des nombres romains ?

Spécification de `roman.py`

1. `toRoman` doit retourner la représentation en nombres 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 romain valide et retourner la valeur qu il représente.
5. `fromRoman` doit échouer s il lui est passé un nombre romain invalide.
6. Si vous prenez un nombre, le convertissez en nombre romain, 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 nombre romain en majuscules.
8. `fromRoman` doit seulement accepter des nombres romains en majuscules (il doit échouer s il lui est passé une entrée en minuscules.

Pour en savoir plus

- Ce site (http://www.deadline.demon.co.uk/roman/front.htm) a plus d information sur les nombres romains, y compris une histoire (http://www.deadline.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).

6.2. Présentation de `romantest.py`

Maintenant que nous avons défini entièrement le comportement que nous attendons de nos fonctions de conversion, nous allons vers 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 le *refactoring* de code, ils garantissent que la nouvelle version se comporte comme l ancienne.
- Pendant la maintenace du code, ils permettent d être couvert si quelqu un vient hurler que votre dernière

modification fait planter leur code. ("Les tests unitaires passaient quand j ai intégré mon code...")

Voici la suite de tests complète de nos fonctions de conversion de nombres romains, qui n ont pas encore été écrite mais 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.

Example 6.1. `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−4.1.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 testDecimal(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()
```

Pour en savoir plus

- La page 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 (http://pyunit.sourceforge.net/pyunit.html) 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.
- *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).
- The 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.

6.3. 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 nombres romains de tous les entiers entre `1` et `3999`.

Example 6.2. `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/romahas passed this test.ins que j ai vérifié manuellement. Elle comprend les dix plus petits nombres, le plus grand nombre, chaque nombre représenté par un seul caractère en romain 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 échouéle.

❹ Ici, nous appelons la véritable fonction `toRoman`. (Et bien, la fonction n a pas encore été écrite, mais quand elle le sera, c est cette ligne qui l appelera.) 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 (le nombre romain). 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 de `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.

6.4. 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 indique 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.

Example 6.3. 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 testDecimal(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 nombres 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 parcequ une valeur est retournée, soit parcequ 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 lieur de `toRoman`:

4. `fromRoman` doit prendre un nombre romain 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.

Example 6.4. 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` plus loin dans ce chapitre.

6.5. 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 nombre romain, 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`.

Example 6.5. 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 nombre romain (`numeral`), puis on converti ce nombre romain 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 un nombre romain en majuscules.
8. `fromRoman` doit seulement accepter des nombres 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.

Example 6.6. 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())
```

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`. [12] 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 ? Parceque 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 ilôt.

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.

6.6. `roman.py`, étape 1

Maintenant que notre test unitaire est complet, 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`.

Example 6.7. `roman1.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−4.1.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 étéape, 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 :

Example 6.8. 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 testDecimal
    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` ("`toRoman` 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.

6.7. `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.

Example 6.9. `roman2.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−4.1.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 nombres romains les plus élémentaires. Notez qu il ne s agit pas seulement des nombres 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 nombres romains. Ils sont listés par ordre décroissant de valeur, de `M` jusqu à `I`.
3. La valeur de chaque nombre romain. Chaque tuple est une paire de `(`*`nombre, 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 nombre romain, 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 nombres romains à la fin de la sortie, soustrayondefines the Roman numeral representations as uppercase. So this test passes alreadys la valeur de l entrée et répétons l opération.

Example 6.10. Comment fonctionne `toRoman`

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.

Example 6.11. Sortie de `romantest2.py` avec `roman2.py`

```
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 testDecimal
    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)
```

6.8. `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).

Example 6.12. `roman3.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−4.1.zip) du livre.

Si vous ne l avez pas déjà fait, vous pouvez télécharger cet exemple ainsi que les autres exemples (http://diveintopython.org/download/diveintopython−examples−4.1.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, "decimals 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 intégrée ou une exception que vous avez défini. 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 nombres romains.

❹ Le reste de la fonction est inchangé.

Example 6.13. 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, "decimals can not be converted"
NotIntegerError: decimals can not be converted
```

Example 6.14. 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 passe 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 arréter d écrire le code de `toRoman` immédiatement. Pas de réglages, pas de bidouilles et pas de vérification supplémentaires "au cas où". Arrétez. Maintenant. Ecartez vous du clavier.

La chose la plus importante que des tests unitaires complets vous disent est quand vous arréter d écrire du code. Quand tous les tests unitaires d une fonction passent, arrétez d écrire le code de la fonction. Quand tous les tests d un module passent, arrétez d écrire le code du module.

6.9. `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`.

Example 6.15. `roman4.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−4.1.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 nombres romains (un tuple de tuples), et au lieu de chercher la plus grande valeur possible, nous cherchons la chaîne représentant le nombre romaine le plus "haut" possible.

Example 6.16. Comment fonctionne `fromRoman`

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, ', adding', integer

>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , adding 1000
found CM , adding 900
found L , adding 50
found X , adding 10
found X , adding 10
found I , adding 1
found I , adding 1
1972
```

Example 6.17. Sortie de `romantest4.py` avec `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 nombres romains érroné pour certaines entrées, *et* que `fromRoman` ait un bogue correspondant produisant les mêmes valeurs entières érronées pour les mêmes nombres 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)
```

6.10. `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 romain 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 Section 4.9, Introduction aux expressions régulières , il est sans doute temps de le faire.

Comme nous l avons vu au début de ce chapitre, il y a plusieurs règles simples pour construire un nombre romain. La première est que les milliers sont représentés par une série de `M`.

Example 6.18. 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 :

1. `^` − 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.
2. `M?` − reconnaît un `M` optionnel. Comme nous le répétons trois fois, nous reconnaissons 0 à 3 `M` se suivant.
3. `$` − 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. Nous n allons pas entrer dans les détails de l objet que `search` retourne (bien que ce soit très intéressant), parce que 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. Gardez ce fait en mémoire, cela sera important dans la prochaîne section.

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 :

1. `CM`
2. `CD`
3. 0 à 3 `C`(0 si les centaines valent 0)
4. `D`, suivi de 0 à 3 `C`

Les deux derniers motifs peuvent être combinés en :

- un `D` optionnel, suivi de 0 à 3 `C`

Example 6.19. 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 deuxièmes et troisièmes `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 deuxièmes et troisièmes `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. (Plus loin dans ce chapitre, vous verrez une syntaxe légèrement différente pour les expressions régulières. Elle est tout aussi complexe, mais permet au moins un peu de documentation des différentes sections de l expression.) Heureusement, si vous avez suivi jusque là, les dizaines sont relativement simples puisqu elles suivent exactement le même motif.

Example 6.20. `roman5.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−4.1.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, "decimals 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 qui gérait les milliers et les centaines. 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 :

Example 6.21. 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.

6.11. Prise en charge 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.

Example 6.22. 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.

Example 6.23. 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 connait, 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 :

Example 6.24. 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)
```

Now we can fix the bug.

Example 6.25. Fixing the bug (`roman62.py`)

```
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 nécéssaires : une vérification explicite de chaîne non nulle et une instruction `raise`.

Example 6.26. 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) nécéssitent 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 problable 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.

6.12. Prise en charge des changements de spécifications

Malgrés 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 nombres 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.

Example 6.27. Modification des cas de test pour prendre en charge de nouvelles spécifications (`romantest71.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−4.1.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 testDecimal(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 raisonnables à 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` to `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.

Example 6.28. 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.)

Example 6.29. Ecrire le code des nouvelles spécifications (`roman72.py`)

```
"""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, "decimals 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 complètement 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 nécéssaire. Vous n avez pas à me croire sur parole, voyez par vous−même :

Example 6.30. 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."

6.13. 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 passe, 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 diBoth mean the same thing: sque" ou simplement "de manière plus élégante". Quoi que cela signifie pour vous, pour votre projet, dans votre environnement, la refactorisation est importante à 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.

Example 6.31. 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 intérrogé 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 reconnaîssance 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, parmis 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.Both mean the same thing:

Example 6.32. Expressions régulières compilées dans `roman81.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−4.1.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 :

Example 6.33. 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.

Example 6.34. `roman82.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−4.1.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 ?

Example 6.35. 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 réécrivant 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 réécrire 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. La documentation est critique et Python fournit une manière de documenté vos expressions régulières de manière détaillée.

Example 6.36. `roman83.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−4.1.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 interieur 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.

Example 6.37. 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 repenchera sur ce module dans six mois aura une chance de comprendre le

fonctionnement de la fonction.

6.14. 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 nécéssaire 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.

Example 6.38. `roman9.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−4.1.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, "decimals 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 ?

Example 6.39. 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 seconds. 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 import n est fait qu une seulle fois, cela est négligeable au bout du compte.

La morale de l histoire ?

- La simplicité est une vertu.
- Particulièremment 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.

6.15. 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 test unitaire pour ne 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 connue
- Utiliser `assertRaises` pour vérifier qu une fonction déclenche une exception connue
- 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.

---

[12] "Je peux résister à tout, sauf à la tentation." −−Oscar Wilde

Chapter 7. Data−Centric Programming

7.1. Diving in

In Chapter 6, *Tests unitaires*, we discussed the philosophy of unit testing and stepped through the implementation of it in Python. This chapter will focus more on advanced Python−specific techniques, centered around the `unittest` module. If you haven t read Chapter 6, *Tests unitaires*, you ll get lost about halfway through this chapter. You have been warned.

The following is a complete Python program that acts as a cheap and simple regression testing framework. It takes unit tests that you ve written for individual modules, collects them all into one big test suite, and runs them all at once. I actually use this script as part of the build process for this book; I have unit tests for several of the example programs (not just the `roman.py` module featured in Chapter 6, *Tests unitaires*), and the first thing my automated build script does is run this program to make sure all my examples still work. If this regression test fails, the build immediately stops. I don t want to release non−working examples any more than you want to download them and sit around scratching your head and yelling at your monitor and wondering why they don t work.

Example 7.1. `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−4.1.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")
```

Running this script in the same directory as the rest of the example scripts that come with this book will find all the unit tests, named *`module`*`test.py`, run them as a single test, and pass or fail them all at once.

Example 7.2. Sample output of `regression.py`

```
[f8dy@oliver 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 The first 5 tests are from `apihelpertest.py`, which tests the example script from Chapter 2, *Le pouvoir de l introspection*.

❷ The next 5 tests are from `odbchelpertest.py`, which tests the example script from Chapter 1, *Faire connaissance de Python*.

❸ The rest are from `romantest.py`, which we studied in depth in Chapter 6, *Tests unitaires*.

7.2. Finding the path

When running Python scripts from the command line, it is sometimes useful to know where the currently running script is located on disk.

This is one of those obscure little tricks that is virtually impossible to figure out on your own, but simple to remember once you see it. The key to it is `sys.argv`. As we saw in Chapter 5, *XML Processing*, this is a list that holds the list of command−line arguments. However, it also holds the name of the running script, exactly as it was called from the command line, and this is enough information to determine its location.

Example 7.3. `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−4.1.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 Regardless of how you run a script, `sys.argv[0]` will always contain the name of the script, exactly as it appears on the command line. This may or may not include any path information, as we ll see shortly.

❷ `os.path.dirname` takes a filename as a string and returns the directory path portion. If the given filename does not include any path information, `os.path.dirname` returns an empty string.

❸ `os.path.abspath` is the key here. It takes a pathname, which can be partial or even blank, and returns a fully qualified pathname.

`os.path.abspath` deserves further explanation. It is very flexible; it can take any kind of pathname.

Example 7.4. Further explanation of `os.path.abspath`

```
>>> import os
>>> os.getcwd()                          ❶
/home/f8dy
>>> os.path.abspath('')                  ❷
/home/f8dy
>>> os.path.abspath('.ssh')              ❸
/home/f8dy/.ssh
>>> os.path.abspath('/home/f8dy/.ssh')   ❹
/home/f8dy/.ssh
>>> os.path.abspath('.ssh/../foo/')      ❺
/home/f8dy/foo
```

❶ `os.getcwd()` returns the current working directory.

❷ Calling `os.path.abspath` with an empty string returns the current working directory, same as `os.getcwd()`.

❸ Calling `os.path.abspath` with a partial pathname constructs a fully qualified pathname out of it, based on the current working directory.

❹ Calling `os.path.abspath` with a full pathname simply returns it.

❺ `os.path.abspath` also *normalizes* the pathname it returns. Note that this example worked even though I don t actually have a 'foo' directory. `os.path.abspath` never checks your actual disk; this is all just string manipulation.

The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

`os.path.abspath` not only constructs full path names, it also normalizes them. If you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin'` will return `/usr/local/bin`. If you just want to normalize a pathname without turning it into a full pathname, use `os.path.normpath` instead.

Example 7.5. Sample output from `fullpath.py`

```
[f8dy@oliver py]$ python /home/f8dy/diveintopython/common/py/fullpath.py ❶
sys.argv[0] = /home/f8dy/diveintopython/common/py/fullpath.py
path = /home/f8dy/diveintopython/common/py
full path = /home/f8dy/diveintopython/common/py
[f8dy@oliver diveintopython]$ python common/py/fullpath.py               ❷
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/f8dy/diveintopython/common/py
[f8dy@oliver diveintopython]$ cd common/py
[f8dy@oliver py]$ python fullpath.py                                     ❸
```

```
sys.argv[0] = fullpath.py
path =
full path = /home/f8dy/diveintopython/common/py
```

❶ In the first case, `sys.argv[0]` includes the full path of the script. We can then use the `os.path.dirname` function to strip off the script name and return the full directory name, and `os.path.abspath` simply returns what we give it.

❷ If the script is run by using a partial pathname, `sys.argv[0]` will still contain exactly what appears on the command line. `os.path.dirname` will then give us a partial pathname (relative to the current directory), and `os.path.abspath` will construct a full pathname from the partial pathname.

❸ If the script is run from the current directory without giving any path, `os.path.dirname` will simply return an empty string. Given an empty string, `os.path.abspath` returns the current directory, which is what we want, since the script was run from the current directory.

Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross−platform. Your results will look slightly different than my examples if you re running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they ll still work. That s the whole point of the `os` module.

Addendum. One reader was dissatisfied with this solution, and wanted to be able to run all the unit tests in the current directory, not the directory where `regression.py` is located. He suggests this approach instead:

Example 7.6. Running scripts in the current directory

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       ❶
    sys.path.append(path)    ❷
    files = os.listdir(path) ❸
```

❶ Instead of setting `path` to the directory where the currently running script is located, we set it to the current working directory instead. This will be whatever directory you were in before you ran the script, which is not necessarily the same as the directory the script is in. (Read that sentence a few times until you get it.)

❷ Append this directory to the Python library search path, so that when we dynamically import the unit test modules later, Python can find them. We didn t have to do this when `path` was the directory of the currently running script, because Python always looks in that directory.

❸ The rest of the function is the same.

This technique will allow you to re−use this `regression.py` script on multiple projects. Just put the script in a common directory, then change to the project s directory before running it. All of that project s unit tests will be found and tested, instead of the unit tests in the common directory where `regression.py` is located.

7.3. Filtering lists revisited

You re already familiar with using list comprehensions to filter lists. There is another way to accomplish this same thing, which some people feel is more expressive.

Python has a built−in `filter` function which takes two arguments, a function and a list, and returns a list.[13] The function passed as the first argument to `filter` must itself take one argument, and the list that `filter` returns will contain all the elements from the list passed to `filter` for which the function passed to `filter` returns true.

Got all that? It s not as difficult as it sounds.

Example 7.7. Introducting `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]
>>> filteredList = []
>>> for n in li:                   ❸
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

❶ `odd` uses the built-in mod function "%" to return `1` if `n` is odd and `0` if `n` is even.

❷ `filter` takes two arguments, a function (`odd`) and a list (`li`). It loops through the list and calls `odd` with each element. If `odd` returns a true value (remember, any non-zero value is true in Python), then the element is included in the returned list, otherwise it is filtered out. The result is a list of only the odd numbers from the original list, in the same order as they appeared in the original.

❸ You could accomplish the same thing with a `for` loop. Depending on your programming background, this may seem more "straightforward", but functions like `filter` are much more expressive. Not only is it easier to write, it s easier to read, too. Reading the `for` loop is like standing too close to a painting; you see all the details, but it may take a few seconds to be able to step back and see the bigger picture: "Oh, we re just filtering the list!"

Example 7.8. `filter` in `regression.py`

```
files = os.listdir(path)                    ❶
test = re.compile("test.py$", re.IGNORECASE) ❷
files = filter(test.search, files)          ❸
```

❶ As we saw in Section 7.2, Finding the path , `path` may contain the full or partial pathname of the directory of the currently running script, or it may contain an empty string if the script is being run from the current directory. Either way, `files` will end up with the names of the files in the same directory as this script we re running.

❷ This is a compiled regular expression. As we saw in Section 6.13, Refactorisation , if you re going to use the same regular expression over and over, you should compile it for faster performance. The compiled object has a `search` method which takes a single argument, the string the search. If the regular expression matches the string, the `search` method returns a `Match` object containing information about the regular expression match; otherwise it returns `None`, the Python null value.

❸ For each element in the `files` list, we re going to call the `search` method of the compiled regular expression object, `test`. If the regular expression matches, the method will return a `Match` object, which Python considers to be true, so the element will be included in the list returned by `filter`. If the regular expression does not match, the `search` method will return `None`, which Python considers to be false, so the element will not be included.

Historical note. Versions of Python prior to 2.0 did not have list comprehensions, so you couldn t filter using list comprehensions; the `filter` function was the only game in town. Even with the introduction of list comprehensions in 2.0, some people still prefer the old-style `filter` (and its companion function, `map`, which we ll see later in this chapter). Both techniques work, and neither is going away, so which one you use is a matter of style.

Example 7.9. Filtering using list comprehensions instead

```
    files = os.listdir(path)
    test = re.compile("test.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] ❶
```

❶ This will accomplish exactly the same result as using the `filter` function. Which way is more expressive? That s up to you.

7.4. Mapping lists revisited

You re already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built−in `map` function. It works much the same way as the `filter` function.

Example 7.10. Introducing `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` takes a function and a list[14] and returns a new list by calling the function with each element of the list in order. In this case, the function simply multiplies each element by 2.

❷ You could accomplish the same thing with a list comprehension. List comprehensions were first introduced in Python 2.0; `map` has been around forever.

❸ You could, if you insist on thinking like a Visual Basic programmer, use a `for` loop to accomplish the same thing.

Example 7.11. `map` with lists of mixed datatypes

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                          ❶
[10, 'aa', (2, 'b', 2, 'b')]
```

❶ As a side note, I d like to point out that `map` works just as well with lists of mixed datatypes, as long as the function you re using correctly handles each type. In this case, our `double` function simply multiples the given argument by 2, and Python Does The Right Thing depending on the datatype of the argument. For integers, this means actually multiplying it by 2; for strings, it means concatenating the string with itself; for tuples, it means making a new tuple that has all of the elements of the original, then all of the elements of the original again.

All right, enough play time. Let s look at some real code.

Example 7.12. `map` in `regression.py`

```
    filenameToModuleName = lambda f: os.path.splitext(f)[0] ❶
    moduleNames = map(filenameToModuleName, files)          ❷
```

❶ As we saw in Section 2.7, Utiliser des fonctions lambda , `lambda` defines an inline function. And as we saw in Example 3.36, Division de noms de chemins , `os.path.splitext` takes a filename and returns a tuple (*`name, extension`*). So `filenameToModuleName` is a function which will take a filename and strip off the file extension, and return just the name.

❷ Calling `map` takes each filename listed in `files`, passes it to our function `filenameToModuleName`, and returns a list of the return values of each of those function calls. In other words, we strip the file extension off of each filename, and store the list of all those stripped filenames in `moduleNames`.

As we ll see in the rest of the chapter, we can extend this type of data−centric thinking all the way to our final goal, which is to define and execute a single test suite that contains the tests from all of those individual test suites.

7.5. Data−centric programming

By now you re probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that s a perfectly valid question. Mostly, it s a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.

In this case, we started with no data at all; the first thing we did was get the directory path of the current script, and got a list of files in that directory. That was our bootstrap, and it gave us real data to work with: a list of filenames.

However, we knew we didn t care about all of those files, only the ones that were actually test suites. We had *too much data*, so we needed to `filter` it. How did we know which data to keep? We needed a test to decide, so we defined one and passed it to the `filter` function. In this case we used a regular expression to decide, but the concept would be the same regardless of how we constructed the test.

Now we had the filenames of each of the test suites (and only the test suites, since everything else had been filtered out), but we really wanted module names instead. We had the right amount of data, but it was *in the wrong format*. So we defined a function that would transform a single filename into a module name, and we mapped that function onto the entire list. From one filename, we can get a module name; from a list of filenames, we can get a list of module names.

Instead of `filter`, we could have used a `for` loop with an `if` statement. Instead of `map`, we could have used a `for` loop with a function call. But using `for` loops like that is busywork. At best, it simply wastes time; at worst, it introduces obscure bugs. For instance, we have to figure out how to test for the condition "is this file a test suite?" anyway; that s our application−specific logic, and no language can write that for us. But once we ve figured that out, do we really want go to all the trouble of defining a new empty list and writing a `for` loop and an `if` statement and manually calling `append` to add each element to the new list if it passes the condition and then keeping track of which variable holds the new filtered data and which one holds the old unfiltered data? Why not just define the test condition, then let Python do the rest of that work for us?

Oh sure, you could try to be fancy and delete elements in place without creating a new list. But you ve been burned by that before. Trying to modify a data structure that you re looping through can be tricky. You delete an element, then loop to the next element, and suddenly you ve skipped one. Is Python one of the languages that works that way? How long would it take you to figure it out? Would you remember for certain whether it was safe the next time you tried? Programmers spend so much time and make so many mistakes dealing with purely technical issues like this, and it s all pointless. It doesn t advance your program at all; it s just busywork.

I resisted list comprehensions when I first learned Python, and I resisted `filter` and `map` even longer. I insisted on making my life more difficult, sticking to the familiar way of `for` loops and `if` statements and step−by−step code−centric programming. And my Python programs looked a lot like Visual Basic programs, detailing every step of every operation in every function. And they had all the same types of little problems and obscure bugs. And it was all pointless.

Let it all go. Busywork code is not important. Data is important. And data is not difficult. It s only data. If you have too much, filter it. If it s not what you want, map it. Focus on the data; leave the busywork behind.

7.6. Dynamically importing modules

Sorry, you ve reached the end of the chapter that s been written so far. Please check back at http://diveintopython.org/ for updates.

---

[13] Technically, the second argument to `filter` can be any sequence, including lists, tuples, and custom classes that act like lists by defining the `__getitem__` special method. If possible, `filter` will return the same datatype as you give it, so filtering a list returns a list, but filtering a tuple returns a tuple.

[14] Again, I should point out that `map` can take a list, a tuple, or any object that acts like a sequence. See previous footnote about `filter`.

Appendix A. Pour en savoir plus

Chapter 1. Faire connaissance de Python

- 1.3. Documentation des fonctions
    - *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) explique la manière d écrire de bonnes `doc string`.
    - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite des conventions d espacements dans les `doc string`s (http://www.python.org/doc/current/tut/node6.html#SECTION006750000000000000000).
- 1.4. Tout est objet
    - *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique exactectement ce que signifie l affirmation 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).
- 1.5. Indentation du code
    - *Python Reference Manual* (http://www.python.org/doc/current/ref/) discute des aspects multi−plateformes de l indentation et présente diverses erreurs d indentation (http://www.python.org/doc/current/ref/indentation.html).
    - *Python Style Guide* (http://www.python.org/doc/essays/styleguide.html) discute du bon usage de l indentation.
- 1.6. Test des modules
    - *Python Reference Manual* (http://www.python.org/doc/current/ref/) explique les détails techniques de l import de modules (http://www.python.org/doc/current/ref/import.html).
- 1.7. Présentation des 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).
    - 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).
    - 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).
    - *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).
- 1.8. Présentation des listes
    - *How to Think Like a Computer Scientist* (http://www.ibiblio.org/obp/thinkCSpy/) enseigne les listes et explique le sujet important du passage de listes comme arguments de fonction (http://www.ibiblio.org/obp/thinkCSpy/chap08.htm).
    - *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).

- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions fréquentes à propos des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/534) et fourni de nombreuxexemples de code utilisant des listes (http://www.faqts.com/knowledge−base/index.phtml/fid/540).
- *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).

• 1.9. 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).
- 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).
- *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).

• 1.10. Définitions de variables

- *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).

• 1.11. Assignation simultanée de plusieurs valeurs

- *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/chap10.htm).

• 1.12. Formatage de chaînes

- *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).

• 1.13. Mutation de listes

- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d une autre manière de transformer des listes avec la fonction intégrée `map` (http://www.python.org/doc/current/tut/node7.html#SECTION007130000000000000000).
- *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).

• 1.14. Jointure de listes et découpage de chaînes

- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions fréquentes 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).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les méthodes de chaînes (http://www.python.org/doc/current/lib/string−methods.html).
- *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.

Chapter 2. Le pouvoir de l introspection

- 2.2. Arguments optionnels et nommés
  - *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.
- 2.3. type, str, dir et autres fonctions intégrées
  - *Python Library Reference* (http://www.python.org/doc/current/lib/) documente toutes les fonctions intégrées (http://www.python.org/doc/current/lib/built-in-funcs.html) et toutes les exceptions intégrées (http://www.python.org/doc/current/lib/module-exceptions.html).
- 2.5. Filtrage de listes
  - *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) traite d une autre manière de filtrer les listes en utilisant la fonction intégrée `filter` (http://www.python.org/doc/current/tut/node7.html#SECTION00713000000000000000).
- 2.6. Particularités de and et or
  - 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).
- 2.7. Utiliser des fonctions lambda
  - 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).
  - *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). (PEP 227 (http://python.sourceforge.net/peps/pep-0227.html) explique comment cela va changer dans les futures versions de Python.)
  - *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).

Chapter 3. Un framework orienté objet

- 3.2. 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).
  - *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#SECTION00841000000000000000).
- 3.3. Définition de classes

- *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/chap11.htm).
- *Python Tutorial* (http://www.python.org/doc/current/tut/tut.html) a une aperçu en profondeur des classes, des espaces de noms et de l héritage (http://www.python.org/doc/current/tut/node11.html).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond à des questions fréquentes à propos des classes (http://www.faqts.com/knowledge−base/index.phtml/fid/242).

• 3.4. Instantiation de classes

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

• 3.5. UserDict : une classe enveloppe

- *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) (http://www.python.org/doc/current/lib/module−copy.html).

• 3.7. Méthodes spéciales avancées

- *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).

• 3.9. Fonctions privées

- *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).

• 3.10. Traitement des exceptions

- *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).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) résume toutes les exceptions intégrées (http://www.python.org/doc/current/lib/module−exceptions.html).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) documente le module getpass (http://www.python.org/doc/current/lib/module−getpass.html).
- *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.
- *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).

• 3.11. Les objets fichier

- *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#SECTION00921000000000000000).

- eff−bot (http://www.effbot.org/guides/) traite de l éfficience et de la performance de différentes manières de lire un fichier (http://www.effbot.org/guides/readline−performance.htm).
- Python Knowledge Base (http://www.faqts.com/knowledge−base/index.phtml/fid/199/) répond aux questions fréquentes à propose des fichiers (http://www.faqts.com/knowledge−base/index.phtml/fid/552).
- *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).

• 3.13. Complément sur les modules

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

• 3.14. Le module os

- 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).
- *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).

Chapter 4. Traitement du HTML

• 4.4. Présentation de BaseHTMLProcessor.py

- W3C (http://www.w3.org/) traite des références de caractères et d entités (http://www.w3.org/TR/REC−html40/charset.html#entities).
- *Python Library Reference* (http://www.python.org/doc/current/lib/) confirme vos soupçons selon lesquels le module `htmlentitydefs` (http://www.python.org/doc/current/lib/module−htmlentitydefs.html) est exactement ce que son nom laisse deviner.

• 4.9. Introduction aux expressions régulières

- La Regular Expression HOWTO (http://py−howto.sourceforge.net/regex/regex.html) présente les expressions régulières et explique comment les utiser 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).

• 4.10. Assembler les pièces

- Vous coyiez que je plaisantait 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/). Je ne sais pas du tout si il est implémenté en Python, mais la page d accueil de ma société est à hurler de rire en *Pig Latin*. Malheureusement, le code source n a pas l air d être disponible.

Chapter 5. XML Processing

• 5.4. Unicode

- Unicode.org (http://www.unicode.org/) is the home page of the unicode standard, including a brief technical introduction (http://www.unicode.org/standard/principles.html).
- Unicode Tutorial (http://www.reportlab.com/i18n/python_unicode_tutorial.html) has some more

examples of how to use Python s unicode functions, including how to force Python to coerce unicode into ASCII even when it doesn t really want to.
- Unicode Proposal (http://www.lemburg.com/files/python/unicode−proposal.txt) is the original technical specification for Python s unicode functionality. For advanced unicode hackers only.

Chapter 6. Tests unitaires

- 6.1. Plonger
  - Ce site (http://www.deadline.demon.co.uk/roman/front.htm) a plus d information sur les nombres romains, y compris une histoire (http://www.deadline.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).
- 6.2. Présentation de romantest.py
  - La page 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 (http://pyunit.sourceforge.net/pyunit.html) 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.
  - *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).
  - The 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.
- 6.15. 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.

Chapter 7. Data−Centric Programming

Appendix B. Survol en cinq minutes

Chapter 1. Faire connaissance de Python

- 1.1. Plonger

  Voici un programme Python complet et fonctionnel.
- 1.2. Déclaration de fonctions

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

  Vous pouvez documenter une fonction Python en lui donnant une chaîne de documentation (`doc string`).
- 1.4. Tout est objet

  Une fonction, comme tout le reste en Python, est un objet.
- 1.5. 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.
- 1.6. 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 facilement tester vos modules lorsque vous les écrivez.
- 1.7. 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.
- 1.8. 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.
- 1.9. Présentation des tuples

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

  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.
- 1.11. Assignation simultanée de plusieurs valeurs

  Un des raccourcis les plus chouettes existant en Python est l utilisation de séquences pour assigner plusieurs valeurs en une fois.
- 1.12. Formatage de chaînes

- 2.7. 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 ou une fonction est nécéssaire.
- 2.8. 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.
- 2.9. Résumé

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

Chapter 3. Un framework orienté objet

- 3.1. Plonger

  Voici un programme Python complet et fonctionnel. Lisez les `doc string`s 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, c est à vous l expliquer que sert la suite du chapitre.
- 3.2. Importation de modules avec from module import

  Python fournit deux manières d importer les modules. Les deux sont utiles, et vous devez savoir quand utiliser laquelle. Vous avez déjà vu la première, `import *module*`, au chapitre 1. La deuxième manière accomplit la même action mais a des différences subtiles mais importante dans son fonctionnement.
- 3.3. Définition de classes

  Python est entièrement orienté objet : vous pouvez définir vos propres classes, hériter de vos classes ou des classes intégrées et instancier les classes que vous avez défini.
- 3.4. 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 seral l objet nouvellement créé.
- 3.5. 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.
- 3.6. 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.
- 3.7. 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.

- 3.8. 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.
- 3.9. Fonctions privées

  Comme la plupart des langages, Python possède le concept de fonctions privées, qui ne peuvent être appelées de l extérieur de leur module, de méthodes de classe privées, qui ne peuvent être appelées de l extérieur de leur classe et d 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.
- 3.10. Traitement des exceptions

  Comme beaucoup de langages orientés objet, Python gère les exception à l aide de blocs `try...except`.
- 3.11. Les objets fichier

  Python a une fonction intégrée, `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.
- 3.12. 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.
- 3.13. Complément sur les modules

  Les modules, comme tout le reste en Python, sont des objets. Une fois importés, vous pouvez toujours obtenir une référence à un module à travers le dictionnaire global `sys.modules`.
- 3.14. Le module os

  Le module `os` a de nombreuses fonctions utiles pour manipuler les fichiers et les processus. `os.path` a des fonctions pour manipuler les chemins de fichiers et de répertoires.
- 3.15. 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.
- 3.16. Résumé

  Le programme `fileinfo.py` devrait maintenant être parfaitement clair.

Chapter 4. Traitement du HTML

- 4.1. 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.

- 4.2. 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.
- 4.3. 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.
- 4.4. 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 utilse, 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 envoit et reconstruit entièrement le document HTML. En termes techniques, cette classe sera un *producteur* de HTML.
- 4.5. locals et globals

  Python a deux fonctions intégrées permettant d accéder aux variables locales et globales sous forme de dictionnaire : `locals` et `globals`.
- 4.6. Formatage de chaînes à l aide d un dictionnaire

  Il existe une technique de formatage de chaînes alternative utilisant un dictionnaire au lieu de valeurs stockées dans un tuple.
- 4.7. 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 attibuts sans guillemets et je veux les mettre entre guillemets. Comment faire ?"[9] (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`.
- 4.8. 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.
- 4.9. Introduction aux expressions régulières

  Les expressions régulières sont un moyen puissant (et relativement 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`

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.

- 6.3. 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.

- 6.4. 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.

- 6.5. 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.

- 6.6. roman.py, étape 1

  Maintenant que notre test unitaire est complet, 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`.

- 6.7. 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.

- 6.8. 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).

- 6.9. 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`.

- 6.10. 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 romain 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.

- 6.11. Prise en charge 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.

- 6.12. Prise en charge des changements de spécifications

accomplish this same thing, which some people feel is more expressive.

- 7.4. Mapping lists revisited

  You re already familiar with using list comprehensions to map one list into another. There is another way to accomplish the same thing, using the built−in `map` function. It works much the same way as the `filter` function.

- 7.5. Data−centric programming

  By now you re probably scratching your head wondering why this is better than using `for` loops and straight function calls. And that s a perfectly valid question. Mostly, it s a matter of perspective. Using `map` and `filter` forces you to center your thinking around your data.

- 7.6. Dynamically importing modules

  Sorry, you ve reached the end of the chapter that s been written so far. Please check back at http://diveintopython.org/ for updates.

Appendix C. Trucs et astuces

Chapter 1. Faire connaissance de Python

- 1.1. Plonger

  Dans l IDE Python sous Windows, vous pouvez exécuter un module avec 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 module depuis la ligne de commande : **`python odbchelper.py`**
- 1.2. 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.
- 1.3. 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`s pour fournir une documentation contextuelle, ainsi lorsque vous tapez le nom d une fonction, sa `doc string` apparaît dans une bulle d aide. Ce peut être incroyablement utile, mais cette utilité est liée à la qualité de votre `doc string`.
- 1.4. Tout est objet

  L `import` de Python est similaire au `require` de 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`*.
- 1.5. Indentation du code

  Python utilise le retour à la ligne pour séparer les instructions et deux points ainsi que l indentation pour séparer les blocs de code. C++ et Java utilisent le point virgule pour séparer les instructions et les accolades pour séparer les blocs de code.
- 1.6. Test des modules

  A l instar de C, Python utilise `==` pour la comparaison et `=` pour l assignation. Mais, au contraire de C, Python ne supporte pas les assignations simultanées 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.
- 1.7. 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.

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). Il y a des façons de le faire, seulement elle ne sont pas intégrées dans le dictionnaire.

- 1.8. 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 `Vector`, qui peut contenir n importe quels objets et qui croît dynamiquement au fur et à mesure que de nouveaux éléments y sont ajoutés.

  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 2.2.1 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.
- 1.9. Présentation des tuples

  Les tuples peuvent être convertis en listes et vice−versa. La fonction intégrée `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.
- 1.10. Définitions de variables

  Lorsq 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.
- 1.12. Formatage de chaînes

  Le formatage de chaîne en Python utilise la même syntaxe que la fonction C `sprintf`.
- 1.14. 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(séparateur, 1)` est une technique pratique lorsque voulez rechercher une sous−chaîne dans une chaîne et traiter tous ce qui se situe avant (soit le premier élément de la liste) et après (le dernier élément de la liste) l occurrence de cette sous−chaîne.

Chapter 2. Le pouvoir de l introspection

- 2.2. 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.
- 2.3. type, str, dir et autres fonctions intégrées

  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, que Dieu vous aide, MSDN) pour vous rappeler l usage de ces modules, Python est en grande partie auto−documenté.
- 2.6. Particularités de and et or

  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.
- 2.7. Utiliser des fonctions lambda

  Les fonctions `lambda` sont une question de style. Les utiliser n est jamais une nécéssité, 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 ligne.
- 2.8. 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.

Chapter 3. Un framework orienté objet

- 3.2. Importation de modules avec from module import

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

  `from` *`module`* `import *` en Python est comme `import` *`module`*`.*` eb Java; `import` *`module`* en Python est comme `import` *`module`* en Java.
- 3.3. 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.

  Nous ne l étudierons pas en détail dans ce livre, mais 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, séparée par des virgules.

  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 noms. Cependant, veuillez ne pas l appeler autre chose que `self`, c est une très forte convention.

  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.

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.

- 3.4. 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.

- 3.5. UserDict : une classe enveloppe

  Dans l IDE Python 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 normal methodsancêtre, 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++ : 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).

- 3.6. Méthodes de classe spéciales

  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`*.

- 3.7. 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").

- 3.8. 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__`.

- 3.9. Fonctions privées

  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.

  En Python, toutes les méthodes spéciales (comme `__setitem__`) et les attributs intégrés (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 apporterais que de la confusion pour vous et les autres.

  Python n a pas de notion de méthodes de classe protégées (accessibles uniquement par leur propre classe et les descendants de celle−ci). Les méthodes de classes sont ou privées (accessibles seulement par leur propre classe) ou publiques (accessible de partout).

- 3.10. 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.

- 3.14. Le module os

  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.

Chapter 4. Traitement du HTML

- 4.2. 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 Python 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.

- 4.4. 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 exêcution 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.

- 4.5. locals et globals

  Python 2.2 a introduit un modification légère mais importante qui affecte l ordre de recherche dans les espaces de noms : les portées imbriquées. Dans les versions précédente de Python, lorsque vous référenciez une variable dans une fonction imbriquée ou une fonction `lambda`, Python recherche 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 les deux 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.

- 4.6. 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 dûe à l appel de `locals`, puisque `locals` effectue une copie de l espace de noms local.

Chapter 5. XML Processing

- 5.2. Packages

  A package is a directory with the special `__init__.py` file in it. The `__init__.py` file defines the attributes and methods of the package. It doesn t have to define anything; it can just be an empty file, but it has to exist. But if `__init__.py` doesn t exist, the directory is just a directory, not a package, and it can t be imported or contain modules or nested packages.
- 5.6. Accessing element attributes

  This section may be a little confusing, because of some overlapping terminology. Elements in an XML document have attributes, and Python objects also have attributes. When we parse an XML document, we get a bunch of Python objects that represent all the pieces of the XML document, and some of these Python objects represent attributes of the XML elements. But the (Python) objects that represent the (XML) attributes also have (Python) attributes, which are used to access various parts of the (XML) attribute that the object represents. I told you it was confusing. I am open to suggestions on how to distinguish these more clearly.

  Like a dictionary, attributes of an XML element have no ordering. Attributes may *happen to be* listed in a certain order in the original XML document, and the `Attr` objects may *happen to be* listed in a certain order when the XML document is parsed into Python objects, but these orders are arbitrary and should carry no special meaning. You should always access individual attributes by name, like the keys of a dictionary.

Chapter 6. Tests unitaires

- 6.2. 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/).
- 6.8. roman.py, étape 3

  La chose la plus importante que des tests unitaires complets vous disent est quand vous arréter d écrire du code. Quand tous les tests unitaires d une fonction passent, arrétez d écrire le code de la fonction. Quand tous les tests d un module passent, arrétez d écrire le code du module.
- 6.10. roman.py, étape 5

  Quand tous vos tests passent, arrêtez d écrire du code.
- 6.13. 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.Both mean the same thing:

Chapter 7. Data−Centric Programming

- 7.2. Finding the path

The pathnames and filenames you pass to `os.path.abspath` do not need to exist.

`os.path.abspath` not only constructs full path names, it also normalizes them. If you are in the `/usr/` directory, `os.path.abspath('bin/../local/bin'` will return `/usr/local/bin`. If you just want to normalize a pathname without turning it into a full pathname, use `os.path.normpath` instead.

Like the other functions in the `os` and `os.path` modules, `os.path.abspath` is cross−platform. Your results will look slightly different than my examples if you re running on Windows (which uses backslash as a path separator) or Mac OS (which uses colons), but they ll still work. That s the whole point of the `os` module.

Appendix D. Liste des exemples

Chapter 1. Faire connaissance de Python

Appendix E. Historique des révisions

Revision History	
Revision 4.1	28 July 2002

- Added Section 5.9, Caching node lookups .
- Added Section 5.10, Finding direct children of a node .
- Added Section 5.11, Creating separate handlers by node type .
- Added Section 5.12, Handling command line arguments .
- Added Section 5.13, Putting it all together .
- Added Section 5.14, Summary .
- Fixed typo in Section 3.14, Le module os . It s `os.getcwd()`, not `os.path.getcwd()`. Thanks, Abhishek.
- Fixed typo in Section 1.14, 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 2.8, 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 5.4, Unicode , "anyway" to "anywhere". Thanks Frank.
- Fixed typo in Section 6.5, Tester la cohérence , doubled word "accept". Thanks Ralph.
- Fixed typo in Section 6.13, Refactorisation , `C?C?C?` matches 0 to 3 `C` characters, not 4. Thanks Ralph.
- Clarified and expanded explanation of implied slice indices in Example 1.18, Raccourci pour le découpage . Thanks Petr.
- Added historical note in Section 3.5, UserDict : une classe enveloppe now that Python 2.2 supports subclassing built−in datatypes directly.
- Added explanation of `update` dictionary method in Example 3.11, Definition de la classe UserDict . Thanks Petr.
- Clarified Python s lack of overloading in Section 3.5, UserDict : une classe enveloppe . Thanks Petr.
- Fixed typo in Example 4.8, 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 4.5, locals et globals now that Python 2.2 is out. Thanks Petr.
- Fixed typo in Example 4.14, 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 3.5, UserDict : une classe enveloppe . Thanks Petr.
- Added title to note on downloading `unittest` in Section 6.13, Refactorisation . Thanks Petr.
- Fixed typesetting problem in Example 7.6, Running scripts in the current directory ; tabs should have been spaces, and the line numbers were misaligned. Thanks Petr.
- Fixed capitalization typo in the tip on truth values in Section 1.8, Présentation des listes . It s `True` and `False`, not `true` and `false`. Thanks to everyone who pointed this out.
- Changed section titles of Section 1.7, Présentation des dictionnaires , Section 1.8, Présentation des listes , and Section 1.9, 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.

Revision 4.0−2	26 April 2002

- Fixed typo in Example 2.16, Présentation de and .
- Fixed typo in Example 1.6, Chemin de recherche pour import .
- Fixed Windows help file (missing table of contents due to base stylesheet changes).

Revision 4.0	19 April 2002

- Expanded Section 1.4, Tout est objet to include more about import search paths.
- Fixed typo in Example 1.16, Indices de liste négatifs . Thanks to Brian for the correction.
- Rewrote the tip on truth values in Section 1.8, Présentation des listes , now that Python has a separate boolean datatype.
- Fixed typo in Section 3.2, Importation de modules avec from module import when comparing syntax to Java. Thanks to Rick for the correction.
- Added note in Section 3.5, UserDict : une classe enveloppe about derived classes always overriding ancestor classes.
- Fixed typo in Example 3.19, Modification des attributs de classe . Thanks to Kevin for the correction.
- Added note in Section 3.10, Traitement des exceptions that you can define and raise your own exceptions. Thanks to Rony for the suggestion.
- Fixed typo in Example 4.16, Traitement de balises spécifiques . Thanks for Rick for the correction.
- Added note in Example 4.17, SGMLParser about what the return codes mean. Thanks to Howard for the suggestion.
- Added `str` function when creating `StringIO` instance in Example 5.30, openAnything . Thanks to Ganesan for the idea.
- Added link in Section 6.2, Présentation de romantest.py to explanation of why test cases belong in a separate file.
- Changed Section 7.2, Finding the path 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.

Revision 3.9	1 January 2002

- Added Section 5.4, Unicode .
- Added Section 5.5, Searching for elements .
- Added Section 5.6, Accessing element attributes .
- Added Section 5.7, Abstracting input sources .
- Added Section 5.8, Standard input, output, and error .
- Added simple counter `for` loop examples (good usage and bad usage) in Section 3.12, Boucles for . Thanks to Kevin for the idea.
- Fixed typo in Example 1.33, keys, values, et items (two elements of `params.values()` were reversed).
- Fixed mistake in Section 2.3, type, str, dir et autres fonctions intégrées with regards to the name of the `__builtin__` module. Thanks to Denis for the correction.
- Added additional example in Section 7.2, Finding the path 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 Example 1.16, 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.

Revision 3.8	18 November 2001

- Added Section 7.2, Finding the path .
- Added Section 7.3, Filtering lists revisited .
- Added Section 7.4, Mapping lists revisited .
- Added Section 7.5, Data−centric programming .
- Expanded sample output in Section 7.1, Diving in .
- Finished Section 5.3, Parsing XML .

Revision 3.7	30 September 2001

- Added Section 5.2, Packages .
- Added Section 5.3, Parsing XML .
- Cleaned up introductory paragraph in Section 5.1, Diving in . Thanks to Matt for this suggestion.
- Added Java tip in Section 3.2, Importation de modules avec from module import . Thanks to Ori for this suggestion.
- Fixed mistake in Section 2.8, 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 1.8, 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 1.8, 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 Chapter 1, *Faire connaissance de 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 1.2, 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/).

Revision 3.6.4	6 September 2001

- Added code in `BaseHTMLProcessor` to handle non−HTML entity references, and added a note about it in Section 4.4, Présentation de BaseHTMLProcessor.py .
- Modified Example 4.11, Présentation de globals to include `htmlentitydefs` in the output.

Revision 3.6.3	4 September 2001

- Fixed typo in Section 5.1, Diving in .
- Added link to Korean translation (http://diveintopython.org/kr/html/index.htm).

Revision 3.6.2	31 August 2001

- Fixed typo in Section 6.5, Tester la cohérence (the last requirement was listed twice).

Revision 3.6	31 August 2001

- Finished Chapter 4, *Traitement du HTML* with Section 4.10, Assembler les pièces and Section 4.11, Résumé .
- Added Section 6.14, Postscriptum .
- Started Chapter 5, *XML Processing* with Section 5.1, Diving in .
- Started Chapter 7, *Data−Centric Programming* with Section 7.1, Diving in .

- Fixed long−standing bug in colorizing script that improperly colorized the examples in Chapter 4, *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.

Revision 3.5	26 June 2001

- Added explanation of strong/weak/static/dynamic datatypes in Section 1.2, Déclaration de fonctions .
- Added case−sensitivity example in Section 1.7, Présentation des dictionnaires .
- Use `os.path.normcase` in Chapter 3, *Un framework orienté objet* to compensate for inferior operating systems whose files aren t case−sensitive.
- Fixed indentation problems in code samples in PDF version.

Revision 3.4	31 May 2001

- Added Section 6.10, roman.py, étape 5 .
- Added Section 6.11, Prise en charge des bogues .
- Added Section 6.12, Prise en charge des changements de spécifications .
- Added Section 6.13, Refactorisation .
- Added Section 6.15, Résumé .
- Fixed yet another stylesheet bug that was dropping nested `</span>` tags.

Revision 3.3	24 May 2001

- Added Section 6.1, Plonger .
- Added Section 6.2, Présentation de romantest.py .
- Added Section 6.3, Tester la réussite .
- Added Section 6.4, Tester l échec .
- Added Section 6.5, Tester la cohérence .
- Added Section 6.6, roman.py, étape 1 .
- Added Section 6.7, roman.py, étape 2 .
- Added Section 6.8, roman.py, étape 3 .
- Added Section 6.9, roman.py, étape 4 .
- Tweaked stylesheets in an endless quest for complete Netscape/Mozilla compatibility.

Revision 3.2	3 May 2001

- Added Section 4.8, Présentation de dialect.py .
- Added Section 4.9, Introduction aux expressions régulières .
- 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.

Revision 3.1	18 Apr 2001

- Added code in `BaseHTMLProcessor.py` to handle declarations, now that Python 2.1 supports them.
- Added note about nested scopes in Section 4.5, locals et globals .
- Fixed obscure bug in Example 4.1, 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.

Revision 3.0	16 Apr 2001

- Fixed minor bug in code listing in Chapter 4, *Traitement du HTML*.
- Added link to Chinese translation on home page (http://diveintopython.org/).

Revision 2.9	13 Apr 2001

- Added Section 4.5, locals et globals .
- Added Section 4.6, Formatage de chaînes à l aide d un dictionnaire .
- Tightened code in Chapter 4, *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 Appendix 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.

Revision 2.8	26 Mar 2001

- Added Section 4.3, Extraction de données de documents HTML .
- Added Section 4.4, Présentation de BaseHTMLProcessor.py .
- Added Section 4.7, Mettre les valeurs d attributs entre guillemets .
- Tightened up code in Chapter 2, *Le pouvoir de l introspection*, using the built−in function `callable` instead of manually checking types.
- Moved Section 3.2, Importation de modules avec from module import from Chapter 2, *Le pouvoir de l introspection* to Chapter 3, *Un framework orienté objet*.
- Fixed typo in code example in Section 3.1, Plonger (added colon).
- Added several additional downloadable example scripts.
- Added Windows Help output format.

Revision 2.7	16 Mar 2001

- Added Section 4.2, Présentation de sgmllib.py .
- Tightened up code in Chapter 4, *Traitement du HTML*.
- Changed code in Chapter 1, *Faire connaissance de Python* to use `items` method instead of `keys`.

- Moved Section 1.11, Assignation simultanée de plusieurs valeurs section to Chapter 1, *Faire connaissance de 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 2.6, 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.

Revision 2.6	28 Feb 2001

- 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.

Revision 2.5	23 Feb 2001

- Added Section 3.13, Complément sur les modules .
- Added Section 3.14, Le module os .
- Moved Example 3.36, Division de noms de chemins from Section 1.11, Assignation simultanée de plusieurs valeurs to Section 3.14, Le module os .
- Added Section 3.15, Assembler les pièces .
- Added Section 3.16, Résumé .
- Added Section 4.1, Plonger .
- Fixed program listing in Example 3.29, Parcourir un dictionnaire which was missing a colon.

Revision 2.4.1	12 Feb 2001

- Changed newsgroup links to use "news:" protocol, now that `deja.com` is defunct.
- Added file sizes to download links.

Revision 2.4	12 Feb 2001

- Added "further reading" links in most sections, and collated them in Appendix A, *Pour en savoir plus*.
- Added URLs in parentheses next to external links in text version.

Revision 2.3	9 Feb 2001

- Rewrote some of the code in Chapter 3, *Un framework orienté objet* to use class attributes and a better example of multi−variable assignment.
- Reorganized Chapter 3, *Un framework orienté objet* to put the class sections first.
- Added Section 3.8, Attributs de classe .
- Added Section 3.10, Traitement des exceptions .
- Added Section 3.11, Les objets fichier .
- Merged the "review" section in Chapter 3, *Un framework orienté objet* into Section 3.1, Plonger .
- Colorized all program listings and examples.
- Fixed important error in Section 1.2, 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 1.3, Documentation des fonctions , Section 1.4, Tout est objet , and Section 1.10, Définitions de variables .

Revision 2.2	2 Feb 2001

- Edited Section 2.4, 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.

Revision 2.1	1 Feb 2001

- Rewrote the example code of Chapter 2, *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 1.8, Présentation des listes .
- Added links to relevant sections in the summary lists at the end of each chapter (Section 1.15, Résumé and Section 2.9, Résumé).

Revision 2.0	31 Jan 2001

- Split Section 3.6, Méthodes de classe spéciales into three sections, Section 3.5, UserDict : une classe enveloppe , Section 3.6, Méthodes de classe spéciales , and Section 3.7, 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.

Revision 1.9	15 Jan 2001

- Removed introduction to Chapter 1, *Faire connaissance de Python*.
- Removed introduction to Chapter 2, *Le pouvoir de l introspection*.
- Removed introduction to Chapter 3, *Un framework orienté objet*.
- Edited text ruthlessly. I tend to ramble.

Revision 1.8	12 Jan 2001

- Added more examples to Section 1.11, Assignation simultanée de plusieurs valeurs .
- Added Section 3.3, Définition de classes .
- Added Section 3.4, Instantiation de classes .
- Added Section 3.6, 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.

Revision 1.71	3 Jan 2001

- Made several modifications to stylesheets to improve browser compatibility.

Revision 1.7	2 Jan 2001

- Added introduction to Chapter 1, *Faire connaissance de Python*.
- Added introduction to Chapter 2, *Le pouvoir de l introspection*.
- Added review section to Chapter 3, *Un framework orienté objet* [later removed]
- Added Section 3.9, Fonctions privées .
- Added Section 3.12, Boucles for .
- Added Section 1.11, 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.

Revision 1.6	11 Dec 2000

- Added Section 2.8, Assembler les pièces .
- Finished Chapter 2, *Le pouvoir de l introspection* with Section 2.9, Résumé .
- Started Chapter 3, *Un framework orienté objet* with Section 3.1, Plonger .

Revision 1.5	22 Nov 2000

- Added Section 2.6, Particularités de and et or .
- Added Section 2.7, 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 1.13, Mutation de listes .
- Encapsulated several more common phrases into entities.
- Upgraded to version 1.25 of the DocBook XSL stylesheets.

Revision 1.4	14 Nov 2000

- Added Section 2.5, Filtrage de listes .
- Added `dir` documentation to Section 2.3, type, str, dir et autres fonctions intégrées .
- Added `in` example in Section 1.9, 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`.

Revision 1.3	9 Nov 2000

- 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.	
Revision 1.2	6 Nov 2000
• 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.	
Revision 1.1	31 Oct 2000
• 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.	
Revision 1.0	30 Oct 2000
• Initial publication	

Appendix 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−4.1.zip) et les scripts de construction (download/diveintopython−common−4.1.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/).

Appendix 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.

Appendix H. Python 2.1.1 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