Initiation à Python par l'exemple Documentation

Version 1.5.0

Raphaël MARVIE

23 February 2012

Table des matières

Contents :

CHAPITRE 1

Avant-propos

Ce support propose une initiation au langage Python par l'exemple. Il fait suite à une formation d'une semaine et comportant 6 cours. Cette version essaye de compléter les transparents, mais ne regroupe malheureusement pas tout le discours accompagnant la présentation des transparents. La formation se faisant en salle machine, tous les exemples étaient testés interactivement pendant le cours. Cette approche est volontairement conservée ici et il est conseillé d'utiliser ce support avec un interpréteur lancé pour «voir ce qui se passe».

Ce support de cours est destiné à un public étant familier avec au moins un langage de programmation, si possible orienté objet, et être familier avec les notions d'objet. Ce support n'est pas un cours de programmation, ni un cours complet sur le langage Python, il ne représente qu'une initiation à différentes choses que l'on peut faire avec Python.

La version utilisée dans ce support est la 2.7. Les premiers compléments de ce support sont les documents associés au langage Python comme la librairie standard, le manuel de référence, ou bien le tutoriel officiel (tous les trois disponibles sur le site officiel http ://www.python.org).

Sans ambition d'être suffisant, la mise en ligne de ce support a pour unique objectif d'être éventuellement utile. Toute remarque concernant ce support est toujours la bienvenue !

r.m.

CHAPITRE 2

Introduction

2.1 Un peu d'histoire

Python est un langage de programmation objet interprété. Son origine est le langage de script du système d'exploitation *Amoeba* (1990). Il a été développé par Guido Von Rossum au CWI, à l'Université d'Amsterdam et nommé par rapport au *Monthy Python's Flying Circus*.

Depuis, Python est devenu un langage de programmation généraliste (*comp.lang.python* est créé en 1994). Il offre un environnement complet de développement comprenant un interpréteur performant et de nombreux modules. Un atout indéniable est sa disponibilité sur la grande majorité des plates-formes courantes (BeOS, Mac OS X, Unix, Windows).

Python est un langage *open source* supporté, développé et utilisé par une large communauté : 300 000 utilisateurs et plus de 500 000 téléchargements par an.

2.2 Python, à quoi bon ?

Pour résumer Python en quatre points forts.
– *Qualité* L'utilisation de Python permet de produire facilement du code évolutif et maintenable et offre les avantages de la programmation orientée-objet.
– *Productivité* Python permet de produire rapidement du code compréhensible en reléguant nombre de détails au niveau de l'interpréteur.
– *Portabilité* La disponibilité de l'interpréteur sur de nombreuses plates-formes permet l'exécution du même code sur un PDA ou un gros système [1].
– *Intégration* L'utilisation de Python est parfaitement adaptée l'intégration de composants écrit dans un autre langage de programmation (C, C++, Java avec Jython). Embarquer un interpréteur dans une application permet l'intégration de scripts Python au sein de programmes.
Quelques caractéristiques intéressantes :
– langage interprété (pas de phase de compilation explicite)
– pas de déclarations de types (déclaration à l'affectation)
– gestion automatique de la mémoire (comptage de références)

1. Même si un tel cas semblerait un peu bizarre.

– programmation orienté objet, procédural et fonctionnel
– par nature dynamique et interactif
– possibilité de générer du byte-code (améliore les performances par rapport à une interprétation perpetuelle)
– interactions standards (appels systèmes, protocoles, etc.)
– intégrations avec les langages C et C++

2.3 Python ou pas Python ?

Comme tout langage de programmation Python n'est pas la solution ultime à tous les besoins. Toutefois, on retrouve son utilisation dans différents contextes [successtory] comme Google, la NASA, YouTube, Industrial Light & Magic, le projet *One Laptop Per Child*, l'environnement Opie sur pocket PC, etc. Python est donc adapté pour réaliser de «grosses applications».

Quand Python ? Python est un très bon langage pour le développement agile cite{agile} et itératif. Il permet d'être réactif et évite le cycle lourd programmer / compiler / tester. De plus, le langage permet de vérifier très rapidement des hypothèses de mise en oeuvre. Une fois la solution réalisée en Python, il est alors toujours possible de l'optimiser davantage en re-développant certain de ses composants en C++ par exemple.

Python permet de produire des applications facilement extensibles non seulement par le développeur mais aussi dans certains cas par l'usager (de nombreuses librairies scientifiques sont disponibles).

Enfin, Python est un bon candidat pour l'automatisation de tâches systèmes. Le système d'administration de la distribution Gentoo Linux *portage* est développé en Python. Une partie au moins du support de l'intégration continue chez Google est fait en Python.

Quand pas python ?

Dynamicité est rarement compatible avec *haute performance*, Python n'est donc certainement pas le meilleur langage pour faire du traitement intensif (calcul numérique, extraction de données, etc.). Mais il reste très pertinent pour piloter ce type d'applications.

Dans certain cas, les usagers préfèrent limiter l'utilisation de Python à l'aspect intégration de composants afin de produire des systèmes performants. La version Python fournit une *preuve de concept* ou une version fonctionnelle avant de re-développer les points faibles. Toutefois, les performances de Python sont souvent suffisante pour des applications à forte charge (cf. YouTube)

Une dernière remarque pour terminer, Python permet d'identifier rapidement une mauvaise idée...

CHAPITRE 3

Premiers pas en Python

3.1 Utilisation de Python

Python, comme la majorité des langages dit de script, peut être utilisé aussi bien en mode interactif qu'en mode script / programme.

Dans le premier cas, il y a un dialogue entre l'utilisateur et l'interprète : les commandes entrées par l'utilisateur sont évaluées au fur et à mesure. Cette première solution est pratique pour réaliser des prototypes, ainsi que pour tester tout ou partie d'un programme ou plus simplement pour interagir aisément et rapidement avec des structures de données complexes. Le listing suivant présente comment lancer l'interprète (simplement en tapant *python* à l'invite du shell [1]) et comment en sortir (en tapant `Ctrl-D`).

```
$ python
Python 2.7.2 (v2.7.2:8527427914a2, Jun 11 2011, 15:22:34)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world!'
hello world!
>>> ^D
$
```

Pour une utilisation en mode *script* les instructions à évaluer par l'interprète sont sauvegardées, comme n'importe quel programme informatique, dans un fichier. Dans ce second cas, l'utilisateur doit saisir l'intégralité des instructions qu'il souhaite voir évaluer à l'aide de son éditeur de texte favori, puis demander leur exécution à l'interprète. Les fichiers Python sont identifiés par l'extension `.py`. Le listing suivant, l'interprète est appelé avec en paramètre le nom du programme à évaluer (le traditionnel *hello world'). Dans le cas d'un système Unix, la première ligne du fichier ''hello.py'* précise quel interprète utiliser pour évaluer le fichier si ce dernier est exécutable. Il suffit dans ce cas de taper le nom du fichier à l'invite du shell. :

```
$ cat hello.py
#! /usr/bin/env python
```

1. Dans tout ce support l'invite du shell sera identifié par la lettre `$`.

```
print 'hello world!'
$ python hello.py
hello world!
```

3.2 Structures de base

3.2.1 Commentaires

Comme dans la majorité des langages de script, les commentaires Python sont définis à l'aide du caractère #. Qu'il soit utilisé comme premier caractère ou non, le # introduit un commentaire jusqu'à la fin de la ligne. Comme toujours, les commentaires sont à utiliser abondamment avec parcimonie. Il ne faut pas hésiter à commenter le code, sans pour autant mettre des commentaires qui n'apportent rien. Le listing suivant présente une ligne de commentaire en Python. Les commentaires seront de nouveaux abordés en [sub :chap1 :doc] pour l'auto-documentation. Les commentaires introduits par # devraient être réservés au remarques sur le code en sa mise en oeuvre.

```
>>> # ceci est un commentaire
>>> print 'bouh' # ceci est aussi un commentaire
bouh
```

3.2.2 Typage en Python

En Python, tout est objet. Quelque soit les données que l'on manipule, ces données sont des objets dont les classes sont définies par l'usager, ou par l'environnement Python pour les types de base. Une conséquence est que l'utilisateur manipule les données au travers de références (qui donnent accès aux fonctionnalités des instances). Cette approche permet de rendre homogène les données manipulées par l'utilisateur (comme c'était le cas en SmallTalk). Ainsi, toutes les données suivantes suivant sont des objets : *1, [2, 3, 4], 5.6, 'toto', une instance de Foo*.

Python est un langage à typage dynamique. Ceci ne veut pas dire que les données que l'on manipule ne sont pas typées, mais que leur type est «calculé» lors de leur utilisation [2]. Dans ce contexte, le type des variables n'est pas défini explicitement par l'utilisateur. Ainsi, une même variable peut référencer dans un programme des objets de types différents [3].

```
>>> x = 1        # x reference un entier
>>> x = 'toto'   # x reference desormais une chaine
>>> x = Foo()    # x reference desormais une instance de Foo
```

2. En Python, ce calcul se résume à la possibilité pour l'objet de recevoir un message particulier.
3. Il est toutefois à noter que cette facilité ne devrait être utilisées que sous couvert du polymorphisme, sans quoi la lisibilité du programme s'en trouve réduite.

3.2.3 Arithmétique

Python permet d'exprimer très simplement des opérations arithmétiques. Dans le cas où tous les opérandes sont des entiers, alors les résultats seront aussi des entiers. Lorsqu'au moins un des opérandes est de type réel, alors tous les opérandes sont automatiquement convertis en réels.

```
>>> x = 1 + 2
>>> y = 5 * 2
>>> y / x
3
>>> y % x
1
>>> y = 5.5 * 2
>>> y
11.0
>>> x = 12.0 / 3
>>> x
4.0
```

Dans le contexte de l'arithmétique, les affections peuvent prendre deux formes. Ces deux formes ont un sens différent. Le listing suivant présente deux affectations qui pourraient être comprises de manière identique. Toutefois, la première forme (ligne 2) a pour conséquence de créer une nouvelle instance d'entier pour contenir l'ajout de 2 à la valeur de `x`. La seconde forme (ligne 3) ajoute 2 à la valeur de `x` sans créer de nouvelle instance. La manière d'écrire une opération a donc un impact sur son évaluation.

```
>>> x = 4
>>> x = x + 2
>>> x += 2
```

3.2.4 Chaînes de caractères

Les chaînes de caractères se définissent de plusieurs manières en Python. Il est possible d'utiliser indifféremment des guillemets simples ou des guillemets doubles. Le choix est souvent imposé par le contenu de la chaîne : une chaîne contenant des guillemets simples sera déclarée avec des guillemets doubles et réciproquement. Pour les autres cas, c'est indifférent. Enfin, comme tout est objet en Python une chaîne est donc un objet. Le listing suivant déclarer deux chaînes référencées par `x` et `y`. Enfin, la chaîne référencée par `z` est une chaîne de caractères multi-lignes (utilisation de trois quotes / guillemets simples ou doubles).

```
>>> x = 'hello '
>>> y = "world!"
>>> z = '''hello
... world'''
```

3.2.5 Concaténation

La concaténation de ces chaînes de caractères peut prendre deux formes. Dans les deux cas, l'opérateur + est utilisé pour exprimer la concaténation. La forme de la ligne 4 est un raccourci d'écriture.

```
>>> z = x + y
>>> z
'hello world!'
>>> x += y
>>> x
'hello world!'
```

3.2.6 Affichage

L'affichage de chaînes de caractères à l'aide de la fonction `print` peut se faire en concaténant explicitement des chaînes (que ce soit avec l'opérateur de concaténation ou en utilisant des virgules) ou en utilisant une chaîne de formatage comme la fonction `printf` du langage C. Cette seconde option est un peu plus puissante, mais aussi un peu plus lourde à utiliser. Le listing suivant présente trois manière d'afficher des chaîne de caractères.

```
>>> print 'I say: ' + x
I say: hello world!
>>> print x, 2, 'times'
hello world! 2 times
>>> print "I say: %s %d time(s)" % (x, 2)
I say: hello world! 2 time(s)
```

3.2.7 Manipulations

Python offre une méthode simple pour accéder aux caractères contenus dans une chaîne : une chaîne est manipulée comme une séquence indexée de caractères. Ainsi, chaque caractère est accessible directement par son index (le premier étant indexé 0) en utilisation des crochets. En plus de cet accès unitaire aux caractères, il est possible d'accéder à des sous-chaînes en précisant la tranche souhaitée l'index de début (qui est inclus) étant séparé de l'index de fin (qui est exclu) par le caractère `:`. Dans le cas des sous-chaînes, la valeur fournie est une copie et non un accès à une partie de la chaîne d'origine.

Le listing suivant donne quelques exemples d'accès à un caractère (ligne 2), ou à une sous-chaîne pour le reste. La colonne de gauche présente des accès à partir du début de la chaînes (les index sont positifs). La ligne 6 signifie que l'on souhaite le contenu de `x` du quatrième caractère à la fin. Enfin, la dernière ligne réalise une copie de la chaîne x.

```
>>> x = 'hello world!'
>>> x[4]
'o'
>>> x[2:4]
'll'
```

```
>>> x[3:]
'lo world!'
>>> x[:]
'hello world!'
```

Enfin, un index négatifs précise que le calcul s'effectue depuis la fin de la chaîne.

```
>>> x[-3:]
'ld!'
>>> x[1:-1]
'ello world'
```

3.2.8 Listes

Les *listes* Python sont des ensemble ordonnés et dynamique d'éléments. Ces ensemble peuvent contenir des éléments de différents types, leur seul point commun est que ce sont des objets. Et comme tout est objet, les listes sont elles mêmes des objets (instances de la classe `list`).

L'exemple suivant crée tout d'abord deux listes vides avec les deux manières possibles.

```
>>> mylist = []
>>> mylist2 = list()
```

Ensuite, après avoir défini `x`, une liste contenant la chaîne 'bar', l'entier 12345 et l'objet référencé par la variable `x` est créée.

```
>>> x = True
>>> foo = ['bar', 12345, x]
>>> foo
['bar', 12345, True]
```

Les listes et les chaînes ont pour point commun le fait d'être des ensembles ordonnés. L'accès à un élément d'une liste se fait donc aussi par indexation (ligne 1). Il est possible de prendre une partie de liste (ligne 3) et d'obtenir une copie de liste (ligne 5). Une copie de liste crée une nouvelle liste, mais partage les éléments contenus dans la liste d'origine.

Cette dernière manipulation est très importante lors de parcours de listes (voir section [sub-Chap1LoopPbl]) pour ne pas modifier la liste qui est parcourue et se retrouver dans une boucle sans fin.

```
>>> foo[2]
True
>>> foo[1:]
[12345, True]
>>> bar = foo[:]
```

Enfin, le contenu de la liste peut être changé par simple affection en utilisant l'index cible. Comme la liste référencée par `bar` est une copie de la liste référencée par `foo`, le dernier élément de `bar` n'est pas modifié par l'affectation du dernier élément de `foo`.

```
>>> foo[2] = 1
>>> foo
['bar', 12345, 1]
>>> bar[-1]
True
```

L'ajout d'éléments dans une liste se fait à l'aide des méthodes `append` pour un ajout en fin de liste, et `insert`, pour un ajout à un index donné. Enfin, la méthode `extend` ajoute le contenu d'une liste passé en paramètre à la fin de la liste.

```
>>> foo.append('new')
>>> foo
['bar', 12345, 1, 'new']
>>> foo.insert(2, 'new')
>>> foo
['bar', 12345, 'new', 1, 'new']
>>> foo.extend([67, 89])
>>> foo
['bar', 12345, 'new', 1, 'new', 67, 89]
```

La méthode `index` permet de connaître l'index de la première occurrence d'un élément dans une liste. Dans le cas où l'élément fournit en paramètre n'est pas présent, la méthode lève l'exception `ValueError`. L'utilisation de la construction `in` retourne quant à elle `True` si l'élément est présent dans la liste et `False` sinon.

```
>>> foo.index('new')
2
>>> foo.index(34)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>> 34 in foo
False
```

Les listes peuvent être fusionnées par concaténation. Ces concaténations peuvent aussi bien se faire par copie que par ajout. La ligne 1 représente une concaténation par copie des deux listes existantes. La ligne 4 présente une concaténation par ajout d'éléments à une liste existante. Enfin, la ligne 7 présente une manière simple de créer une liste par répétition d'un motif qui doit être lui même une liste.

```
>>> bar = [0, 1] + [1, 0]
>>> bar
[0, 1, 1, 0]
>>> bar += [2, 3]
>>> bar
[0, 1, 1, 0, 2, 3]
>>> [0, 1] * 3
[0, 1, 0, 1, 0, 1]
```

Attention Dans le cas d'une création de liste par répétition, les éléments ne sont pas dupliqués, mais ce sont leurs références qui sont répétées (et donc présentes plusieurs fois dans la liste

finale). Utiliser l'opérateur * pour la construction de matrice est donc une mauvaise idée car toutes les lignes seraient alors la même liste référencée plusieurs fois.

```
>>> matrix = [[1, 2, 3]] * 3
>>> matrix
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> matrix[1][1] = 4
>>> matrix
[[1, 4, 3], [1, 4, 3], [1, 4, 3]]
```

De manière symétrique à l'ajout d'éléments dans une liste, il est possible d'en supprimer. La méthode `remove` permet de supprimer la première occurrence d'un élément d'une liste en le désignant.

```
>>> foo.remove('new')
>>> foo
['bar', 12345, 1, 'new', 67, 89]
```

Si l'élément fourni en paramètre n'existe pas dans la liste, l'exception `ValueError` est levée.

```
>>> foo.remove(34)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

L'opérateur `del` (*delete*) permet de détruire une référence à un objet Python, ou à une partie d'une liste[4].

```
>>> del foo[1:3]
>>> foo
['bar', 'new', 67, 89]
```

D'autres formes plus avancées de manipulation de listes sont présentées dans la section [sub :chap1 :listes].

3.2.9 Listes et chaînes de caractères

Les listes et les chaînes de caractères sont similaire dans leur structure et dans leur manipulation. Certaines méthodes disponibles sur les chaînes manipulent les deux structures de données.

La méthode `join` disponible sur les chaîne permet de construire une chaîne de caractère depuis une liste de chaînes. La chaîne sur laquelle est invoquée la méthode `join` est alors utilisé comme séparateur des différents éléments de la liste.

```
>>> ' ; '.join(['a', 'b', 'c'])
'a ; b ; c'
```

4. Lorsque la dernière référence à un objet est détruite alors l'objet et lui même effectivement détruit.

De manière symétrique, la méthode `split`, disponible sur les chaînes, permet de décomposer une chaîne de caractères en une liste de sous chaînes. Ce découpage se fait par rapport à un ou plusieurs caractères. Dans le cas où un entier est passé comme second argument, il précise le nombre maximum de découpage.

```
>>> 'hello crazy world!'.split(" ")
['hello', 'crazy', 'world!']
>>> 'hello crazy world!'.split(" ", 1)
['hello', 'crazy world!']
```

3.2.10 Tuples

Les `tuples` sont des ensemble ordonnés et immuables d'éléments. Comme les listes, les tuples peuvent contenir des données de différents types. La première ligne présente une déclaration classique (avec des parenthèses) alors que la secone ligne présente la notation abrégée. La virgule est importante pour préciser que l'on parle d'un tuple à un élément et non de la valeur 12. Cette remarque serait valable dans la cas d'une déclaration *parenthésée* d'un tuple à un élément.

```
>>> foo = ('bar', 12345, x)
>>> bar = 12,
```

Comme les listes, les tuples sont accessibles par indexation, et la construction `in` permet de tester la présence d'un élément dans le tuple. Cependant, une fois créé le contenu d'un tuple ne peut être modifié.

```
>>> foo[1]
12345
>>> foo[:-1]
('bar', 12345)
>>> 'bar' in foo
True
>>> foo[1]  = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Remarque Critères de choix entre une liste et un tuple :
– les tuples sont plus rapides à parcourir que les listes,
– pour définir des constantes, utiliser des tuples.
Il est possible de convertir des tuples vers des listes et réciproquement.

```
>>> list((1, 2, 3))
[1, 2, 3]
>>> foo = tuple([1, 2, 3])
>>> foo
(1, 2, 3)
```

Pour terminer, Python offre l'affectation multiple pour assigner différentes valeurs depuis un tuple de manière simultanée. Ici encore, les parenthèses peuvent être omises. Il est aussi possible de faire des assignation en cascade.

```
>>> (x, y, z) = foo
>>> x
1
>>> a, b, c = foo
>>> b
2
>>> d = e = b
>>> d
2
>>> e
2
```

3.2.11 Dictionnaires

Les dictionnaires, parfois appelés tableaux associatifs, sont des ensembles non ordonnés d'éléments indexés par des clés. Une clé doit obligatoirement être immuable (chaîne, entier ou tuple). D'autre part, une clé est toujours unique.

Un dictionnaire est déclaré par deux accolades ou en utilisant le type `dict`. Il peut être créé vide ou avec des valeurs initiales.

```
>>> dict1 = {}
>>> dict2 = {'foo': 456, 123: 'bar'}

>>> dict3 = dict()
>>> dict4 = dict(foo=456, bar=123)
```

L'ajout et l'accès aux éléments d'un dictionnaire se fait par indexation sur la clé associé à l'élément.

```
>>> dict1['foo'] = 456
>>> dict1[123] = 'bar'

>>> dict1[123]
'bar'
```

L'affichage d'un dictionnaire présente une liste de paires «clé : valeur».

```
>>> dict1
{123: 'bar', 'foo': 456}
>>> dict4
{'foo': 456, 'bar': 123}
```

L'utilisation de la classe `dict` pour créer un dictionnaire non vide, ne peut se faire qu'avec des chaines (utilisation d'arguments nommés).

```
>>> dict4
{'foo': 456, 'bar': 123}
```

Les dictionnaires offrent des méthodes pour manipuler les clés. La méthode `keys` retourne une liste de toutes les clés du dictionnaire et la méthode `has_key` retourne `True` si la clé donné en paramètre est présente dans le dictionnaire et `False` dans le cas contraire. Il est aussi possible d'utiliser l'opérateur `in`

```
>>> dict1.keys()
[123, 'foo']

>>> dict1.has_key('bar')
False

>>> 123 in dict1
True
```

La méthode `values` donne accès à une liste des valeurs contenues dans le dictionnaire et la méthode `items` donne une liste de tuples, contenant chacun une paire clé, valeur.

```
>>> dict1.values()
['bar', 456]
>>> dict1.items()
[(123, 'bar'), ('foo', 456)]
```

La modification d'une valeur associée à une clé se fait simplement en affectant de nouveau la valeur indexée dans le dictionnaire par la clé en question. Enfin, l'opérateur `del` permet de supprimer une association du dictionnaire.

```
>>> dict1[123] = 789
>>> dict1
{123: 789, 'foo': 456}
>>> del dict1['foo']
>>> dict1
{123: 789}
```

A l'aides des méthodes de manipulation des clé et des valeurs, il est possible de parcourir un dictionnaire de plusieurs manières. Les quelques lignes suivantes donnent deux exemples de parcours.

```
>>> flames = {'windows': 'bof', 'unix': 'cool'}
>>> for key in flames.keys():
...     print key, 'is', flames[key]
...
windows is bof
unix is cool
>>> for key, value in flames.items():
...     print key, 'is', value
...
windows is bof
unix is cool
```

3.3 Constructions

3.3.1 Structuration et indentation

La structuration d'un programme Python est définie par son indentation. Le début d'un bloc est défini par un ':', la première ligne pouvant être considérée comme un en-tête (test, boucle, définition, etc.). Le corps du bloc est alors indenté de manière plus importante (mais régulière) que l'en-tête. Enfin, la fin du bloc est délimité par le retour à l'indentation de l'en-tête. La convention en Python est d'utiliser quatre espaces pour chaque niveau d'indentation. Les bloc peuvent être imbriqués.

```
<en-tete>:
    <instructions>
```

Dans le cas de bloc de taille réduite, par exemple une seule instruction, un bloc peut être défini sur une seule ligne. Le caractère : sert toujours à délimiter l'en-tête du corps. Cependant, cette utilisation n'est pas vraiment bénéfique quant à la lisibilité du code, si ce n'est pour faire tenir du code sur un transparent. Elle est donc à éviter, l'époque où les caractères étaient comptés dans un fichier est bien révolue.

```
<en-tete>: <instruction>
```

Cette structuration est utilisée aussi bien pour définir des boucles, des tests, des fonctions, des classes ou encore des méthodes.

3.3.2 Tests

Conditions booléennes En Python, tout ce qui n'est pas faux est vrai. Les conditions booléennes fausses se résument au données «vide» en plus du faux :

```
False, None, 0, "", [], list(), {}, dict(), (), tuple()
```

Expression booléennes Python propose les opérateurs de comparaison et les opérateurs booléen suivants :

```
<, <=, >, >=, !=, ==, is
and, or, not
```

L'utilisation du == permet de comparer l'équivalence de valeurs (donc d'objets), alors que `is` permet de comparer si deux variables référence une même instance.

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
>>> l1 == l2
True
>>> l1 is l2
False
>>> l3 = l1
```

```
>>> l3 is l1
True
```

Python offre une seule construction pour réaliser des tests : le `if then else`. Une particularité de cette mise en oeuvre est la possibilité d'enchaîner les tests avec la construction `elif`. Le `else` et le `elif` sont bien sûr optionnels. Enfin, a clause `elif` peut être utilisée plusieurs fois dans un même bloc.

```
if x == 'hello':
    print 'hello too!'
elif x == 'bonjour':
    print 'bonjour aussi!'
else:
    print 'moi pas comprendre'
```

Depuis la version 2.5, il est aussi possible d'utiliser une expression conditionnelle (par opposition à la *construction* `if` du paragraphe précédent)

```
oeufs = 4 if souhaite_une_tarte() else 0
```

L'absence du `switch` se justifie par le fait que Python est un langage objet. Dans cette approche, l'utilisation du polymorphisme replace naturellement le switch. Lorsque l'on ne développe pas avec une approche objet, le switch se remplace simplement par l'utilisation d'un dictionnaire. Ce dictionnaire contient, par exemple, des fonctions associées à des clés (pas nécessairement des chaînes) pour choisir la bonne fonction en fonction du contexte

```
traitements = {
  'en': traiter_anglais,
  'de': traiter_allemand,
  'fr': traiter_francais,
  # etc.
}

def traiter_commande(langue, commande):
  traiter = traitements[langue]
  traiter(commande)
```

Les comparaisons entre chaînes de caractères se fait selon l'ordre lexicographique. Les comparaisons entre listes et tuples se fait éléments par éléments de la gauche vers la droite.

```
>>> 'abc' < 'abd'
True
>>> 'abc' < 'abcd'
True
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3) < (1, 2, 3, 4)
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
```

3.3.3 Boucles

Deux types de boucles sont disponibles : les boucles énumérées (`for`) et les boucles basées sur un test de fin (`while`). Ces deux constructions suivent le même schéma : un en-tête qui décrit l'évolution de la boucle, un ensemble d'instructions qui sont évaluées à chaque tour de boucle et une partie optionnel qui est évaluée en sortie de boucle (introduite par le mot-clé `else`). Enfin, comme en C les boucles peuvent contenir les branchements `continue` pour passer à l'itération suivante et `break` pour sortir de la boucle (dans ce cas la clause `else` n'est pas évaluée).

Boucles énumérées

Une boucle `for` définit une variable qui prend successivement toutes les valeurs de la séquence (liste ou tuple) parcourue (ligne 1). La clause `else` est évalué lorsque la séquence est épuisée et s'il n'y a pas eu de sortie de boucle avec un `break` (ligne 3).

```
for <var> in <sequence>:
    <instructions>
else:
    <instructions, sequence epuisee sans break>
```

La fonction `range` produit une liste de tous les entiers entre une borne inférieur et une borne supérieur. Cette construction est utile lorsqu'une boucle `for` repose sur l'utilisation d'une séquence d'entiers. La fonction `range` est utilisable de trois manières :
– un seul paramètre spécifiant le nombre d'éléments (ligne 1),
– deux paramètres spécifiant la borne inférieure (inclue) et supérieure (exclue) (ligne 3),
– trois paramètres spécifiant les bornes et le saut (incrément entre deux éléments de la séquence) (ligne 5).

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

Boucle `for` parcourant une séquence de chaîne.

```
>>> a = ['hello', 'world']
>>> for elt in a:
...     print elt
...
hello
world
```

Boucle parcourant une liste de valeurs pour accéder aux éléments d'une liste.

```
>>> for idx in range(len(a)):
...     print idx, a[idx]
...
```

```
0 hello
1 world
```

S'il est nécessaire de parcourir une séquence tout en connaissant l'index de l'élément courant, la fonction `enumerate` est une solution plus «pythonesque».

```
>>> for idx, val in enumerate(a):
...     print idx, val
...
0 hello
1 world
```

Enfin, une chaîne étant une séquence de lettre, elle peut être parcourue comme une séquence.

```
>>> for lettre in 'bar':
...     print lettre
...
b
a
r
```

Boucles avec condition

Une boucle `while` définit une condition booléenne avec les même règles que pour les tests (ligne 1). Tant que cette condition est respectée, les instruction du bloc associé au `while` sont évaluées. La clause `else` est évalué lorsque la condition est fausse et qu'il n'y a pas eu de sortie de boucle avec un `break` (ligne 3).

```
while <condition>:
    <instructions>
else:
    <instructions, condition fausse>
```

La boucle suivante représente la boucle minimale définissable avec un `while`. Certes, elle est stupide mais elle permet d'illustrer l'utilisation du `Ctrl-C` pour interrompre un traitement en cours d'exécution.

```
>>> while 1:
...     pass
...
KeyboardInterrupt
```

3.3.4 Fonctions

Il n'y a que des fonctions en Python (et non une distinction entre fonction et procédures comme dans certains langages). Une fonction est définie avec le mot clé `def`. Une fonction retourne toujours une valeur. Si une fonction ne contient pas de clause `return`, la valeur `None` est alors retournée.

```
>>> def fib(n): # suite de fibonacci jusque n
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a + b
...
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89
```

Les paramètres d'une fonction peuvent être définis avec une valeur par défaut. A l'utilisation de la fonction, ces paramètres sont alors optionnels. Dans le cas d'une fonction avec des paramètres optionnels, l'utilisation des paramètres doit être ordonnée, les paramètres sont affectés dans l'ordre de leur définition, ou nommée.

```
>>> def welcome(name, greeting='Hello', mark='!'):
...     print greeting, name, mark
...
>>> welcome('world')
Hello world !
>>> welcome('monde', 'Bonjour')
Bonjour monde !
>>> welcome('world', mark='...')
Hello world ...
```

Python permet de définir des fonctions anonymes, en utilisant la forme *lambda*[5]. Une telle fonction est réduite à une simple expression. Cette construction est utile pour passer des fonctions en paramètre d'autre fonctions pour les configurer ou dans le cas des *list mappings* (voir section [subChap1List]). Nous verrons un peu plus tard que ce n'est pas la seule possibilité.

```
>>> def compare(a, b, func=(lambda x,y: x < y)):
...     return func(a,b)

>>> compare(1, 2)
True
>>> compare(1, 2, func=(lambda x,y: x > y))
False
```

3.3.5 Documenter

La documentation fait partie intégrante du code. En plus de la possibilité de mettre des commentaires avec l'utilisation de #, Python offre la possibilité d'écrire du code auto-documenté. Ceci est valable pour les fonctions, les classes et les modules. Une telle documentation d'un élément est accessible vie l'attribut `__doc__` de ces différents éléments. Cette technique est utilisable automatiquement par les environnements intégrés de développement ou par l'utilisation de la fonction `help` en mode interactif.

La définition des commentaires avec # devrait être réservé aux remarques techniques sur le code et pour les développeur de l'élément en question. La documentation doit quant à elle être

5. Ici encore, cela sent la programmation fonctionnelle type Lisp à plein nez...

destinée au développeurs mais surtout aux utilisateurs de l'élément documenté. Elle doit donc refléter le comportement de l'élément. Cette seconde forme est à mettre en parallèle avec la *javadoc* pour les habitué(e)s.

La définition de la documentation se fait sur la première ligne après la déclaration en utilisant des chaînes de caractères multi-lignes, même si la documentation tient sur une seule, en utilisant des guillemets doubles (ligne 2-5). Une bonne manière de définir la documentation est de fournir une version courte sur une ligne, de passer une ligne et de fournir des détails.

```
>>> def dummy():
...     """Cette fonction est du pipeau...
...
...     Cette fonction est vraiment du pipeau,
...     elle ne fait absolument rien."""
...     pass

>>> help(dummy)
Help on function dummy in module __main__:

dummy()
        Cette fonction est du pipeau...

        Cette fonction est vraiment du pipeau,
        elle ne fait absolument rien.
```

3.4 Autres éléments sur les séquences

3.4.1 Fonctions de manipulation de séquences

Python offre un certain nombre de fonction permettant d'appliquer des traitements sur tous les éléments d'une séquence afin de définir des filtres, de réaliser des calculs ou de calculer des *hashcode*.

Filtre

La fonction `filter` applique la fonction passée en premier argument sur chacun des éléments de la séquence passée en second argument et retourne une nouvelle liste qui contient tous les éléments de la séquence pour lesquels la fonction a retourné une valeur vrai.

```
>>> def funct1(val):
...     return val > 0
...
>>> filter(funct1, [1, -2, 3, -4, 5])
[1, 3, 5]
>>> def iseven(x):
...     return x % 2
...
```

```
>>> filter(iseven, [1, 2, 3, 4, 5, 6])
[1, 3, 5]
```

Application

La fonction `map` applique la fonction passée en premier argument sur chacun des éléments de la ou des séquences passées en paramètre. Dans le cas où plusieurs séquences sont passées en paramètre, la fonction doit prendre autant de paramètres qu'il y a de séquences. `map` retourne une liste contenant le résultat de chacun des calculs.

```
>>> def sum(x, y):
...     return x + y
...
>>> map(sum,  [1, 2, 3], [4, 5, 6])
[5, 7, 9]
>>> map(iseven, [1, 2, 3, 4, 5, 6])
[1, 0, 1, 0, 1, 0]
```

Réduction

La fonction `reduce` réduit une séquence par l'application récursive d'une fonction sur chacun de ses éléments. La fonction passée comme premier paramètre doit prendre deux arguments. La fonction `reduce` prend un troisième paramètre optionnel qui est la valeur initiale du calcul récursif.

```
>>> reduce(sum, [1, 2, 3, 4, 5])
15
>>> reduce(sum, [1, 2, 3, 4, 5], -5)
10
```

3.4.2 Listes en compréhension

La définition de listes en compréhension [6] permet de créer des listes de manière concise sans utiliser aux fonctions `map`, `filter`, etc.

L'exemple suivant (ligne 2) permet d'appliquer une même opération à tous les éléments d'une liste. Cet exemple est donc similaire à l'utilisation de la fonction `map`. La suite (ligne 4) permet de ne sélectionner que certains éléments de la liste, ici les nombres impaires. Cet exemple est similaire à l'utilisation de la fonction `filter`. La ligne 6 montre une combinaison des deux formes, en élevant au carré les nombres pairs de la liste `foo`.

```
>>> foo = [1, 2, 3, 4]
>>> [elt * 2 for elt in foo]
[2, 4, 6, 8]
```

6. Compréhension : Ensemble des caractères qui appartiennent à un concept et servent à le définir. *Définir un ensemble en compréhension*. Par opposition à *en extension*.

```
>>> [elt for elt in foo if elt % 2]
[1, 3]
>>> [elt**2 for elt in foo if elt % 2 is 0]
[4, 16]
```

Il est enfin à noter que l'utilisation de ces constructions est en général plus performante que l'utilisation d'une boucle `for`. En effet, leur mise en oeuvre est faite au coeur de l'interprète au lieu d'être interprétée.

3.4.3 Itérateurs et générateurs

Les itérateurs (PEP 234) et les générateurs (PEP 255) sont apparus avec les versions 2.2 et 2.3 de Python. Ils sont monnaie courante depuis Python 2.4, et leur usage est recommandé.

Itérateurs

L'apparition des itérateurs a modifié la philosophie des boucles `for` pour le parcours de séquences. Dans les version précédentes de Python, l'itération sur une séquence se faisait en utilisant l'index pour accéder aux éléments de la séquence. Lorsque la séquence était terminée, l'exception `IndexError` (voir section [sec :chap3 :exceptions] pour les exceptions) signalait la fin de l'itération.

Désormais, un itérateur est un objet (voir le chapitre [chap3] pour les objets) auquel on demande l'élément suivant. Lorsqu'il n'y a plus de suivant, l'exception `StopIteration` signale la fin de l'itération. Les types de bases comme les listes, dictionnaires, tuples retournent désormais des itérateurs pour leur parcours. Ces itérateurs sont utilisés automatiquement par les boucles `for` et par l'opérateur `in` par exemple.

La définition d'un itérateur peut se faire avec la fonction `iter` des deux manières suivantes : soit l'argument est une séquence (ou fournit un itérateur), soit le paramètre est une fonction qui sera appelée tant que la sentinelle n'est pas retournée par la fonction. L'exemple suivant présente la première formes.

```
>>> iter([1,2,3,4])
<listiterator object at 0x...>
```

Générateurs

Les générateurs peuvent être considérés comme une évolution des listes en compréhension. Syntaxiquement, les `[]` ou `list()` deviennent des `()`, comme le montre l'exemple suivant. Une des différences entre les deux formes concerne l'utilisation de la mémoire : un générateur ne construit pas toute la séquence en mémoire *a priori*, mais chaque élément de la séquence est produit lorsque c'est nécessaire. Pour obtenir une séquence construite par un générateur, il suffit de passer le générateur en paramètre de la construction d'une liste [7].

7. Depuis Python 2.4, la notation des listes en compréhension revient à la construction d'une liste à partir d'un générateur.

```
>>> (elt * 2 for elt in foo)
<generator object <genexpr> at 0x...>

>>> for i in (elt * 2 for elt in foo):
...     print i,
2 4 6 8
>>> list(elt * 2 for elt in foo)
[2, 4, 6, 8]
```

La définition de fonctions générateurs se fait en utilisant le mot clé `yield`. Le déroulement de la fonction est interrompu à chaque utilisation de `yield`, le reste sera évalué lorsque l'appelant demandera le prochain élément. Le `yield` peut être considéré comme un `return` qui ne met pas fin à l'exécution de la fonction.

Lorsqu'il est appelé, un générateur retourne un itérateur, ce qui permet une utilisation dans une boucle `for` comme dans l'exemple suivant. C'est l'utilisation de l'itérateur retourné qui pilote le déroulement du générateur : l'exécution s'arrête sur le `yield` qui est évalué à chaque appel du suivant sur l'itérateur.

```
>>> def generateur():
...     i = 0
...     while i < 10:
...             yield i
...             i += 1
...
>>> for i in generateur():
...     print i,
...
0 1 2 3 4 5 6 7 8 9
```

3.4.4 Boucles énumérées problématiques

Certaines boucles de manipulation des listes peuvent être problématiques, par exemple si la boucle modifie la liste qu'elle est en train de parcourir. Pour que ce genre de boucle se passe bien, et ne soit pas sans fin, il est important dans certains cas de faire une copie de la liste pour le parcours et de travailler sur la version originale de la liste. Par défaut, tout argument est passé par référence et donc sans copie (en fait copie de la liste des références et non des objets contenus) la boucle suivante serait sans fin : Cette boucle rajoute les éléments positifs en début de liste, comme le premier élément est positif cette boucle est toujours sur l'élément 1 alors que la liste ne fait que croître.

```
>>> a = [1, -2, 3, -4, 5, -6]
>>> for elt in a [:]:
...     if elt > 0: a.insert(0, elt)
...
>>> a
[5, 3, 1, 1, -2, 3, -4, 5, -6]
```

3.5 Exercices

3.5.1 Manipulations de données

Le fichier `listetu.py` contient une liste de 10 dictionnaires. Etudier cette structure de données et définir une fonction qui écrit chaque fiche sur la sortie standard de la manière suivante (en sautant une ligne entre chaque fiche) :

```
dossier    : 1
nom        : doe
prenom     : john
universite : lille1
discipline : informatique
niveau     : 4
moyenne    : 17
```

CHAPITRE 4

Quelques modules et *built-in*

4.1 Définition et utilisation de modules

4.1.1 Définition

Dans le chapitre précédent, l'ensemble des extraits de code ont été saisis de manière interactive. Cette méthode n'est pas viable pour les traitements que l'on exécute plusieurs fois ou que l'on souhaite exécuter sur plusieurs machines. Pour rendre le code persistant, la première solution est d'écrire un «programme», c'est-à-dire de saisir le code dans un fichier texte avec l'extension `.py`. Un programme peut ainsi être exécuté plusieurs fois. Cependant, même si ce programme est correctement structuré avec des fonctions, il n'est pas possible de réutiliser facilement ces dernières (si ce n'est avec un copier-coller qui est une abomination).

Pour capitaliser les développement, Python propose la notion de module. Un module permet de fournir des bibliothèques de fonctions, structures de données, classes, à intégrer dans les programmes. Dans le cas de python, produire un module est identique à produire un programme : faire un fichier. Les définitions contenues dans un fichier sont utilisable globalement ou unitairement. Ainsi, le fichier `examples.py` peut être utilisé comme un module (nommé comme le fichier) et offrir l'accès aux deux fonctions à tout programme en ayant l'utilité.

La chaîne de documentation doit être mise avant toute déclaration (c'est-à-dire aussi avant les clauses `import`) pour être considérée comme documentation du module. Les deux premières lignes de ce fichier (optionnelles) ont la significations suivante :

– la première ligne de ce fichier indique que ce fichier sera reconnu comme un programme Python par un environnement Unix (s'il a les droits en exécution) [1] ;
– la seconde ligne précise le type d'encodage du fichier : ici l'encodage utf-8 supportant la gestion des accents. Dès qu'un fichier Python contient un accent, cette ligne est obligatoire (dans le cas contraire elle peut être omise).

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
#
# examples.py
```

1. La construction # ! sert à préciser, sous Unix, le programme à utiliser pour exécuter un script.

```
#

"""
Regroupe les définitions des fonctions relatives au chapitre 1 de
'Initiation a python par l'exemple'.
"""

def fib(n):
    """Calcule la suite de Fibonacci jusque n"""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a + b

def welcome(name, greeting='Hello', mark='!'):
    """Un hello world configurable"""
    print greeting, name, mark
```

Utilisation d'un module

La construction `import` permet d'importer un module et de fournir accès à son contenu[2]. L'importation d'un module peut se faire de deux manière. La première solution est de désigner le module que l'on souhaite utiliser, son contenu est alors utilisable de manière «scopée», c'est-à-dire en préfixant le nom de la fonctionnalité du nom du module (ligne 2). La seconde solution repose sur la construction `from import` où l'on identifie la ou les fonctionnalités que l'on souhaite importer d'un module (ligne 4). Dans le cas d'un import de plusieurs fonctionnalités, les noms sont séparés par des virgules. L'utilisation se fait alors sans préfixer par le nom de module («non scopé»). Enfin, il est possible d'importer, avec cette seconde approche, tous les éléments d'un module en utilisant la notation `*` (ligne 7). Attention, avec cette dernière forme car il y a pollution du namespace global : deux imports consécutifs peuvent charger deux définition d'une même fonction. La seconde masquera la première.

```
>>> import examples
>>> examples.welcome('raphael')
Hello raphael !
>>> from examples import welcome
>>> welcome('raphael')
Hello raphael !
>>> from examples import *
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89
```

Utilisation mixte

Un fichier peut à la fois définir un module et un programme. Cette définition mixte est intéressante pour, par exemple, utiliser les fonctionnalités d'un programme indépendamment de

2. Nous considérons ici que l'interprète a été lancé dans le répertoire contenant le fichier `examples.py`.

celui-ci, ou pour associer le code de test à un module. Dans ce cas, la fin du fichier contient la définition de l'initialisation du code comme un programme. Pour cela, un test est réalisé afin de savoir si le code est importé ou exécuté. Si le code est importé, la variable `__name__` contient le nom du module, sinon elle contient la chaîne `__main__`. Ici, dans le cas d'une exécution certaines des fonctions du module sont testées. (Sinon il ne se passe rien.)

```
if __name__ == '__main__':
    welcome('world')
    welcome('monde', 'Bonjour')
    welcome('world', mark='...')
    fib(100)
```

L'utilisation du fichier `examples` comme un programme donne la trace d'exécution suivante. Que l'on utilise explicitement l'interprète ou non (du fait de la première ligne du fichier) la trace est identique.

```
$ python examples.py
Hello world !
Bonjour monde !
Hello world ...
1 1 2 3 5 8 13 21 34 55 89
$ ls -l
total 100
-rwxr-x---  1 raphael raphael  1046 2005-03-11 11:51 examples.py*
$ ./examples.py
Hello world !
Bonjour monde !
Hello world ...
1 1 2 3 5 8 13 21 34 55 89
```

Remarques

Lors de l'utilisation de la construction `import`, l'interprète recherche la disponibilité des modules demandé dans le chemin décrit par la variable d'environnement `PYTHONPATH`. La valeur par défaut de cette variable contient le répertoire d'installation de Python et le répertoire courant.

Lors de l'utilisation de la construction `from` module `import *`, tout le contenu du module est importé à l'exception des définition dont le nom commence par un _ (qui reflètent la notion de privé en Python). L'utilisation de ce type d'importation allège le code saisi, ce qui est intéressant en interactif. Toutefois, une fois le préfixe par le nom de module perdu il peut de nouveau y avoir des conflits de noms entre les fonctionnalités importées depuis plusieurs modules.

Enfin, les modules peuvent être organisés selon une structure hiérarchique. Dans ce cas, les modules contenant des sous-modules, encore nommés packages, sont définis comme des répertoires sur le système de fichier. Le nommage des modules se fait alors par concaténation : `mod.submod`. Afin de faciliter la portabilité du code, tout répertoire doit contenir un fichier nommé `__init__.py`. Pour permettre le chargement de tous les sous-modules, ce fichier doit contenir la liste des modules du package (répertoire) stockée dans la variable `__all__`.

```
$ cat graphical/__init__.py
__all__ = ['basic', 'advanced']
```

4.2 Quelques modules standards

Python offre un grand nombre de modules. Ne sont présentés ici que les quatre modules considérés comme incontournables (C'est-à-dire ceux que l'auteur utilise couramment) :
– `sys` fournit les paramètres et fonctions liées à l'environnement d'exécution,
– `string` fournit des opérations courantes sur les chaînes de caractères (équivalentes aux méthodes de la classe `string` plus quelques bonus),
– `re` fournit le support des expressions régulières pour faire du *pattern matching* et des substitutions,
– `os` fournit l'accès aux services génériques du système d'exploitation.
Pour chacun de ces modules, les fonctionnalités de base sont présentées et illustrées sur des exemples.

4.2.1 Le module `sys`

Quelques constantes

`argv` Séquence des paramètres passé sur la ligne de commande (`argv[0]` représente le nom du script).

`stdin stdout stderr` Objets de type `file` (fichier) représentant les entrées et sorties standard. Ces objets peuvent être remplacés par tout objet offrant une méthode `write`.

`path` Séquence contenant les chemins de la variable d'environnement `PYTHONPATH`. Cette séquence peut être manipulée dynamiquement.

```
>>> sys.path.append('/tmp/python')
['', '/home/raphael', '/usr/lib/python2.6', '/usr/lib/python2.6/plat-linux2',
 '/usr/lib/python2.6/lib-tk', '/usr/lib/python2.6/lib-old',
 '/usr/lib/python2.6/lib-dynload', '/usr/lib/python2.6/dist-packages',
 '/var/lib/python-support/python2.6', '/usr/local/lib/python2.6/dist-packages',
 '/tmp/python']
```

`platform` Nom du système d'exploitation.

```
# sous linux
>>> sys.platform
'linux2'

# sous OSX
>>> sys.platform
'darwin'
```

– `ps1, ps2` Variables contenant les valeurs des prompts, par défaut '>>>' et '...'.

Quelques fonctions

`exit([arg])` Mettre fin à l'exécution d'un programme, `arg` étant le statut de sortie pour le système d'exploitation. Cette fonction respecte le nettoyage des clauses `finally` (voir section ref{sec :chap3 :exceptions}).

4.2.2 Le module `string`

Ce module fournit un certain nombre de constantes et de fonctions de manipulation des chaînes de caractères. Il est généralement recommandé d'utiliser les méthodes disponibles sur les objets de type `string` équivalentes.

```
>>> import string
```

Quelques constantes

Des constantes pratiques offertes par le module `string` définissent des ensembles de caractères :

```
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.letters
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
>>> string.digits
'0123456789'
>>> string.whitespace
'\t\n\x0b\x0c\r '
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Les fonctions principales

`lower upper cap*` permettent de gérer la casse d'une chaîne de caractères : mise en minuscules, en majuscules, capitalisation de la phrase ou des mots (avec la réduction des espaces).

```
>>> string.lower('FOO')
'foo'
>>> string.upper('foo')
'FOO'
>>> string.capitalize('foo')
'Foo'
>>> string.capwords(' hello    world! ')
'Hello World!'
```

`strip expandtabs` permettent de gérer les espaces dans une chaîne de caractères en supprimant les blancs non significatifs ou en remplaçant les tabulations par un nombre fixe d'espaces.

```
>>> string.strip('  hello  world! \n ') # [2nd arg]
'hello  world!'
>>> string.expandtabs('\thello world!', 4)
'    hello world!'
```

`find` permet de rechercher un motif dans une chaîne (à partir du début ou de la fin pour `rfind`) en retournant l'indice où le motif est trouvé pour la première fois ou `-1`. (La fonction `index` est similaire mais lève une exception pour un motif non trouvé.)

```
>>> string.find('bonjour le monde', 'on')
1
>>> string.rfind('bonjour le monde', 'on')
12
>>> string.rfind('bonjour le monde', 'om')
-1
```

`split` permet de découper une chaîne en une liste de mots (par défaut sur les espaces, ou alors sur une chaîne passée en second argument) et `join` permet l'opération inverse qui est de construire une chaîne à partir d'une liste de mots et une chaîne de liaison.

```
>>> string.split('foo bar 123')
['foo', 'bar', '123']
>>> string.split("hello world", "wo")
['hello ', 'rld']
>>> string.join(['foo', 'bar', '123'], ';')
'foo;bar;123'
```

`count replace` permettent respectivement de compter les occurrence d'un motif dans une chaîne et le remplacement de ce motif par un autre.

```
>>> string.count('bonjour le monde', 'on')
2
>>> string.replace('bonjour le monde', 'on', 'ONON')
'bONONjour le mONONde'
```

`zfill center rjust ljust` permettent de gérer l'affichage : `zfill` complète un nombre avec des zéros en tête pour une largeur d'affichage constant, les trois autres fonctions permettent de justifier une phrase sur une largeur donnée.

```
>>> string.zfill(str(123), 5)
'00123'
>>> string.center('hi!', 10)
'   hi!    '
>>> string.rjust('hi!', 10)
'       hi!'
>>> string.ljust('hi!', 10)
'hi!       '
```

4.2.3 Le module `re`

Ce module permet la manipulation des expressions régulières [3]. Ces expressions sont par défaut identiques à Perl et l'ouvrage «Mastering Regular Expressions» de *Jeffrey Friedl* (O'Reilly) est un bon complément pour qui doit utiliser les expressions régulières. Afin de ne pas devoir toujours utiliser des '\' dans les expressions, il est possible d'utiliser des *Raw regular expressions* en préfixant la chaîne définissant l'expression par un 'r' : `r'(.*)\n'`.

```
>>> import re
```

Opérations de recherche

`search(pattern, chaine)` permet de rechercher le pattern dans la chaîne et retourne un objet de type `SRE_Match` décrivant la réponse ou bien `None`. La classe `SRE_Match` fournit des méthodes pour obtenir l'indice de début (`start()`) et de fin (`end()`) du motif dans la chaîne. Ici, la méthode `matching` utilise `span()` qui fournit un tuple contenant les deux indices.

```
>>> def matching(res):
...     if res: print res.span()
...     else: print 'no matching'
>>> matching(re.search('ada', 'abracadabra'))
(5, 8)
```

`match(pattern, chaine)` test si la chaîne commence par le pattern et retourne un objet de type `SRE_Match` décrivant la réponse ou bien `None`.

```
>>> res = re.match('abr', 'abracadabra')
>>> matching(res)
(0, 3)
```

Opérations de manipulation

`split(pattern, chaine)` découpe la chaîne par rapport au pattern. L'exemple suivant fait un découpage en mots (tout ce qui n'est pas espace ou ponctuation).

```
>>> re.split('\W+', 'Hello world !')
['Hello', 'world', '']
```

`sub(pattern, repl, chaine)` retourne une chaîne ou toutes les occurrences du pattern dans la chaîne fournie sont remplacés par `repl`. La chaîne n'est pas modifiée, une copie est créée [4].

```
>>> re.sub('hello', 'bonjour', 'hello foo hello bar')
'bonjour foo bonjour bar'
```

3. Voir http ://www.amk.ca/python/howto/regex/
4. En Python les chaînes sont des objets immuables, toute «modification» d'une chaîne entraîne en fait la création d'une nouvelle chaîne contenant la version «modifiée».

Utilisation d'expressions compilées

Lorsqu'une expression régulière est utilisée de manière répétitive dans un programme, il devient intéressant de compiler le motif à rechercher. Les fonctionnalités sont similaires à celles vues précédemment (méthodes) mais la performance est meilleure.

`compile` permet la création d'un objet représentant l'expression régulière sous forme compilée. Un fois l'objet créé, les fonctions sont appelées sur cet objet et non par rapport au module. La méthode `group()` sur l'objet `SRE_Match` donne accès au *i*-ème motif (dans le cas d'une expression composée de plusieurs motifs [5] comme ici) trouvé par la fonction `search`.

```
>>> exp = re.compile('<a href="(.*)">(.*)</a>')
>>> res = exp.search('Cliquer <a href="foo.html">ici</a>!')
>>> matching(res)
(8, 34)
>>> res.group(0)
'<a href="foo.html">ici</a>'
>>> res.group(1)
'foo.html'
>>> res.group(2)
'ici'

>>> exp = re.compile(r'.*<a href="(.*)">.*</a>.*')
>>> exp.sub(r'\1', 'Cliquer <a href="foo.html">ici</a>!')
'foo.html'
```

4.2.4 Le module `os`

Ce module permet d'accéder aux fonctions des systèmes d'exploitation et de faire de la programmation ou de l'administration système. Les fonctions sont disponibles sur (presque) tous les système. Toutefois, il faut toujours être conscient des limites pour la portabilité du code : la notion de droits sur un fichier n'est pas similaire sur tous les systèmes.

Les principaux sous modules de `os` sont :
– `path` manipulation de chemins,
– `glob fnmatch` pattern matching de fichiers et répertoires,
– `time` accès et manipulation de l'heure,
– `getpass` manipulation de mots de passe et identifiants. utilisateur

```
>>> import os
```

Quelques constantes

`name` donne le nom de l'implémentation du module `os` : `posix`, `nt`, `dos`, `mac`, `java`, etc.

5. Un motif au sein d'une expression régulière est défini par des parenthèses.

```
>>> os.name
'posix'
```

`environ` est un dictionnaire contenant les variables d'environnement (au sens Unix mais aussi relatives au lancement de l'interprète Python).

```
>>> os.environ
{'USER': 'raphael', 'HOME': '/home/raphael',
'PATH': '/bin:/usr/bin:/opt/bin:/home/raphael/bin',
'HOSTNAME': 'alfri.lifl.fr',
'PWD': '/home/raphael/enseign/python/scripts'}
```

4.2.5 Le module `os.path`

Ce module fournit quelques primitives (courantes sous les environnements Unix) de manipulation des noms de fichiers et des fichiers.

`basename dirname` permettent respectivement d'extraire d'un chemin le nom du fichier ou le nom du répertoire (ou de l'arborescence dans le cas d'un chemin composé).

```
>>> os.path.basename('/tmp/foo.txt')
'foo.txt'
>>> os.path.dirname('/tmp/foo.txt')
'/tmp'
```

`split join` permettent respectivement de découper et de construire un chemin. L'utilisation de la fonction `join` est fortement recommandée pour construire des chemin car elle respecte implicitement le bon séparateur de chemin pour le système courant.

```
>>> os.path.split('/tmp/foo.txt')
('/tmp', 'foo.txt')
>>> os.path.join('/tmp', 'foo.txt')
'/tmp/foo.txt'
```

`exists isdir isfile` permettent de tester l'existence et le type d'un fichier (ou répertoire).

```
>>> os.path.exists('/tmp/bar.txt')
False
>>> os.path.isdir('/tmp')
True
>>> os.path.isfile('/tmp')
False
```

4.2.6 Les modules `glob` et `fnmatch`

Ces deux modules fournissent principalement une fonction portant le même nom.

`glob` est l'équivalent du `ls` Unix. Cette fonction retourne une liste des fichiers d'un répertoire avec utilisation de *jockers*.

```
>>> import glob
>>> glob.glob('*.py')
['hello.py', 'examples.py']
>>> glob.glob('/tmp/*.tmp')
['/tmp/sv3e2.tmp', '/tmp/sv001.tmp', '/tmp/sv3e4.tmp']
```

`fnmatch` et `filter` permettent de faire du pattern matching sur des chaînes de caractères représentant des noms de fichiers : respectivement est ce qu'un nom suit un pattern de nommage et quels sont les éléments d'une liste respectant un pattern.

```
>>> import fnmatch
>>> fnmatch.fnmatch('examples.py', '*.py')
True
>>> fnmatch.filter(['examples.py', 'hello.pyc'], '*.py')
['examples.py']
```

4.2.7 Le module `getpass`

Ce module permet de demander au système le nom de l'utilisateur connecté et de demander de manière «cachée» un mot de passe.

```
>>> import getpass
```

`getuser()` demande au système le nom de login de l'usager.

```
>>> getpass.getuser()
'raphael'
```

`getpass()` demande proprement (en masquant les caractères saisis) un mot de passe à l'usager. L'appel à cette fonction est bloquant jusqu'à ce que l'usager ait tapé entrée.

```
>>> p = getpass.getpass() # bloquant jusqu'au '\n'
Password:
>>> print p
'quelmauvaismotdepasse'
```

4.3 Built-in en Python

Les *built-in* sont les fonctionnalités câblées en dur dans l'interprète Python, et dans un certain sens l'interprète lui-même. Ces fonctionnalités ne sont pas écrite en Python car elles sont plus que *largement utilisées* et leur mise en {oe}uvre en C contribue à obtenir de meilleures performances.

4.3.1 Les fichiers

Les objets fichiers

Les fichiers sont représentés comme des objets de type `file`. Ils peuvent être textuels ou binaires et être utilisés en lecture ou en écriture.

`open` ouvre un fichier (crée l'objet associé) en lecture par défaut.

```
>>> foo = open('/tmp/foo.txt')
>>> foo
<open file '/tmp/foo.txt', mode 'r' at 0x...>
```

`close` ferme un fichier (mais ne détruit pas l'objet associé).

```
>>> foo.close()
>>> foo
<closed file '/tmp/foo.txt', mode 'r' at 0x...>
```

Lecture dans un fichier

Il y a plusieurs manières de lire un fichier.

`readline()` permet de lire une ligne (position courante jusqu'au prochain `\n`) à la fois dans le fichier.

```
>>> foo = open('/tmp/foo.txt', 'r')
>>> foo.readline()
'hello world!\n'
```

`readlines()` permet de lire toutes les lignes d'un fichier en une seule fois (retourne une séquence de chaînes).

```
>>> foo.readlines()
['bonjour le monde!\n', 'au revoir le monde!\n']
```

`read([n])` permet de lire tout le fichier à partir de la position courante, ou au maximum `n` octets lorsque un argument est donné. Cette fonction retourne une chaîne. La fonction `seek` permet de se déplacer dans le fichier de façon absolue : ici nous retournons au début (indice 0).

```
>>> foo.seek(0) ; foo.read()
'hello world!\nbonjour le monde!\nau revoir le monde!\n'
>>> foo.close()
```

Comparaison des approches de lecture

Il n'y a pas de solution idéale pour la lecture des fichiers. Il faut choisir en fonction de la situation et des besoins.

– La lecture globale (`readlines`) d'un fichier est plus performante pour l'aspect récupération des informations car elle représente un gros accès disque puis un parcours de séquence en mémoire. Toutefois, cette approche est coûteuse en mémoire vive : imaginez la présence d'un fichier texte de 500Mo en mémoire.
– Lecture ligne par ligne (`readline`) est plus coûteuse pour lire un fichier car elle représente de nombreux accès disques pour des petites quantités d'information. Toutefois, cette approche permet de manipuler des gros fichiers : un fichier de 10Go peut être lu ligne par ligne sur une machine ayant 64Mo de mémoire.

Depuis Python 2.3, il est possible d'itérer simplement ligne par ligne sur un fichier ouvert (sans avoir à utiliser la fonction `readline`). Cette manière de faire repose sur les itérateurs (voir la section ref{ssub :chap1 :iterateurs}).

```
>>> fichier = open('/tmp/foo.txt')
>>> for line in fichier:
...     print line.strip()
...
hello world!
bonjour le monde!
au revoir le monde!

>>> fichier.close()
```

Ecriture dans un fichier

Il faut qu'un fichier soit ouvert en écriture pour pouvoir écrire dedans, on le précise donc à l'ouverture en donnant un second paramètre `'w'` (Python suit les modes d'ouverture du langage C, pour les fichiers binaires il faut préciser `'b'` en plus). Tant que le fichier n'est pas fermé, son contenu n'est pas garanti sur le disque.

`write` permet d'écrire des données (une chaîne de texte) représentant une ou plusieurs lignes de texte (utilisation de '`\n`'). Les donnés peuvent aussi être binaires.

```
>>> foo = open('/tmp/foo.txt', 'w')
>>> foo.write('hello world!\n')
```

`writelines` permet d'écrire des données contenues dans une séquence (de chaînes). Si chaque chaîne contenue dans la séquence représente une ligne de texte dans le fichier, il faut alors qu'elles contiennent toute la séquence de fin de ligne '`\n`'. Dans le cas contraire, tout sera écrit dans le fichier, mais la notion de ligne sera définie par les `\n` contenus (même s'ils n'étaient pas en fin d'une des chaînes).

```
>>> lines = ['bonjour le monde!\n', 'au revoir le monde!\n']
>>> foo.writelines(lines)
>>> foo.close()
```

4.3.2 Conversions de types

Utilisation du nom des types pour convertir les chaînes, en entiers, flottants, etc. (et réciproquement.) La fonction `str` permet de convertir tout objet en chaîne.

```
>>> str(123)
'123'
>>> int('123')
123
>>> float('123')
123.0
>>> float(123)
123.0
>>> long('123')
123L
```

4.3.3 Evaluation dynamique

Python permet d'exécuter des commandes à la volée[6] : une chaîne de caractères représentant une commande est exécutée. La chaîne peut être compilée explicitement ou non avant son évaluation. Cette possibilité permet de faire de la génération dynamique de commandes.

`compile()` retourne une version compilée de l'expression (ou du fichier) passé en paramètre. Le second argument précise où doit se faire la sortie d'erreur au cas où une exception serait levée. Le dernier argument précise quel usage est envisagé, ici nous prévoyons de l'utiliser avec la fonction `eval()`.

```
>>> obj = compile('x + 1', '<string>', 'eval')
>>> obj
<code object <module> at 0x... file "<string>", line 1>
```

`eval()` évalue une expression (compilée ou non) passé en paramètre.

```
>>> x = 2
>>> eval('x + 1')
3
>>> eval(obj)
3
```

4.3.4 Assertions

Les assertions permettent de traiter les situations sans appel : soit la condition est respectée, soit le programme est arrêté. (Dans la pratique, une exception est levée.)

`assert` évalue une expression logique et arrête le programme en affichant un message d'erreur (qui est optionnel et fourni séparé par une virgule) si l'expression logique est fausse (cf section ref{sub :chap1 :tests}). L'expression peut inclure des appels de fonctions.

6. Bien que tout est dynamiquement évalué en python, il est possible de faire des choses encore plus dynamiquement, comme de la production de code exécutable à la volée.

```
>>> assert 1
>>> assert 0, 'oups'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError: oups
>>> try: assert 0
... except AssertionError: print 'sic'
...
sic
```

4.4 Exercices

4.4.1 Utilitaire de sauvegarde

En s'aidant de la fonction `os.stat` qui permet de connaître la dates de modification des fichiers, écrire un utilitaire de backup d'une arborescence. Cet utilitaire doit faire un backup incrémental (sauf la première fois) des fichiers modifiés.

```
>>> help(os.stat)
Help on built-in function stat in module posix:

stat(...)
    stat(path) -> stat result

    Perform a stat system call on the given path.
```

4.4.2 Extraction de données

Cet exercice propose d'utiliser des fonctions du module *regular exressions* pour extraire des données d'un fichier texte. Ecrire un programme analysant un fichier texte de type `mbox` et qui produit la liste de toutes les personnes ayant envoyé un e-mail, en triant la liste de leurs adresses par ordre alphabétique avec la date de l'e-mail reçu le plus récent.

CHAPITRE 5

Plongeon dans les objets

5.1 Des objets, rien que des objets

5.1.1 Rappels de principes de base en programmation orientée objet

Instanciation

Un objet est créé à partir d'un moule, sa classe qui définit une structure (les attributs) et un comportement (les méthodes). Une classe définit un type d'objets. Un objet est instance d'une unique classe.

Encapsulation

Les données sont «cachées» au sein des objets, et leur accès est contrôlé par les méthodes. L'état d'un objet ne devrait pas être manipulé directement si un contrôle est nécessaire. L'objet peut être vu comme un fournisseur de services (plus que de données).

Encapsulation n'implique pas attributs privés comme dans certains langages de programmation. Un attribut privé ne protège pas d'une utilisation abusive, mai considère le développeur comme un danger. Python prend une approche différente : le développeur a un cerveau, et donc, faisons lui confiance.

Polymorphisme

Des objets respectant une même interface (la signature de méthodes suffit dans le cas de Python) peuvent être manipulés de manière générique, même si leur type exact est différent. Ce principe permet aussi la substitution d'une instance par une autre (tant que les interfaces sont compatibles).

Héritage

Mécanisme qui permet la réutilisation de définitions de base, comportements par défaut, et la spécialisation de certains comportements. La relation d'héritage entre deux classes ne se limite pas à une facilité de capitalisation de code. La classe qui hérite doit «être un» de la classe dont elle hérite : un chat «est un» animal, chat peut donc être une sous classe de animal. Mais une voiture ne doit pas hériter d'un moteur : une voiture n'est pas un moteur elle contient un moteur.

5.1.2 Objets et références

En Python, le monde est uniforme.
– *Tout est objet* : les chaînes, les entiers, les listes, les fonctions, les classes, les modules, etc. Tout est réifié et manipulable dynamiquement.
– *Tout objet est manipulé par référence* : une variable contient une référence vers un objet et un objet peut être référencé par plusieurs variables.
– Une fonction, une classe, un module sont des *espaces de nommage organisés de manière hiérarchique* : un module contient des classes qui contiennent des fonctions (les méthodes).

5.1.3 Classes

Définition et instanciation

La définition d'une classe suit la règle des blocs (cf section ref{sub :chap1 :blocs}) en utilisant le mot clé `class`. (Voir la section sur l'unification des types et des classes pour une explication sur l'héritage explicite de `object`).

```
>>> class Empty(object):
...     pass
```

Toute méthode est définie comme une fonction avec un premier argument (`self`) qui représente l'objet sur lequel elle sera appliquée à l'exécution. En Java ou C++, le `this` est définie implicitement, en Python il est explicite est c'est toujours le premier paramètre d'une méthode. Le nom est libre, mais on utilise en général `self`. Une méthode ne prenant pas de paramètres aura donc quand même un argument.

```
>>> class Dummy(object):
...     def hello(self):
...         print 'hello world!'
```

L'instanciation se fait sans mot clé particulier (il n'y a pas de `new` en Python). Il suffit de faire suivre un nom de classe de parenthèses (contenant ou non des paramètres) pour déclencher une instanciation. L'invocation se fait à l'aide de la notation pointée, un appel de méthode sur une variable référençant un objet.

```
>>> d = Dummy()
>>> d
<Dummy object at 0x...>
```

```
>>> d.hello()
hello world!
```

Définition d'attributs et du constructeur

Les attributs sont définis à leur première affectation. Une variable est considérée comme un attribut si elle est «rattachée» à l'objet : elle est accédée par `self`. Tout accès à un attribut ou à une méthode de l'objet depuis sa mise en oeuvre se fait obligatoirement par la variable `self`. Le constructeur est une «méthode» nommée `__init__`. Comme toute méthode, certain de ses paramètres peuvent avoir des valeurs par défaut. Cette possibilité est important car une classe a un seul constructeur en Python (contrairement à d'autre langages).

```
>>> class Compteur(object):
...     def __init__(self, v=0):
...         self.val = v
...     def value(self):
...         return self.val
```

En termes de génie logiciel (et de bon sens), il est recommandé de toujours initialiser l'état des objets (ses attributs) lors de son instanciation, donc dans le constructeur.

Visibilité des attributs et méthodes

Les attributs, comme les méthodes, ont par défaut une visibilité «publique» en Python. Dans le cas des instances de la classe `Compteur`, il est possible de manipuler directement la valeur de l'attribut (ou méthodes) `val`.

Une convention pour certains développeurs Python est de préfixer les attributs par un sougliné '_' pour annoncer qu'ils ne sont pas destinés à être utilisés par les clients. Il est à noter que c'est une convention et que rien n'empêche un utilisateur de lire un attribut dont le nom est préfixé par un souligné [1].

Enfin, l'utilisation de deux soulignés comme préfixe (et pas de souligné comme postfixe) d'un nom d'attribut (ou de méthode) permet de définir un attribut «privé» et donc non visible par les clients de la classe.

```
>>> class Foo(object):
...     def __init__(self):
...         self._a = 0
...         self.__b = 1
...
>>> f = Foo()
>>> f._a
0
>>> f.__b
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: Compteur instance has no attribute '__b'
```

1. Il est vrai que dans le cas de Python la notion de «protégé» repose sur la confiance, mais si vous roulez à 180 km/h en ville qui est responsable, la voiture ou vous ?

Note

La philosophie de Python est de faire confiance au programmeur plutôt que de vouloir tout «blinder» comme dans d'autres langages (qui ne blindent pas grand chose au final).

5.1.4 Héritage

Python supporte l'héritage simple et l'héritage multiple. Dans une relation d'héritage, il est faut préciser le nom de la classe mère (entre parenthèses après le nom de la classe en cours de définition) **et** appeler explicitement le constructeur de la super classe.

```
>>> class A(object):
...     def __init__(self, n='none'):
...         self._name = n
...     def name(self):
...         return self._name

>>> class B(A):
...     def __init__(self, val=0, n='none'):
...         A.__init__(self, n)
...         self._wheels = val
...     def wheels(self):
...         return self._wheels
```

Dans le cas de l'héritage multiple, il suffit de préciser les différentes classes mères et d'appeler leurs constructeurs respectifs.

```
>>> class C(object):
...     def __init__(self, t=''):
...         self._title = t
...     def title(self):
...         return self._title

>>> class D(A, C):
...     def __init__(self, n='none', t=''):
...         A.__init__(self, n)
...         C.__init__(self, t)
...     def fullname(self):
...         return self._title + ' ' + self._name
```

L'exemple suivant donne une utilisation des trois classes précédentes. Il n'y a rien de particulier à l'utilisation que la classe ait été définie entièrement ou par héritage ne change rien.

```
>>> a = A('raphael')
>>> print a.name()
raphael
>>> b = B(4, 'car')
>>> print b.name()
car
>>> print b.wheels()
```

```
4
>>> d = D(t='dr')
>>> print d.fullname()
dr none
```

Héritage multiple problématique

L'héritage multiple pose problème lorsque deux classes héritées offre une méthode portant le même nom. Dans ce cas, il est possible de préfixer les méthodes par le nom de la classe qui les définies ou d'utiliser des alias de méthodes pour résoudre les conflits.

```
>>> class X(object):
...     def name(self):
...         return 'I am an X'

>>> class Y(object):
...     def name(self):
...         return 'I am an Y'

>>> class Z(X, Y):
...     xname = X.name
...     yname = Y.name
...     def name(self):
...         return 'I am an Z, ie ' + self.xname() + \
...             ' and ' + self.yname()
```

L'exemple suivant propose une utilisation des trois classes `X`, `Y`, `Z` précédentes avec une invocation de la méthode `name()` sur les trois instances créées.

```
>>> for clss in [X, Y, Z]:
...     obj = clss()
...     print obj.name()
...
I am an X
I am an Y
I am an Z, ie I am an X and I am an Y
```

5.1.5 Classes *vs* modules

Il semblerait que la vision suivante doit bien acceptée de la communauté de développeurs Python :

- Utiliser des classes lorsque
  - Les données représentent un état et doivent être protégées.
  - Les traitements et données sont fortement liés (les données évoluent).
- Utiliser des modules de fonctions lorsque
  - Les traitements et données sont «indépendants» (ex : data mining).
  - Les données sont stockées dans des BD ou fichiers, et non modifiées.

5.2 Structures des objets

5.2.1 Introspection simple

La fonction `dir()` donne le contenu de tout objet (liste de ses attributs et méthodes). Associée à l'attribut `__doc__`, cela fournit une documentation de base sur tout objet. D'où l'intérêt de bien auto-documenter le code (voir section ref{sub :chap1 :doc}).

```
>>> dir(Compteur)
['__class__', ... '__init__', ... 'value']
>>> c = Compteur()
>>> dir(c)
['__class__', ... '__init__', ... 'val', 'value']
```

La fonction `type()` donne quant à elle le type d'une référence.

```
>>> type(c)
<class 'Compteur'>
>>> type([1, 2])
<type 'list'>
```

5.2.2 Classes et attributs

Une classe ne contient que des attributs. Elle peut être vue comme un simple dictionnaire contenant des associations nom / référence. Une fonction de la classe (ou méthode) est en fait un attribut qui est appelable (*callable*).

```
>>> class OneTwoThree(object):
...     value = 123             # reference un entier
...     def function(self):     # reference une fonction
...         return self.value
...
>>> ott = OneTwoThree()
>>> dir(ott)
['__class__', ... '__init__', ... 'function', 'value']
>>> ott.function
<bound method OneTwoThree.function of <OneTwoThree object at 0x...>>
```

Dans le cas d'un conflit de nom entre un attribut et une méthode, la priorité va à l'attribut. Ici, l'interpréteur explique que l'on ne peut utiliser l'attribut `name` comme si c'était une fonction (ou méthode) car c'est une chaîne, donc on ne peut demander son exécution (`object is not callable`). La méthode `name` est masquée par l'attribut portant le même nom car celui-ci sera initialisé après la définition de la méthode.

```
>>> class Conflict(object):
...     def __init__(self):
...         self.name = 'Conflict'
...     def name(self):
```

```
...         return 'You will never get this string!'
...
>>> c = Conflict()
>>> c.name()
Traceback (most recent call last):
  ...
TypeError: 'str' object is not callable
```

Dans ce cas, L'utilisation d'attributs «protégés» est donc une bonne idée qui a pour conséquence d'éviter les conflits de noms entre attributs et méthodes (même si dans le cas présent, je reconnais que l'existence d'une méthode name est discutable).

```
>>> class NoConflict(object):
...     def __init__(self):
...         self._name = 'NoConflict'
...     def name(self):
...         return self._name
...
>>> c = NoConflict()
>>> print c.name()
NoConflict
```

Il est possible de définir dynamiquement des attributs sur une classe ou une instance. Ce genre de facilité ne se limite pas à l'aspect ludique, c'est parfois la meilleure solution à un problème. Toutefois, attention où l'on met les pieds lorsque l'on modifie dynamiquement des objets.

```
>>> class Empty(object):
...     pass
>>> Empty.value = 0
>>> def funct(self):
...     self.value += 1
...     return self.value
...
>>> Empty.funct = funct
>>> dir(Empty)
['__class__', ... '__init__', ... 'funct', 'value']
>>> e = Empty()
>>> e.funct()
1
```

Quelques attributs notoires

- `__doc__` contient la documentation de la classe, de l'objet, de la fonction, du module, etc.
- `__module__` contient le nom du module contenant la définition de la classe, de l'objet, de la fonction, du module, etc.
- `__name__` contient le nom de la fonction ou de la méthode.
- `__file__` contient le nom du fichier contenant le code du module.

5.3 Les objets, version avancée

5.3.1 Un peu de réflexion

Les techniques de réflexion permettent de découvrir (introspection) et de manipuler dynamiquement et automatiquement les attributs (et fonctions) d'un objet. Ces techniques sont utilisables, par exemple, pour le chargement de code dynamique comme la mise en oeuvre d'un mécanisme de plug-ins. Elles sont aussi couramment utilisés dans les environnements de développement qui font de la complétion automatique (il faut bien aller chercher l'information quelque part).
Il existe trois fonctions de base en Python :
– `hasattr()` test si un attribut existe,
– `getattr()` donne accès à la valeur d'un attribut,
– `setattr()` fixe la valeur d'un attribut (avec création si l'attribut est inexistant).
Dans l'exemple suivant, nous manipulons l'instance étendue de la classe `Empty` : après avoir testé son existence, nous récupérons dynamiquement la méthode nommée `funct` pour en demander ensuite l'exécution.

```
>>> hasattr(e, 'funct')
True
>>> f = getattr(e, 'funct')
>>> f
<bound method Empty.funct of <Empty object at 0x...>>
>>> f()
2
```

Il est possible de faire la même chose en demandant à la classe la fonction `funct`. Mais, dans ce cas, pour demander son exécution il faut passer l'objet `e` sur lequel doit s'appliquer l'appel. Cette version illustre bien le pourquoi du premier argument (`self`) des méthode.

```
>>> hasattr(Empty, 'funct')
True
>>> f = getattr(Empty, 'funct')
>>> f
<unbound method Empty.funct>
>>> f(e)
3
```

Enfin, nous définissons un nouvel attribut `name` dont la valeur est fixée à `myempty`.

```
>>> setattr(e, 'name', 'myempty')
>>> e.name
'myempty'
```

5.3.2 Un peu plus de réflexion avec `inspect`

Le module `inspect` offre des moyens supplémentaires pour l'introspection, par exemple découvrir ce que contient une classe, un objet ou un module.

```
>>> import inspect
```

L'exemple suivant récupère les membres de la classe `E` puis les membres d'une instance de cette classe. Pour chaque membre de l'objet (classe ou instance), un tuple est fourni contenant le nom de l'attribut et sa valeur. Nous pouvons constater que la différence se limite à l'état de la fonction `f` : non liée dans le cas de la classe, donc que l'on ne peut exécuter directement, et liée à l'instance dans le cas de l'objet que l'on peut donc exécuter directement après un `getattr`.

```
>>> class E(object):
...     def f(self):
...             return 'hello'
...
>>> e = E()

>>> inspect.getmembers(E)
[('__class__', <type 'type'>), ... ('f', <unbound method E.f>)]

>>> inspect.getmembers (e)
[('__class__', <class 'E'>), ... ('f', <bound method E.f of <E object at 0x...>>
```

Ce module permet aussi de savoir ce que l'on est en train de manipuler : quel est le type d'un objet (classe, instance, attribut). La fonction `ismethod()` permet de savoir si un objet donné est, ou non, une méthode (liée ou non).

```
>>> inspect.isclass(E)
True
>>> f = getattr(e, 'f')
>>> inspect.isfunction(f)
False
>>> inspect.ismethod(f)
True
>>> F = getattr(E, 'f')
>>> inspect.ismethod(F)
True
```

L'association de l'introspection de base et du module `inspect` permet d'automatiser l'utilisation des méthodes d'une instance : invocation générique d'une méthode récupérée dynamiquement après contrôle que c'est bien une méthode. L'exemple suivant montre aussi les deux manières d'invoquer une méthode, et la signification du `self` : `a.foo()` est en fait traduit en `A.foo(a)` par l'interpréteur (considérant que `a` est une instance de la classe `A`).

```
>>> f1 = getattr(e, 'f')
>>> f2 = getattr(E, 'f')
>>> if inspect.ismethod(f1):
...     f1()  # 'f1' est liee a 'e'
...
'hello'
>>> if inspect.ismethod(f2):
...     f2(e)  # 'f2' n'est pas liee, argument 1 == self
...
'hello'
```

Le module `inspect` permet même d'accéder dynamiquement au code source d'un objet. Toutefois, cela n'est pas valable pour le code saisi en interactif (ce qui est somme toute normale vu qu'il n'est pas stocké dans un fichier).

```
>>> from examples import Conflict
```

Quelques fonctions de inspect

La fonction `getfile()` donne le nom du fichier de définition de l'objet (TypeError si cette opération est impossible).

```
>>> inspect.getfile(Conflict)
'examples.py'
```

La fonction `getmodule()` donne le nom du module définissant l'objet (sans garantie par exemple dans le cas du code dynamiquement créé).

```
>>> inspect.getmodule(Conflict)
<module 'examples' from 'examples.py'>
```

La fonction `getdoc()` retourne la documentation de l'objet.

```
>>> inspect.getdoc(Conflict)
'Illustration de conflit attribut / methode'
```

La fonction `getcomments()` retourne la ligne de commentaire précédant la définition de l'objet.

La fonction `getsourcelines()` donne un tuple contenant la liste des lignes de code source définissant l'objet passé en paramètre et la ligne du début de la définition dans le fichier.

```
>>> lines, num = inspect.getsourcelines(Conflict)
>>> for i, l in enumerate(lines):
...     print num + i, l
...
34 class Conflict(object):
35     '''Illustration de conflit attribut / methode'''
36     def __init__(self):
37         self.name = 'Conflict'
38     def name(self):
39         return 'You will never get this string!'
```

La fonction `getsource()` est similaire à la précédente, mais retourne uniquement le code de la définition sous la forme d'une chaîne de caractères.

5.4 Les exceptions en python

5.4.1 Définition et levée

Les exceptions sont des objets comme les autres, donc définies par des classes comme les autres (ou presque). Elles sont simplement un penchant pour ne rien faire (les classes d'exception contiennent rarement des traitements). La définition minimale d'une classe pour une exception revient à étendre la classe de base `Exception`.

```
>>> class MonErreur(Exception):
...     pass
...
```

La mot-clé `raise` permet de lever une exception (définie par l'usager ou standard). Si l'on souhaite associé de l'information à une exception, il suffit de faire suivre le type de l'exception d'un message (séparé par une virgule). [2]

```
>>> raise MonErreur
Traceback (most recent call last):
  ...
MonErreur
>>> raise MonErreur()
Traceback (most recent call last):
  ...
MonErreur
>>> raise NameError, 'cela coince'
Traceback (most recent call last):
  ...
NameError: cela coince
```

Une exception peut aussi contenir des attributs et méthodes (en général pour fournir de l'information). Ici, la méthode standard `__str__` (traduction de l'objet en chaîne) et utilisée pour accéder facilement à la valeur contenue dans l'exception (par défaut le mécanisme d'exception appelle cette méthode). Pour fournir de l'information à une exception, il suffit de l'instancier en passant des paramètres au constructeur lors de l'utilisation de `raise`.

```
>>> class MonErreurToo(Exception):
...     def __init__(self, val=None):
...         self._value = val
...     def __str__(self):
...         return str(self._value)
...
>>> raise MonErreurToo(12)
Traceback (most recent call last):
  ...
MonErreurToo: 12
```

2. Les exceptions sont levées comme des objets. Une classe est définie par un objet dans l'interpréteur. Donc la définition de la classe peut être levée comme une exception. Ce qui n'est pas si choquant pour les cas simple (exception avec message d'erreur).

5.4.2 Traitement des exceptions

Il y a deux possibilités complémentaires pour le traitement des exceptions en Python :
– quoi faire en cas de problème,
– que faut il toujours faire même s'il y a un problème.
Gestion des problèmes, le `try ... except` permet en cas de levée d'exception, de pouvoir tenter de traiter le problème. Il peut y avoir plusieurs clauses `except` pour traiter différents cas exceptionnels. La clause `else` est optionnelle est permet de faire un traitement si tout s'est bien passé (le `else` et le `except` sont en exclusion mutuelle). Enfin, la clause optionnelle `finally` permet d'effectuer un traitement qu'il y ait eu ou non levée d'exceptionfootnote{L'utilisation conjointe des clauses `except` et `finally` est possible depuis la version 2.5. Dans les version précédentes il faut imbriquer les des constructions.}.

```
try:
    <instructions>
(except <exception>:
    <instructions traitement exception>)+
[else:
    <instructions si traitement correct>]
[finally:
    <instructions dans tous les cas>]
```

L'utilisation de la clause `else` est intéressante car elle permet d'isoler la ligne du bloc `try` qui peut lever une exception des lignes qui en dépendent.

L'identification d'une `<exception>` peut prendre plusieurs formes :
– `ExceptionType` est utile lorsque seul le type de l'exception est important, par exemple parce qu'elle ne contient pas d'information~ ;
– `ExceptionType as variable` permet de récupérer l'exception du type spécifié dans `variable` et donc d'exploiter les données et traitements de l'exception.

Lecture d'un fichier texte

Il est très *pythonesque* d'utiliser la construction ouverture d'une ressource, utilisation dans un bloc `try` et libération de la ressource dans un bloc `finally`. On ouvre le fichier puis quoi qu'il arrive on ferme le fichier par la suite. Un fichier devrait toujours être fermé après utilisation.

```
>>> fichier = open('/tmp/foo.txt')
>>> try:
...     for line in fichier.readlines():
...         print line,
... finally:
...     fichier.close()
...
hello world!
bonjour le monde!
au revoir le monde!
```

Gestion de plusieurs exceptions

L'affichage va dépendre de l'exception qui est levée (elle même levée de manière aléatoire).

```
>>> import random
>>> class Impair(Exception): pass
>>> class Pair(Exception): pass

>>> try:
...     if int(random.random() * 100) % 2:
...         raise Impair()
...     else:
...         raise Pair()
... except Pair:
...     print "c'est un nombre pair"
... except Impair:
...     print "c'est un nombre pair"
...
c'est un nombre pair
```

Attention

Un `return` dans le `finally` masque un `return` dans le `try`.

5.4.3 Traitement d'exceptions et héritage

Petit problème courant, attention à l'ordonnancement des `except` ! Il faut toujours, dans le cas où les exceptions sont définies par une relation d'héritage, commencer par traiter les exceptions les plus spécifiques. Dans le cas contraire, la gestion des exceptions les plus générales masque les traitements relatifs aux exceptions les plus spécifiques qui ne sont jamais utilisés.

```
>>> class A(Exception): pass
>>> class B(A): pass

>>> for cls in [A, B]:
...     try: raise cls()
...     except B: print 'B',
...     except A: print 'A',
A B
>>> for cls in [A, B]:
...     try: raise cls()
...     except A: print 'A',
...     except B: print 'B',
A A
```

5.5 Toujours à propos des objets

Cette section est pour certain aspects plutôt à lire une fois que l'on est familier avec Python et avec la notion d'objet.

5.5.1 Unification des types et des classes

Depuis Python 2.2, les types et les classes ont été unifiés en Python. C'est-à-dire qu'il n'y a plus de différence fondamentale entre les types de base et les classes définies par l'utilisateur. Le bénéfice de cette unification est qu'il est désormais possible de définir des classes héritant des types de base.

Suite à cette unification, et pour ne pas briser la compatibilité avec les programmes existants, la déclaration des classe a un peu évolué (c'est la version que nous avons utilisé jusqu'ici). Pour préciser qu'une classe est définie avec la nouvelle forme, il faut la faire hériter de `object`, d'un type de base ou d'une classe héritant de `object`). Attention, toutes les classes de la librairie standard ne suivent pas toute la nouvelle forme et sont donc considérées comme «différentes».

L'exemple suivant présente les deux formes de déclaration de classes simultanément. L'utilisation de la fonction `type` sur des instances de ces deux classes illustre bien la différence de statut vis-à-vis de l'interpréteur.

```
>>> class Foo:
...     pass
...
>>> class Bar(object):
...     pass
...
>>> f = Foo()
>>> b = Bar()
>>> type(f)
<type 'instance'>
>>> type(b)
<class 'Bar'>
```

Avec la nouvelle forme, l'utilisation de méthodes fournies par les super-classes est aussi modifiée. Lors de la redéfinition d'une méthode dans une sous-classe, la fonction `super` permet d'accéder à la définition (originale) de cette méthode dans la super-classe (ou une des super-classes).

La fonction `super` prend comme premier paramètre le type à partir duquel doit se faire la recherche [3] et comme second paramètre `self`. Contrairement à l'ancienne forme, il n'est plus nécessaire de passer `self` en premier argument de la méthode (voir la section ref{sub :chap3 :heritage}). La fonction `super` est aussi utilisable dans la définition des constructeurs.

```
>>> class Bar(object):
...     def bar(self):
...         print 'Bar.bar'
...
>>> class BarToo(Bar):
...     def bar(self):
...         print 'BarToo.bar'
...         super(BarToo, self).bar()
...
```

3. Dans le cas d'une hiérarchie à plusieurs niveaux, la recherche de la méthode hérité ne se fait pas nécessairement à partir de la super-classe, mais elle peut se faire à partir de la super-super-classe par exemple.

```
>>> f2 = BarToo()
>>> f2.bar()
BarToo.bar
Bar.bar
```

5.5.2 Définition de propriétés

Avec l'unification classe / type, la notion de propriété est apparue en Python. Une propriété permet de manipuler un attribut à l'aide d'accesseurs de manière transparente (c'est-à-dire que l'on ne dit pas explicitement que l'on utilise les accesseurs). Une propriété est définie à l'aide de la fonction `property` dans le contexte d'une classe qui étend (même indirectement) `object`.

La fonction `property` prend en paramètre la fonction pour la lecture, la fonction pour l'écriture, la fonction pour la suppression de la valeur associée et une chaîne de documentation. Tous les paramètres sont optionnels (eh oui, il est même possible de définir une propriété qui ne fait rien pas même être lue).

Dans l'exemple suivant, la classe `A` est définie avec une propriété `value` en lecture et écriture : elle est définie avec une méthode pour la lecture et une pour l'écriture. La classe `B` est quant à elle définie avec une propriété `value` en lecture seule : seule une méthode de lecture est fournie.

```
>>> class A(object):
...     def __init__(self, val=0):
...         self._value = val
...     def get_value(self):
...         return self._value
...     def set_value(self, val):
...         self._value = val
...     value = property(get_value, set_value)
...
>>> class B(object):
...     def __init__(self, val):
...         self._value = val
...     def get_value(self):
...         return self._value
...     value = property(get_value)
...
```

Une propriété est utilisée de la même manière qu'un attribut. Dans le cas courant l'utilisation est peu différente d'un attribut classique, même les méthode de lecture / écriture pourrait très bien aller chercher l'information dans un fichier ou autre. Enfin, la tentative d'écriture la propriété `value` sur une instance de `B` se solde par une exception. Il est donc intéressant d'utiliser des propriétés pour faire du contrôle d'accès sur des attributs.

```
>>> a = A(1)
>>> a.value
1
>>> a.value = 2
>>> a.value
```

```
2
>>> b = B(2)
>>> b.value
2
>>> b.value = 1
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

Note

Le choix entre un attribut "publique" et une propriété se résume à : "est ce une valeur calculée ?" Si une valeur est stockée, utiliser un attribut, si elle doit être calculée, utiliser une propriété. On retrouve ici une illustration de la philosophie de Python : faire confiance.

Pour ne pas trop «polluer» la classes, il est courant de définir en Python les fonctions utilisées par les propriétés comme des fonctions internes. Une fois l'attribut `value` définie comme une propriété, la fonction `value` n'est plus visible dans la définition de la classe.

```
>>> class Dummy(object):
...     def __init__(self, value, tax=1.196):
...         self._value = value
...         self.tax = tax
...     def value():
...         doc = "The value with taxes."
...         def fget(self):
...             return self._value * self.tax
...         def fset(self, value):
...             self._value = value
...         return locals()
...     value = property(**value())
...
>>> d = Dummy(20)
>>> d.value
23.919999999999998
```

En utilisant cette approche la documentation n'est plus polluée.

```
>>> help(Dummy)
Help on class Dummy in module __main__:

class Dummy(__builtin__.object)
 |  Methods defined here:
 |
 |  __init__(self, value, tax=19.600000000000001)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  value
 |      The value property.
```

5.5.3 Décorateurs

Le concept de décorateur de fonction (ou de méthode) a été introduit dans la version 2.4 (PEP 318). La notion de décorateur n'est pas des plus simple à expliquer, mais on peut la voire comme une fonction qui applique des traitements sur une autre fonction pour en changer le comportement. Un décorateur peut donc être toute fonction prenant en argument une fonction (ou tout objet «appelable»). L'utilisation d'une fonction comme décorateur d'une autre fonction se fait en utilisant le nouvel opérateur `@`. En considérant un décorateur `decorateur` et une fonction `foo` les deux déclarations suivantes sont équivalentes.

```
@decorateur
def foo():
    pass

def foo():
    pass
foo = decorateur(foo)
```

Python propose quelques décorateurs par défaut par exemple pour les propriété et pour les méthodes de classe. Le décorateur de propriété ne peut être utilisé que pour une propriété en lecture seule.

```
>>> class B(object):
...     def __init__(self, val):
...         self._value = val
...     @property
...     def value(self):
...         return self._value

>>> b = B(2)
>>> b.value
2
>>> b.value = 1
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

L'exemple suivant prend le grand classique de la trace des appel de fonction pour illustrer la définition et l'utilisation d'un décorateur. La fonction `trace` définie une fonction d'emballage (un *wrapper*) `_trace` qui va, en plus du comportement normal de la fonction qu'elle emballe, afficher le nom et les arguments de celle-ci. L'utilisation de `trace` permet donc de substituer la définition de base d'une fonction par une version incluant la trace : `trace` retourne tout simplement une fonction.

Notre fonction `trace` va être utilisable sur toute fonction comme un décorateur pour afficher sur la sortie standard les appels à cette fonction. Ici, nous traçons la fonction `bar`, dont la définition est tout ce qu'il y a de plus normal, si ce n'est la déclaration `@trace`.

```
>>> def trace(fct):
...     def _trace(*args):
...         print '<entering %s%s >' % (fct.__name__, args)
```

```
...         return fct(args)
...     return _trace
...
>>> @trace
... def bar(name):
...     print 'hello %s' % name
...
>>> bar('raphael')
<entering bar('raphael',) >
hello raphael
```

Il est à noter qu'une fonction (ou méthode) peut être décorée plusieurs fois.

5.6 Exercices

5.6.1 Premières classes

Implanter des classes `Pile` et `File` utilisant les liste comme structure de données interne et respectant l'*interface* suivante (utiliser l'héritage).

```
class Base(object):
    def pop(self):
        pass
    def push(self, elt):
        pass
```

Reprendre l'exercice ref{sub :chap2 :exo2} en implantant les fonctionnalités au sein d'une classe.

5.6.2 Design pattern état

L'état d'un objet `a` de type `A` est implanté par plusieurs classes. L'instance de ces classes qui représente l'état de l'objet varie au cours du cycle de vie de l'objet. Les méthodes de l'objet sont déléguées à l'instance représentant l'état à un moment donné.

Implanter ce *design pattern* pour représenter un feu tricolore (une classe d'état par couleur, une méthode pour changer la couleur du feu et une méthode qui donne le droit de passage).

CHAPITRE 6

Python et XML

6.1 XML, avec une vision DOM

6.1.1 Langages de balises

- Au commencement était SGML
  - *Standard Generalized Markup Language*
  - Simplifié énormément avec HTML pour le Web
  - Transformé en produit marketing avec XML
- Python supporte toute la petite famille
  - HTML, XHTML et SGML pour HTML
  - XML, DOM et SAX

6.1.2 XML, quels outils ?

Il existe plusieurs librairies pour manipuler des documents XML avec Python, mais ce chapitre se limite à une partie du package standard `xml`. Ce package fournit plusieurs sous-packages :
- `dom` contient `minidom`, une implémentation de DOM en Python,
- `sax` est un dictionnaire des implémentation disponibles,
- `parsers` contient les parsers utilisés en interne par DOM et SAX.

En compléments, plusieurs librairies et extensions sont disponibles. Elles fournissent, par exemple, des abstractions de plus haut niveau et des outils typés par rapport à une utilisation donnée.

6.1.3 DOM, petit rappel

Pour ce qui est de la manipulation d'un fichier XML, nous allons uniquement utiliser DOM (Document Object Model). DOM propose de manipuler un document XML en mémoire comme un arbre d'objets représentant les noeuds du document. Les interfaces standard DOM sont les suivantes.

Interface	Représentation de
Node	Interface de base des noeuds
NodeList	Séquence de noeuds
Document	Représente un document complet
Element	Elément de la hiérarchie
Attr	Noeud représentant un attribut d'un noeud
Comment	Noeud repréntant un commentaire
Text	Noeud de données textuelles

6.1.4 Exemple du chapitre

Le listing suivant présente un exemple de document XML utilisé dans la suite du chapitre pour illustrer la manipulation de tels documents.

```
<?xml version="1.0" ?>
<contacts>
  <contact name="doe" firstname="john">
    <address>
      <road value="10, binary street" />
      <postal value="0001" />
      <city value="cpu" />
    </address>
    <programming lang="asm" />
  </contact>
  <contact name="dupont" firstname="jean">
    <address>
      <road value="impasse de l'assembleur" />
      <postal value="0100" />
      <city value="dram" />
    </address>
    <programming lang="c" />
  </contact>
  <contact name="terprette" firstname="quentin">
    <address>
      <road value="avenue du script" />
      <postal value="1001" />
      <city value="salt snake city" />
    </address>
    <programming lang="python" />
  </contact>
</contacts>
```

6.2 Naviguer dans un arbre DOM

6.2.1 `minidom` «il fait le maximum»

Le module `minidom` est une mise en oeuvre de DOM pour Python. Il fournit toutes les interfaces de base DOM et un parser de fichier (ou de chaîne) XML. Il est disponible en standard dans l'environnement (librairie PyXML).

6.2.2 Parser un document XML

La fonction `minidom.parse` permet d'analyser un fichier XML et d'obtenir un arbre DOM correspondant. Elle retourne un objet de type `Document`. En une ligne, elle est pas belle la vie ?

```
>>> from xml.dom import minidom
>>> doc = minidom.parse('/tmp/contacts.xml')
>>> doc
<xml.dom.minidom.Document instance at 0x...>
```

6.2.3 Parcourir un arbre DOM

Un certain nombre d'opérations et d'attributs disponibles sur tous les éléments d'un arbre DOM pour en permettre son parcours.

La fonction `hasChildNodes()` indique si un noeud a des fils.

```
>>> doc.hasChildNodes()
True
```

L'attribut `childNodes` donne accès aux fils d'un noeud (sous la forme d'une liste).

```
>>> doc.childNodes
[<DOM Element: contacts at 0x...>]
>>> doc.childNodes[0].hasChildNodes()
True
```

L'attribut `documentElement` fournit le noeud racine d'un document DOM.

```
>>> root = doc.documentElement
```

Les attributs `firstChild, lastChild` donnent accès respectivement au premier et au dernier fils d'un noeud.

```
>>> root.childNodes
[<DOM Text node "u'\n  '">, <DOM Element: contact at 0x...>, <DOM Text node "u'\
>>> root.firstChild
<DOM Text node "u'\n  '">
>>> root.lastChild
<DOM Text node "u'\n'">
```

Les attributs `nextSibling, previousSibling` donnent respectivement accès au fils suivant et au fils précédent (ou `None` si plus de fils). Le calcul se fait par rapport à une racine commune.

```
>>> current = root.firstChild
>>> while current:
...     print current
...     current = current.nextSibling
...
<DOM Text node "u'\n  '">
<DOM Element: contact at 0x...>
<DOM Text node "u'\n  '">
<DOM Element: contact at 0x...>
<DOM Text node "u'\n  '">
<DOM Element: contact at 0x...>
<DOM Text node "u'\n'">
```

L'attributs `parentNode` donne accès au parent (direct) d'un noeud.

```
.. doctest::

>>> root
<DOM Element: contacts at 0x...>
>>> root.firstChild
<DOM Text node "u'\n  '">
>>> root.firstChild.parentNode
<DOM Element: contacts at 0x...>
```

La fonction `isSameNode()` permet de tester si deux noeuds sont égaux (égalité au sens de l'identité, parle-t-on de la même personne).

```
>>> root.firstChild.isSameNode(root.lastChild)
False
>>> root.firstChild.isSameNode(root.childNodes[0])
True
```

6.2.4 Recherche dans un arbre DOM

La recherche dans un arbre DOM se fait principalement par nom de tag (nom de noeud). La fonction `getElementsByTagName()`, disponible sur les noeuds de type `Document` et `Element`, donne tous les fils (et sous-fils et descendants) portant le nom de tag fourni.

```
>>> for element in root.getElementsByTagName('contact'):
...     print element
<DOM Element: contact at 0x...>
<DOM Element: contact at 0x...>
<DOM Element: contact at 0x...>

>>> for element in root.getElementsByTagName('programming'):
...     print element
```

```
<DOM Element: programming at 0x...>
<DOM Element: programming at 0x...>
<DOM Element: programming at 0x...>
```

6.2.5 `NodeList` et objets séquences

Le résultat retourné par les fonctions `childNodes` et `getElementsByTagName` est de type `NodeList` et se comporte comme une séquence «classique».

> Pour se comporter comme une séquence, un objet doit offrir (au minimum) les méthodes suivantes :
> – `__len__()` et `__getitem__(i)` pour l'accès aux éléments (boucle `for`),
> – `__setitem__(i)` et `__delitem__(i)` pour les séquences modifiables.

6.3 Accéder aux informations d'un noeud

6.3.1 Informations liées au noeud

L'attribute `nodeType` donne une constante (1-10) représentant le type du noeud : `ELEMENT_NODE`, `ATTRIBUTE_NODE`, `TEXT_NODE`, `CDATA_SECTION_NODE`, `ENTITY_NODE`, `PROCESSING_INSTRUCTION_NODE`, `COMMENT_NODE`, `DOCUMENT_NODE`, `DOCUMENT_TYPE_NODE`, `NOTATION_NODE`.

```
>>> root.nodeType == minidom.Node.ELEMENT_NODE
True
>>> root.firstChild.nodeType == minidom.Node.ELEMENT_NODE
False
>>> root.firstChild.nodeType == minidom.Node.TEXT_NODE
True
```

L'attribute `nodeName` donne le nom du n{oe}ud (tag de la balise). Attention, le comportement est variable selon les noeuds : pour un élément ce sera le nom de la balise XML, pour un noeud représentant du texte, ce sera `#text`.

```
>>> root.nodeName
u'contacts'
>>> root.firstChild.nodeName
'#text'
```

L'attribut `nodeValue` donne la valeur du n{oe}ud (contenu). Attention, le comportement est variable selon les noeuds : pour un élément ce sera `None` pour un noeuds contenant de l'information ce sera cette information. Pour les noeuds contenant du texte, il est aussi possible d'utiliser l'attribut `data`.

```
>>> root.nodeValue
>>> root.firstChild.nodeValue
u'\n  '
```

```
>>> root.firstChild.data
u'\n  '
```

6.3.2 Informations liées aux attributs d'un noeud

La fonction `hasAttributes()` teste la présence d'attributs dans un noeud.

```
>>> root.hasAttributes()
False
>>> root.childNodes[1].hasAttributes()
True
```

L'attribute `attributes` fourni les attributs du noeud (objet du type `NamedNodeMap` ou `None`).

```
>>> root.firstChild.attributes
>>> attributes = root.childNodes[1].attributes
>>> attributes
<xml.dom.minidom.NamedNodeMap object at 0x...>
```

La fonction `getAttribute()` retourne la valeur d'un attribut comme une chaîne.

```
>>> c = root.childNodes[1]
>>> c.getAttribute('name')
u'doe'
```

La fonction `getAttributeNode()` retourne la valeur d'un attribut comme un objet de type `Attr`.

```
>>> c.getAttributeNode('name')
<xml.dom.minidom.Attr instance at 0x...>
```

Une `NamedNodeMap` se comporte comme un dictionnaire. La méthode `item()` permet d'accéder aux couples nom d'attribut / valeur textuelle d'attribut, la méthode `keys()` donne accès aux noms des attributs et la méthode `values()` donne accès à la liste des objets `Attr` représentant les attributs.

```
>>> attributes.items()
[(u'name', u'doe'), (u'firstname', u'john')]
>>> attributes.keys()
[u'name', u'firstname']
>>> attributes.values()
[<xml.dom.minidom.Attr instance at 0x...>, <xml.dom.minidom.Attr instance at 0x.
>>> for a in attributes.values():
...     print a.nodeType, a.nodeName, a.nodeValue
...
2 name doe
2 firstname john
```

En plus des méthodes classiques sur les dictionnaires, il existe deux méthodes pour manipuler les `NamedNodeMap` :

– `length` donne la taille du dictionnaire d'attributs,
– `item(i)` donne le i-eme élément (de type `Attr`) du dictionnaire ou `None`.

```
>>> attrs = root.childNodes[1].attributes
>>> for idx in range(0, attrs.length):
...     a = attrs.item(idx)
...     print '(' + a.name + ')',
...     print a.nodeType, a.nodeName, a.nodeValue
...
(name) 2 name doe
(firstname) 2 firstname john
```

6.4 Construire un document XML

6.4.1 Créer un arbre DOM

La création d'un document (de son arbre DOM) se fait par instanciation de la classe `Document` (c'est un objet comme un autre). Cette classe fournit des méthodes pour fabriquer les noeuds. A la création, un document ne contient rien, pas même de noeud racine.

```
>>> newdoc = minidom.Document()
>>> newdoc
<xml.dom.minidom.Document instance at 0x...>
>>> newdoc.documentElement
>>>
```

6.4.2 Créer des noeuds DOM

Les méthodes de création de noeud n'incluent pas l'ajout de l'élément créé dans l'arbre. Ici, nous allons créer un document excessivement simple :
– une racine `root` avec un attribut `name`,
– un commentaire,
– un noeud de texte contenu dans un noeud `sometext`.
La fonction `createElement(tag)` crée un nouvel élément (de type `Element`) avec le nom de tag passé en paramètre et le retourne.

```
>>> newroot = newdoc.createElement('root')
```

La fonction `createAttribute(name)` crée un noeud de type `Attr` avec le nom `name`. Une fois le noeud créé, la valeur peut être fixée avec l'attribut `nodeValue`.

```
>>> rootattr = newdoc.createAttribute('name')
>>> rootattr.nodeValue = 'foo'
```

La fonction `createTextNode(data)` crée un noeud de donnée contenant le texte passé en paramètre (à inclure dans un noeud englobant, ici le noeud `sometext` créé pour).

```
>>> textnode = newdoc.createElement('sometext')
>>> text = newdoc.createTextNode('this node\ncontains text')
>>> textnode.appendChild(text)
<DOM Text node "'this node\n'..."
```

La fonction `createComment(text)` crée un noeud de commentaires contenant le texte passé en paramètre.

```
>>> comment = newdoc.createComment('a very usefull comment')
```

6.4.3 Ajout de noeuds dans un arbre

Les méthodes d'ajout dans un arbre viennent en complément des méthodes de création. Leur dissociation est due aux multiples usages que l'on peut faire d'un noeud. Il est possible de construire un document XML partir d'un document existant, dans ce cas on ne va pas recréer tout les noeuds, mais peut être simplement les réorganiser.

La fonction `appendChild(new)` ajoute un élément à la fin de la liste des fils d'un noeud.

```
>>> newdoc.appendChild(newroot)
<DOM Element: root at 0x...>
>>> textnode.appendChild(text)
<DOM Text node "'this node\n'..."
>>> newroot.appendChild(textnode)
<DOM Element: sometext at 0x...>
```

La fonction `insertBefore(new, old)` ajoute un élément avant un fils donné d'un noeud.

```
>>> newroot.insertBefore(comment, textnode)
<DOM Comment node "'a very use'..."
```

La fonction `replaceChild(new, old)` remplace un élément fils d'un noeud par un autre élément.

La fonction `setAttribute(name, value)` crée un nouvel attribut sur un noeud sans passer par une instance de type `Attr`.

```
>>> newroot.setAttribute('usefull', 'nop')
```

La fonction `setAttributeNode(new)` ajoute un noeud attribut au noeud considéré.

```
>>> newroot.setAttributeNode(rootattr)
```

6.4.4 Supprimer des noeuds d'un arbre DOM

Il est aussi possible de supprimer des noeuds dans un arbre DOM, par exemple pour en créer une nouvelle version plus épurée ou pour une restructuration d'un document XML.

La fonction `removeChild(old)` supprime un fils d'un noeud à utiliser conjointement avec `unlink()`.

```
>>> try:
...     old = root.removeChild(root.firstChild)
...     old.unlink()
... except ValueError: print 'failed'
...
```

La fonction `removeAttribute(name)` supprime un attribut d'un noeud en le désignant par son nom.

```
>>> root.firstChild
<DOM Element: contact at 0x...>
>>> root.firstChild.removeAttribute('firstname')
```

La fonction `removeAttributeNode(old)` supprime un attribut d'un noeud par la référence de l'objet `Attr` le représentant ou lève l'exception `NotFoundErr`.

6.4.5 Sérialiser un document XML

L'objectif de faire un document XML est d'obtenir un fichier. Les fonctions `toxml()`, `toprettyxml()` donnent la version texte d'un arbre DOM (une ligne ou multi-ligne indentées). Elles sont disponibles sur tous les noeuds et l'évaluation est automatiquement récursive. Pour sérialiser un document, il suffit d'appeler une de ces méthodes sur l'objet document. Si l'appel est fait sur un noeud, alors le fichier sera un sous ensemble du document XML.

```
>>> newdoc.toxml()
'<?xml version="1.0" ?><root name="foo" usefull="nop"><!--a very usefull comment
```

Et la version lisible

```
>>> print newdoc.toprettyxml()
<?xml version="1.0" ?>
<root name="foo" usefull="nop">
    <!--a very usefull comment-->
    <sometext>
            this node
contains text
    </sometext>
</root>
```

6.5 Exercices

Le fichier `etudiants.xml` contient 400 dossiers d'étudiants (générés aléatoirement) avec la structure décrite ci-dessous. Le but de l'exercice est de produire trois listes sous forme de fichiers XML contenant (par ordre décroissant pour les deux premières) les étudiants admissibles, sur liste complémentaire ou refusés.

```
<?xml version="1.0" ?>
<etudiants>
  <etudiant nom="doe" prenom="john" dossier="0">
    <origine universite="mit" />
    <formation discipline="informatique" niveau="4" />
    <resultats moyenne="16" />
  </etudiant>
</etudiants>
```

Pour les trois listes, les informations conservées sont le nom, prénom, l'université d'origine et les résultats. Les règles de sélection sont les suivantes.

– pour être admis, un(e) étudiant(e) doit avoir un niveau Bac+4, dans une formation en informatique, avec une moyenne supérieure à 12 ;
– si sa discipline n'est pas informatique (mais toujours avec les critères de niveau et de moyenne), il (elle) est sur liste complémentaire ;
– sinon il(elle) est refusé(e).

Il faut pour cela parser le fichier `etudiants.xml`, naviguer dans l'arbre DOM ainsi produit et construire trois arbres DOM qui seront en fin de traitement sérialisés dans les fichiers `admissibles.xml`, `complementaires.xml` et `refuses.xml`. La structure de ces trois fichiers est décrite ci-dessous.

```
<?xml version="1.0" ?>
<admissibles|complementaires|refuses>
  <etudiant nom="doe" prenom="john" dossier="0">
    <origine universite="mit" />
    <resultats moyenne="16" />
  </etudiant>
</admissibles|complementaires|refuses>
```

CHAPITRE 7

Python et la persistance de données

7.1 Fichiers DBM

7.1.1 Description et utilisation

Le module `anydbm`[1] offre une solution pour des besoins simples de persistance en utilisant des fichiers plats. Il propose une solution standard indépendante de l'implémentation. Un fichier DBM s'utilise comme un dictionnaire, donc relativement simplement, seul l'initialisation est différente (ouverture du fichier). Les données sont manipulées comme des chaînes et accessibles à l'aide de clés.

La fonction `open(name, 'c')` ouvre le fichier nommé `name` (`'c'` signifie «en le créant s'il n'existe pas»).

```
>>> import anydbm
>>> dbmfile = anydbm.open('blabla', 'c')
```

La fonction `close()` ferme le fichier (requis ou non selon les implémentations, pour être sûr que les données sont sauvées il vaut mieux utiliser explicitement `close`).

```
>>> dbmfile.close()
```

La lecture et l'écriture se fait comme dans un dictionnaire, mais les clés sont obligatoirement des chaînes. L'exemple suivant crée deux entrées (`foo` et `bar`) dans le fichier DBM, les lit depuis ce fichier et ferme le fichier.

```
>>> dbmfile = anydbm.open('blabla', 'c')
>>> dbmfile['foo'] = 'perrier c foo'
>>> dbmfile['bar'] = 'cd bar ; more beer'
>>> print dbmfile['foo']
perrier c foo
>>> print dbmfile['bar']
```

1. L'utilisation du module `dbm` revient à utiliser une mise en oeuvre particulière.

```
cd bar ; more beer
>>> dbmfile.close()
```

La fonction `has_key()` test la présence d'une clé.

```
>>> dbmfile = anydbm.open('blabla', 'c')
>>> dbmfile.has_key('foo')
True
```

La fonction `keys()` donne la liste des clés du fichier. Obtenir les clé est un pré-requis à tout parcours d'un fichier DBM. Comme pour les dictionnaires, il est possible de parcourir les clés d'un fichier DBM avec un itérateur.

```
>>> dbmfile.keys()
['foo', 'bar']
>>> for key in dbmfile:
...     print key, dbmfile[key]
foo perrier c foo
bar cd bar ; more beer
```

La fonction `len()` donne le nombre d'entrées d'un fichier.

```
>>> len(dbmfile)
2
```

L'opérateur `del` permet de supprimer une entrée (ce qui suit le fonctionnement standard d'un dictionnaire).

```
>>> dbmfile.keys()
['foo', 'bar']
>>> del dbmfile['foo']
>>> dbmfile.keys()
['bar']
>>> len(dbmfile)
1
>>> dbmfile.close()
```

7.1.2 Limitations

Les fichiers DBM permettent uniquement de rendre persistant des chaînes de caractères. La conversion objet vers chaîne (et réciproquement) doit être gérée manuellement, ce qui rend leur utilisation rapidement complexe pour des objets composites.

7.2 Pickle et Shelve

7.2.1 *Object pickling*

Le module `pickle` (standard en Python) permet la sérialisation des objets mémoire en chaînes (et réciproquement). Il est utile pour la persistance et le transferts de données sur un réseau.

```
>>> import pickle
```

La manipulation des données utilise un fichier par sérialisation, où les données sont stockées sous forme de chaînes. Il n'y a pas de structuration des données (donc pas de recherche rapide possible).

7.2.2 (Dé)Sérialisation et fichiers

La classe `Pickler` est un sérialiseur vers un fichier et la fonction `dump()` réalise la sérialisation d'un objet (il existe aussi une fonction équivalente dans le module). L'exemple suivant sérialise un dictionnaire dans le fichier `foo.saved`.

```
>>> foo = {'a': 'aaa', 'b': 'bbb', 'c': 'ccc'}
>>> output = open('foo.saved', 'w')
>>> p = pickle.Pickler(output)                    #(1)
>>> p.dump(foo)                                   #(2)
>>> output.close()
```

Ce qui est équivalent à l'utilisation de la fonction `dump` du module.

```
>>> output = open('foo.saved', 'w')
>>> pickle.dump(foo, output)                      #(1,2)
>>> output.close()
```

La classe `Unpickler` est un dé-sérialiseur depuis un fichier et la fonction `load()` réalise la dé-sérialisation d'un objet (il existe aussi une fonction équivalente dans le module). L'exemple suivant recharge depuis le fichier `foo.saved` le dictionnaire sérialisé dans l'exemple précédent.

```
>>> input = open('foo.saved', 'r')
>>> p = pickle.Unpickler(input)                   #(1)
>>> foo2 = p.load()                               #(2)
>>> input.close()
>>> foo2
{'a': 'aaa', 'c': 'ccc', 'b': 'bbb'}
```

Ce qui est équivalent à l'utilisation de la fonction `load` du module.

```
>>> input = open('foo.saved', 'r')
>>> foo2 = pickle.load(input)                     #(1,2)
>>> input.close()
```

7.2.3 (Dé)Sérialisation et chaînes de caractères

La fonction `dumps()` sérialise un objet vers une chaîne (et non plus vers un fichier). Cette opération est pratique pour échanger des messages sur un réseau par exemple, ou bien pour les stocker dans un fichier DBM.

```
>>> data = pickle.dumps(foo)
>>> data
"(dp0\nS'a'\np1\nS'aaa'\np2\nsS'c'\np3\nS'ccc'\np4\nsS'b'\np5\nS'bbb'\np6\ns."
```

La fonction `loads()` dé-sérialise une chaîne vers un objet.

```
>>> foo3 = pickle.loads(data)
>>> foo3
{'a': 'aaa', 'c': 'ccc', 'b': 'bbb'}
```

7.2.4 DBM + Pickle = Shelves

Le module `shelves` exploite les deux modules précédents (en offrant l'interface du second) :
– `pickle` pour sérialiser les objets (éventuellement complexes),
– `anydbm` pour la gestion des clés et des fichiers.

```
>>> import shelve
```

L'exemple suivant crée un fichier `base` dans lequel nous stockons un dictionnaire associé à la clé `foo`. Puis, nous ouvrons de nouveau ce fichier pour récupérer dans le dictionnaire stocké l'entrée associée à la clé `a`.

```
>>> base = shelve.open('base')
>>> base['foo'] = {'a': ['a1','a2'], 'b': ['b1','b2']}
>>> base.close()
>>> base2 = shelve.open('base')
>>> print base2['foo']['a']
['a1', 'a2']
>>> base2.close()
```

7.2.5 Remarques

– La concurrence de mise à jour n'est pas supportée avec `shelve`, une possibilité est d'utiliser `fcntl`.
– La définition des classes doit être importable au moment du chargement des données par `pickle`. Il n'est donc pas suffisant de partager le fichier de donnée, mais il faut aussi partager les modules de mise en oeuvre des classes utilisées (qui doivent être présentes dans le `PYTHONPATH`).
– La compatibilité des fichiers n'est pas garantie entre deux implémentations de DBM. Il n'y a donc que portabilité du code, pas des données.

7.3 Python et SQL

7.3.1 Exemple avec SQlite

Python inclut un module d'accès aux bases de données offrant une interface standardisée (de facto). Des mises en oeuvre existent pour les différentes bases de données courantes. Toutefois, l'utilisation de chaque base peut varier, donc le code Python d'accès à une base n'est pas *100% portable*. Il est donc recommandé dans une application de définir une couche d'abstraction de la base.

L'utilisation d'une base de données est ici illustré avec une base `sqlite3`.
– Une base sqlite3 est représentée sur le disque par un fichier.
– C'est une base embarquée (il n'y a pas de serveur à faire tourner).
– Elle gère bien des table de plusieurs centaines de milliers de lignes.
– Elle est incluse dans toutes les distributions de Python depuis la version 2.5.

```
>>> import sqlite3
```

7.3.2 Opérations de base

La fonction `connect()` crée un objet représentant la connexion à une base de données. Si la base n'existe pas, elle sera créé.

```
>>> connexion = sqlite3.connect('test.sqlite3')
```

Il est aussi possible de créer une base en mémoire (et donc sans version persistante sur le disque).

```
>>> connexion = sqlite3.connect(':memory:')
```

Des curseurs sont utilisés pour les interactions avec la base : émission de requêtes SQL. Ils sont créés par l'objet connexion. Un curseur offre la méthode `execute` qui permet de demander à la base l'évaluation d'une requête SQL. Le curseur est ici utilisé pour créer une table `test1` ayant un seul champ `val` de type entier. Ensuite, le curseur est utilisé pour insérer une valeur dans cette table.

```
>>> curseur = connexion.cursor()

>>> curseur.execute("CREATE TABLE test1 (val integer)")
<sqlite3.Cursor object at 0x...>

>>> curseur.execute("INSERT INTO test1 VALUES(-1)")
<sqlite3.Cursor object at 0x...>
```

Insérer des données

Une insertion peut se faire en utilisant un tuple (comme dans l'exemple précédent) ou un dictionnaire pour fournir les valeurs. Dans le cas de l'utilisation des tuples, les données sont prises dans l'ordre du tuple qui doit être de même longueur que le tuple de format de la requête (argument de la clause `VALUES`). L'exemple suivant crée une nouvelle table et utilise un dictionnaire pour fournir les valeurs à insérer. La méthode `executemany` permet d'insérer une séquence de données dans une table sans avoir à faire une boucle en Python.

```
>>> curseur = connexion.cursor()
>>> curseur.execute('CREATE TABLE test2 (name text, firstname text)')
<sqlite3.Cursor object at 0x...>

>>> curseur.execute('INSERT INTO test2 VALUES (?, ?)', ('doe', 'john'))
<sqlite3.Cursor object at 0x...>

>>> valeurs = (('martin', 'pierre'), ('dupont', 'paul'))
>>> curseur.executemany('INSERT INTO test2 VALUES (?, ?)', valeurs)
<sqlite3.Cursor object at 0x...>
```

Récupérer des données

Les résultats de requêtes (réponses à une command `select`) sont des structures de données Python : listes (ensemble des réponses) de tuples (les données d'une réponse). Les opération `fetchone()` et `fetchall()` offertes par le curseur permettent de récupérer respectivement une valeur (en fait les valeurs une par une) ou toutes les valeurs. Attention sur les `select` retournant beaucoup de réponses, le `fetchall()` a une borne supérieure. L'exemple suivant exécute une requête sur la base et récupère d'abord la première réponse, en affichant le premier champs, puis toutes les réponses restantes, en affichant les deux champs.

```
>>> curseur.execute("SELECT * FROM test2")
<sqlite3.Cursor object at 0x...>

>>> print curseur.fetchone()
(u'doe', u'john')

>>> valeurs = curseur.fetchall()
>>> for v in valeurs:
...     print v[0], v[1]
...
martin pierre
dupont paul
```

Opérations complémentaires

L'objet représentant une connexion offre les méthodes suivantes.

– `close()` demande explicitement la fermeture d'une connexion (c'est implicite lorsque l'objet connexion est détruit au sens Python).

– `commit()` valide une transaction avec la base.
– `rollback()` annule une transaction avec la base.
En complément, [2] présente le module dans son ensemble.

7.4 Exercices

7.4.1 Module «Modèle» du MVC

Ecrire un script Python qui crée une table représentant des étudiants avec les champs suivants (le type des données est précisé entre parenthèses) : num dossier (`integer`), nom (`text`), prénom (`text`), université (`text`), discipline (`text`), niveau (`integer`), moyenne(`integer`).

Ecrire un programme Python chargeant l'ensemble des données du fichier `etudiants.xml` dans cette table. Pour cela, une partie du code de l'exercice précédent est réutilisable pour le parcours de l'arbre DOM.

7.4.2 Module «Contrôleur» du MVC

Développer une classe Controleur qui propose les traitements suivants sur la base de données :
– obtenir la liste des numéros de dossiers, nom et prénom des étudiants,
– obtenir la fiche complète d'un étudiant par rapport à son numéro de dossier,
– insérer une fiche d'étudiant en fournissant toutes les informations.

2. http ://docs.python.org/library/sqlite3.html

CHAPITRE 8

Python et les interfaces graphiques

8.1 Python et Tkinter

8.1.1 Tkinter

Le module `Tkinter` cite{tkinter-intro} est basé sur Tk, la librairie développée par J.~Ousterout pour faire des interfaces graphiques en Tclcite{tcltk}, et disponible sur plusieurs plates-formes (dont X11, Mac OS X, MS-Windows). `Tkinter` est le standard de facto pour Python offrant une vision objet de Tk. Toutefois, Tcl est un langage où tout est chaînes de caractères. Il en résulte qu'un certain nombre de valeur utilisées dans le cas de `Tkinter` sont des chaînes de caractères (utilisées directement comme tel ou bien sous couvert de variables offertes par le module).

Comme tout programme avec une interface graphique, l'exécution est dirigée par les événements. Les interactions entre Python et l'interface graphique peut prendre plusieurs formes :
– Traitement GUI Python puis Tkinter puis Tk puis Librairie graphique.
– Evénement graphique puis Tk puis Tkinter puis Traitements Python.

8.1.2 Premiers pas

Le «hello world !» version graphique tient en quatre lignes, présentées dans l'exemple suivant. Le module `Tkinter` est chargé. Un widget (ici un `Label`) est créé puis ajouté à l'environnement graphique (`pack`). Enfin, la boucle d'événements est démarrée. Elle est active tant que la fenêtre n'est pas fermée (et le prompt de l'interpréteur reste suspendu).

```
>>> import Tkinter
>>> widget = Tkinter.Label(None, text='hello world!')
>>> widget.pack()
>>> widget.mainloop()
```

8.1.3 Configuration de widgets

Dans le cas du `Label` de «hello world !», il n'y a pas de widget parent (premier argument `None`). Il est attaché directement à la fenêtre par défaut. Le texte du `Label` représente sa configuration. La configuration peut être réalisée ou changée après la création avec la méthode `config()` des widgets. L'exemple suivant est identique en terme de résultat à l'exemple précédent.

```
>>> widget = Tkinter.Label(None)
>>> widget.config(text='hello world!')
>>> widget.pack()
>>> widget.mainloop()
```

8.1.4 Composition de widgets avec `pack`

Les composition des widgets est délégué au gestionnaire de géométrie.

– Le placement du widget se fait par rapport à son contenant (avec l'option de configuration `side`) en utilisant un placement cardinal (`Tkinter.TOP`, `Tkinter.LEFT`, `Tkinter.RIGHT`, `Tkinter.BOTTOM`). Par défaut, un widget est attaché en haut, ou au-dessous des widgets existants
– Si l'on souhaite une occupation maximale de l'espace, utiliser l'option de configuration `expand=YES`. La direction de l'occupation maximale est précisée par `fill`. Elle peut être en largeur (`Tkinter.X`), en hauteur (`Tkinter.Y`) ou les deux (`Tkinter.BOTH`).

8.1.5 Lancer des commandes

Pour initier un traitement depuis une interface graphique, la méthode courante est de définir un bouton, un menu, une scrollbar, etc. et d'y associer un traitement (fonction ou méthode sans argument). L'exemple suivant crée un bouton et y associe la commande `sys.exit` pour quitter l'exécution du programme (dans le cas courant l'interpréteur complet). L'option `text` précise le texte qui apparaît sur le bouton.

```
>>> import sys
>>> widget = Tkinter.Button(None)
>>> widget.config(text='press to quit', command=sys.exit)
>>> widget.pack()
>>> widget.mainloop()
```

8.1.6 Composition et redimensionnement

Relations entre la composition et le redimensionnement :
– par défaut le widget conserve sa taille initiale,
– pour suivre les changements utiliser `expand`.
L'exemple suivant configure le bouton de telle sorte qu'il occupe toujours tout l'espace en largeur lors du redimensionnement de la fenêtre. Par contre, le bouton n'occupe pas tout l'espace en hauteur.

```
>>> import Tkinter
>>> import sys
>>> widget = Tkinter.Button(None)
>>> widget.config(text='press to quit', command=sys.exit)
>>> widget.pack(expand=Tkinter.YES, fill=Tkinter.X)
>>> widget.mainloop()
```

8.1.7 Boutons et traitements

Un bouton ne peut déclencher qu'un traitement, il est donc souvent nécessaire d'utiliser une fonction qui regroupe les traitements. Dans l'exemple suivant, la fonction `handler` affiche «hello world» sur la sortie standard avant de quitter l'application.

```
>>> import Tkinter
>>> import sys
>>> def handler():
...     print 'hello world!'
...     sys.exit()
...
>>> widget = Tkinter.Button(None)
>>> widget.config(text='press to quit', command=handler)
>>> widget.pack(expand=Tkinter.YES, fill=Tkinter.X)
>>> widget.mainloop()
```

Une classe d'utilisation d'un widget graphique est une bonne manière de créer une relation forte entre widget, traitements et données. Cet ensemble (bouton, données, traitements) est facilement réutilisable. L'exemple présente un exemple de bouton pour quitter une application où le message est paramétrable.

```
>>> import Tkinter
>>> import sys
>>> class HelloQuitButton:
...     def __init__(self, msg='Quit'):
...         self.msg = msg
...         b = Tkinter.Button(None)
```

```
...             b.config(text=self.msg, command=self.handle)
...             b.pack()
...         def handle(self):
...             print self.msg
...             sys.exit()
...
>>> hqb = HelloQuitButton()
>>> Tkinter.mainloop()
```

Un *handler* peut être défini comme un objet : définition d'une classe représentant une fonction et instanciation pour l'utilisation. L'exemple suivant redéfinit le handler précédent comme une classe.

```
>>> import Tkinter
>>> import sys
>>> class Callable:
...     def __init__(self):
...             self.msg = 'hello world!'
...     def __call__(self):
...             print self.msg
...             import sys
...             sys.exit()
...
>>> widget = Tkinter.Button(None)
>>> widget.config(text='hello', command=Callable())
>>> widget.pack()
>>> Tkinter.mainloop()
```

8.1.8 Définition de *bindings*

La fonction `bind` permet d'associer une fonction (à 1 argument) à un événement et à un widget. Les événements les plus courants sont les clics souris et la saisie au clavier. L'exemple suivant crée deux fonctions pour afficher «hello world !» et pour afficher «bye» sur la sortie standard avant de quitter le programme. Ensuite, il crée un label et y associe la fonction `hello` au clic gauche et la fonction `quit` au double clic.

```
>>> import Tkinter
>>> import sys
>>> def hello(event):
...     print 'hello world!'
...
>>> def quit(event):
...     print 'bye'
...     sys.exit()
...
>>> widget = Tkinter.Label(None, text='press')
>>> widget.pack()
>>> widget.bind('<Button-1>', hello)
'805810704hello'
```

```
>>> widget.bind('<Double-1>', quit)
'805810224quit'
>>> widget.mainloop()
```

L'événement «reçu» par un *handler* contient des informations qui peuvent être interprétées dans le cadre du traitement. L'exemple suivant définit deux fonctions dont le but est d'afficher un caractère tapé au clavier et d'afficher les coordonnées de la souris. La classe `Tk` représente la fenêtre principale du programme. Le label créé est associé à cette fenêtre (premier paramètre du constructeur). Puis, les deux fonctions sont associées au clavier et au clic sur le label. L'utilisation de `focus` sélectionne le label pour recevoir les événements clavier. Enfin, la boucle d'événements est démarré sur la fenêtre principale. Lorsque la fenêtre est active et que l'on tape une touche au clavier, cela génère un événement passé à la fonction `onKey` qui va en extraire le caractère associé. (Même principe pour le clic de souris.)

```
>>> import Tkinter
>>> def onKey(event):
...     print 'got key', event.char
...
>>> def onClick(event):
...     print event.widget, event.x, event.y
...
>>> root = Tkinter.Tk()
>>> lab = Tkinter.Label(root, text='hello world')
>>> lab.bind('<KeyPress>', onKey)
'805787968onKey'
>>> lab.bind('<Button-1>', onClick)
'808388888onClick'
>>> lab.focus()
>>> lab.pack()
>>> root.mainloop()
.805789368 30 14
.805789368 44 11
got key e
got key r
```

Quelques noms de bindings courants

Nom	Signification
<KeyPress>	Pression sur une touche
<KeyPress-a>	Pression sur la touche 'A' (minuscule)
<KeyPress-A>	Pression sur la touche 'A' (majuscule)
<Return>	Pression sur entrée
<Escape>	Touche d'échappement
<Up> <Down>	Pression sur les flèches
<Button-1>	Click souris gauche
<Button-2>	Click souris milieu (ou les deux)
<Button-3>	Click souris droit
<ButtonRelease>	Fin de click gauche
<Motion>	Mouvement de la souris
<B1-Motion>	Mouvement de la souris avec click gauche
<Enter> <Leave>	Entrée et sortie souris d'un widget
<Configure>	Redimensionnement de la fenêtre
<Map> <Unmap>	Ouverture et iconification de la fenêtre

8.1.9 Assemblage de widgets

L'assemblage de widgets se fait par utilisation de *containers* graphiques, principalement en définissant des `Frames` auxquels sont rattachés les widgets. L'exemple suivant utilise une `Frame` pour regrouper deux boutons. La commande associée au second bouton permet la fermeture de la fenêtre sans arrêter l'interpréteur.

```
>>> import Tkinter
>>> def hello():
...     print 'hello world!'
...
>>> win = Tkinter.Frame()
>>> win.pack()
>>> Tkinter.Label(win, text='hello world!').pack(side=Tkinter.TOP)
>>> b1 = Tkinter.Button(win, text='hello', command=hello)
>>> b1.pack(side=Tkinter.LEFT)
>>> b2 = Tkinter.Button(win, text='quit', command=win.quit)
>>> b2.pack(side=Tkinter.RIGHT)
>>> win.mainloop()
```

8.1.10 Widgets orientés objet

L'approche objet s'applique bien aux interfaces graphiques en utilisant l'héritage pour définir de nouveaux widgets par spécialisation ou extension. L'exemple suivant présente une spécialisation du widget `Button` en bouton pour quitter l'application (en fait le bouton ne détruit que lui).

```
>>> class MyByeButton(Tkinter.Button):
...     def __init__(self, parent=None, **config):
...         Tkinter.Button.__init__(self, parent, config)
...         self.pack()
...         self.config(command=self.callback)
...     def callback(self):
...         print 'bye...'
...         self.destroy()
...
>>> MyByeButton(text='hello world')
>>> Tkinter.mainloop()
```

8.1.11 A propos de l'aspect

La spécification de la taille et de la couleur est valable pour tous les widgets de `Tkinter`. La spécification des polices, tailles et styles de caractères est valable pour tout widget contenant du texte. L'exemple suivant définit un label écrit en jaune sur fond bleu avec la police Courrier 20 gras dans une surface totale de 3 lignes sur 20 caractères.

```
>>> root = Tkinter.Tk()
>>> msg = Tkinter.Label(root, text='hello world')
>>> msg.config(font=('courier', 20, 'bold'))
>>> msg.config(bg='blue', fg='yellow')
>>> msg.config(height=3, width=20)
>>> msg.pack(expand=Tkinter.YES, fill=Tkinter.BOTH)
```

8.1.12 Application multi-fenêtres

Il est possible de définir dans une même applications des fenêtres indépendantes (non contenues visuellement dans la même fenêtre). L'exemple suivant définit deux fenêtres indépendantes contenant les labels «hello» et «world».

```
>>> root = Tkinter.Tk()
>>> win1 = Tkinter.Toplevel(root)
>>> Tkinter.Label(win1, text='hello').pack()
>>> win2 = Tkinter.Toplevel(root)
>>> Tkinter.Label(win2, text='world').pack()
```

Terminaison d'une application multi-fenêtres :
– `destroy()` supprime récursivement la fenêtre concernée,
– `quit()` termine la boucle d'événements sans détruire la fenêtre.

```
>>> root = Tkinter.Tk()
>>> win1 = Tkinter.Toplevel(root)
>>> b1 = Tkinter.Button(win1)
>>> b1.config(text='moi', command=win1.destroy)
>>> b1.pack()
>>> win2 = Tkinter.Toplevel(root)
>>> b2 = Tkinter.Button(win2)
>>> b2.config(text='nous', command=root.destroy)
>>> b2.pack()
>>> root.mainloop()
```

8.2 Petit tour des widgets courants

Dans la suite du chapitre nous considérons que le contenu module `Tkinter` est importé dans l'espace courant. Bien qu'étant plutôt un défenseur de l'utilisation des modules avec leur préfixe, il est vrai que, en mode interactif, cela a ses avantages de tout importer dans le scope global. Mais faites toujours attention aux risques de conflits de nom !

```
>>> from Tkinter import *
```

8.2.1 Champ de texte

Le widget `Entry` offre un champ de saisie de texte sur une seule ligne. Il dépend des options de configuration suivantes :
– `state='disable'` lecture seule,
– `show='*'` saisie masquée.
L'exemple suivant construit une petit interface avec une zone de saisie et deux boutons. La touche entrée est associée dans la boite de saisie à la fonction `fetch()`. Le premier bouton affiche sur la sortie standard le contenu de la zone de saisie. Pour cela, il utilise aussi la commande `fetch()`. Le second bouton ferme l'interface.

```
>>> def fetch():
...     print 'Texte: <%s>' % ent.get()
...
>>> root = Tk()
>>> ent = Entry(root)
>>> ent.insert(0, 'Saisir texte')
>>> ent.pack(side=TOP, fill=X)
>>> ent.bind('<Return>', (lambda event: fetch()))
>>> b1 = Button(root, text='Fetch', command=fetch)
>>> b1.pack(side=LEFT)
>>> b2 = Button(root, text='Quit', command=root.destroy)
>>> b2.pack(side=RIGHT)
>>> root.mainloop()
```

Manipulations de texte

– index dans une entrée de texte : De `'0'` à `END`,
– position d'insertion du curseur `INSERT`,
– `get(deb, fin)` retourne le texte à la position donnée,
– `insert(pos, txt)` insert le texte `txt` à la position `pos`,
– `delete(deb [,fin])` supprime le texte entre les position `deb` et `fin` (le second est optionnel).

8.2.2 Agencement de widgets

Un formulaire de saisie est un exemple type d'agencement de plusieurs widgets `Label` et `Entry`. Afin d'avoir un formulaire bien aligné, l'utilisation du gestionnaire cardinal (au travers de `pack`) est remplacé par l'utilisation du gestionnaire «grille». La méthode `grid(row=X, column=Y)` permet l'alignement des différents éléments dans une grille. L'exemple suivant construit un formulaire à trois entrées.

```
>>> root = Tk()
>>> r = 0
>>> for line, item in enumerate(['foo', 'bar', 'stuff']):
...     l = Label(root, text=item, width=10)
...     e = Entry(root, width=10)
...     l.grid(row=line, column=0)
...     e.grid(row=line, column=1)
```

Attention

Les deux gestionnaires (cardinal et grille) ne se marient pas toujours bien.

8.2.3 Zone de texte

Le widget `Text` offre une zone de saisie et de manipulation de texte sous la forme d'une chaîne de caractères multi-lignes. L'exemple suivant définit une zone de texte et un bouton. L'utilisation de ce dernier affiche le contenu de la zone de texte sur la sortie standard à l'aide de la fonction `fetch()`. L'option `wrap=WORD` de la zone de texte précise que le découpage de son contenu se fait sur les mots (lorsqu'un mot ne tient pas sur la ligne, la zone de texte passe automatiquement à la ligne).

```
>>> def fetch():
...     print txt.get('1.0', END+'-1c')
...
>>> root = Tk()
>>> txt = Text(root, height=5, width=20, wrap=WORD)
>>> txt.insert('1.0', 'Saisir du texte sans couper les mots...')
>>> txt.pack(side=TOP)
>>> Button(root, text='fetch', command=fetch).pack()
>>> root.mainloop()
```

Manipulation de texte

- index dans une zone de texte : de `'1.0'` (ligne.colonne) à `END`,
- `INSERT` donne la position d'insertion du curseur,
- `SEL_FIRST, SEL_LAST` contient les position de début et fin de la sélection,
- `get(deb, fin)` retourne le texte à la position donnée,
- `get('1.0', END+'-1c')` retourne tout le texte contenu dans la zone.
- `insert('li.co', txt)` ajoute le texte à la position 'li.co',
- `delete(deb [,fin])` supprime le texte entre les position deb et fin,
- `search(chaine, deb, fin)` retourne la position de début de la chaîne dans la zone de texte,

– `see('li.co')` Fait défiler l'affichage pour rendre la position donnée visible,
– pour les déplacements relatifs dans une zone de texte par rapport à une position : verb@+/-@ un certain nombre de `'c'` (caractère), `'w'` (mot) ou `'l'` (ligne).

8.2.4 Listboxes

Le widget `Listbox` permet de gérer des listes d'éléments et offre des méthodes pour l'ajout, le retrait et la sélection d'éléments. L'exemple suivant définit une listbox contenant 10 entrée et un bouton. L'utilisation du bouton déclenche la fonction `fetch()` qui affiche sur la sortie standard l'entrée de la liste sélectionnée (ou la première ligne par défaut).

```
>>> def fetch():
...     print list.get(ACTIVE)
...
>>> root = Tk()
>>> list = Listbox(root)
>>> list.pack(side=TOP)
>>> Button(root, text='fetch', command=fetch).pack()
>>> for index in range(10):
...     list.insert(index, 'ligne-' + str(index))
...
>>> root.mainloop()
```

Manipulation de listes

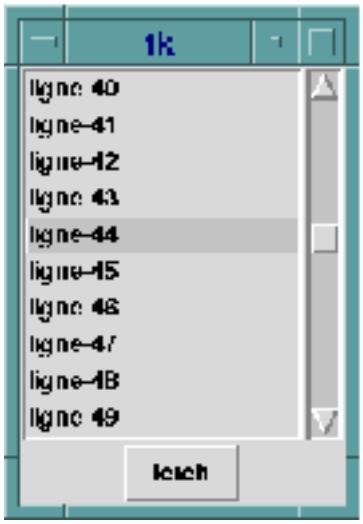
– index dans une liste : de `'0'` à `END`.
– `curselection()` retourne l'index sélectionné,
– `get(idx)` retourne l'élément à l'index `idx`,
– `get(ACTIVE)` Retourne l'élément sélectionné,
– `insert(idx, element)` ajoute l'élément à l'index `idx`,
– `delete(deb [,fin])` supprime les éléments entre les index deb et fin (le second est optionnel).

8.2.5 Barres de défilement

Le widget `scrollbar` permet de gérer le défilement (horizontal ou vertical) des zones de texte et des listboxes. Il utilise des méthodes de défilement fournies par ces deux widgets. L'exemple suivant reprend le précédent en y ajoutant une barre de défilement pour gérer une liste de 100 éléments. Dans un premier temps, il crée une frame pour regrouper la liste et la barre. Ces deux dernières sont configurées pour que la liste évolue en fonction de l'utilisation de la barre.

```
>>> root = Tk()
>>> frame = Frame(root)
>>> frame.pack(side=TOP)
>>> list = Listbox(frame)
```

```
>>> sbar = Scrollbar(frame)
>>> sbar.config(command=list.yview)
>>> list.config(yscrollcommand=sbar.set)
>>> sbar.pack(side=RIGHT, fill=Y)
>>> list.pack(side=LEFT, expand=YES, fill=BOTH)
>>> Button(root, text='fetch', command=fetch).pack(side=BOTTOM)
>>> for index in range(100):
...     list.insert(index, 'ligne-' + str(index))
...
>>> root.mainloop()
```



8.2.6 Boîtes à cocher et boutons radio

Les deux widgets suivants permettent de gérer des collections de données :
– `checkbutton` permet un choix multiple (boîtes à cocher),
– `radiobutton` permet un choix unique (boutons radio).
Les options de ces deux widgets sont :
– `command` pour associer une action,
– `variable` pour fixer et récupérer la valeur.
La variable doit être déclarée comme une variable `Tkinter`. Cela permet une interaction entre Tk et Python initiée par Tk (principe du `callback`) : `IntVar` et `StringVar` permettent de créer des objets de stockage de valeurs entières ou de chaînes.

Boîte à cocher

L'exemple suivant crée un ensemble de trois boîtes à cocher dont la valeur va être représentée par un entier. Chaque boîte est indépendante, et c'est le fait de mémoriser les trois variables Tkinter dans une liste qui les relie. La fonction `state()` déclenchée par le bouton affiche l'état des différentes boîtes.

```
>>> vars = list()
>>> root = Tk()
>>> for item in ['foo', 'bar', 'stuff']:
```

```
>>>     var = IntVar()
>>>     Checkbutton(root, text=item,
...         variable=var).pack(side=LEFT)
...     vars.append(var)
...
>>> def state():
...     for v in vars:
...         print v.get(),
...
>>> b = Button(root, text='State', command=state)
>>> b.pack(side=LEFT)
>>> root.mainloop()
```

Boutons radio

L'exemple suivant crée un ensemble de trois boutons radio dont la valeur va être représentée par une chaîne. Les trois boutons sont indépendant et c'est le fait d'utiliser la même variable Tkinter qui les relie. La fonction `state()` déclenchée par le bouton affiche le nom du bouton radio actuellement sélectionné.

```
>>> root = Tk()
>>> var = StringVar()
>>> for item in ['foo', 'bar', 'stuff']:
...     rb = Radiobutton(root, text=item,
...                      value=item,
...                      variable=var)
...     rb.pack(side=LEFT)
...
>>> def state():
...     print var.get()
...
>>> b = Button(root, text='State', command=state)
>>> b.pack()
>>> root.mainloop()
```

8.3 Autres widgets prêt à l'emploi

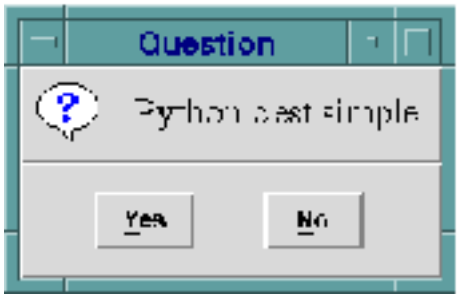
8.3.1 Boîtes de dialogue

Le module `tkMessageBox` offre un certain nombre de boîtes de dialogue standard pour l'affichage de messages (avec une icône associée). Ces boîtes prennent le contrôle de la boucle d'événements (tant qu'elle ne sont par fermées). Les trois boîtes de base sont :
– `showinfo` pour donner de l'information,
– `showwarning` pour signaler un warning,
– `showerror` pour signaler une erreur.

```
>>> from Tkinter import *
>>> from tkMessageBox import *
>>> root = Tk()
>>> showinfo('Information', "Python c'est simple")
'ok'
>>> showwarning('Attention', "Risque de dependance")
'ok'
>>> showerror('Trop tard', "Peux plus m'en passer")
'ok'
```

Pour les interactions primitives avec l'usager, les widgets `askyesno` et `askokcancel` permettent d'avoir une réponse booléenne à une question : `1` si 'yes' et `0` si 'no'.

```
>>> root = Tk()
>>> if askyesno('Question', "Python c'est simple"):
...     showinfo('xxx', 'Merci !!!')
... else:
...     showwarning('xxx', 'Dommage...')
...
```



Pour des interactions basiques complémentaires avec l'usager, `askfloat`, `askinteger` et `askstring` permettent de demander des réponses typées.

```
>>> from tkSimpleDialog import *
>>> root = Tk()
>>> askfloat('Float', 'Entrer un nombre decimal')
123
>>> askinteger('Integer', 'Entrer un nombre entier')
123.4
>>> askstring('String', 'Entrer une chaine')
'qwerty'
```

Le module `tkFileDialog` fournit des boîtes de dialogue pour les interactions relatives aux fichiers. Les différentes widgets offerts respectent les normes de l'interface graphique (X11, Mac OS X ou MS-Windows) :

- `askopenfilename`, `asksaveasfilename` et `askdirectory` retournent le nom du fichier sélectionné ou une chaîne vide,
- `askopenfile` et `asksaveasfile` retournent l'objet représentant le fichier en mode `'r'` (lecture), `'w'` (écriture) , ou `None`,
- `filetypes`, `initialdir` et `initialfile` sont des variables destinés configurer la recherche de fichiers.

```
>>> from tkFileDialog import *
>>> askopenfilename(filetypes=[('Text', '.txt')])
'/home/raphael/enseign/python/bar.txt'
>>> asksaveasfilename(initialdir='/tmp')
'/tmp/foo.txt'
```

8.3.2 Barre de menus

La définition de menus dans une fenêtre se fait à l'aide d'une `Frame` dédiée. Le widget `Menubutton` permet de définir un élément de la barre et le widget `Menu` permet de définir les commandes associées à l'élément. L'exemple suivant crée une frame pour servir de barre de boutons puis un élément de cette barre pour les commandes relatives aux fichiers. Ensuite, une entrée est ajouté dans ce premier menu offrant la fonction «Quit».

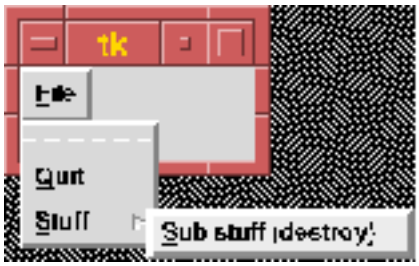
```
>>> root = Tk()
>>> menubar = Frame(root)
>>> menubar.pack(side=TOP, fill=X)
>>> fbutton = Menubutton(menubar, text='File', underline=0)
>>> fbutton.pack(side=LEFT)
>>> filemenu = Menu(fbutton)
>>> filemenu.add_command(label='Quit', command=root.destroy, underline=0)
>>> fbutton.config(menu=file) # lien entre bouton et menu
```

L'organisation des menus dans la barre se fait à l'aide des variables et fonctions suivantes :

- `bd, relief` nombre de points et relief du menu (`RAISED, SUNKEN, RIDGE`),
- `add_separator()` ajoute une séparation entre deux commandes d'un menu,
- `add_cascade()` création d'un sous-menu,
- `entryconfig(num, state=...)` fixe l'aspect disponible (`ENABLED`, valeur par défaut) ou non disponible (`DISABLED`) d'une entrée d'un menu.

L'exemple suivant définit un sous menu «Stuff» à notre menu «File».

```
>>> submenu = Menu(fbutton, tearoff=0)
>>> submenu.config(bd=2)
>>> submenu.add_command(label='Sub stuff(destroy)',
...                     command=root.destroy,
...                     underline=0)
>>> filemenu.add_cascade(label='Stuff', menu=submenu, underline=0)
```



8.4 Autres extensions disponibles

– **Tix** *Tk Interface eXtension* cite{tixbook,tixuser} est une extension de Tkinter qui offre des widgets de plus haut niveau (plus de 40) comme `ComboBox`, `NoteBook`, `DirTree`, `FileSelectBox`, etc. `http://tix.sourceforge.net`
– **wxPython** offre l'accès à la librairie wxWindow `http://www.wxpython.org`
– **PyQt** offre l'accès au toolkit Qt (Opie pour Ipaq).
– **PyKDE** offre l'accès aux librairies KDE.
– **PyGTK** offre l'accès aux widgets GTK.

8.5 Exercices

8.5.1 Module «Vue» du MVC

Développer une interface graphique en utilisant `Tkinter`, reposant sur les traitements du contrôleur défini dans le chapitre précédent et composées des éléments suivants :
– une `Listbox` associée à une `Scrollbar` donne la liste des numéros de dossier, nom et prénom des étudiants~ ;
– un formulaire (ensemble de `Label` et `Entry`) donnant les information d'un étudiant sélectionné dans la `Listbox` (par exemple avec un double clic)~ ;
– un formulaire de saisie associé à un `Button` pour entrer de nouvelles fiches d'étudiants.

CHAPITRE 9

Conclusion

Voilà qui termine notre petit tour du langage Python. En espérant que ce support vous a assisté dans la découverte de Python. Il reste plein de sujets à aborder comme la programmation réseau avec Python, les *frameworks* de développement web, les modules de test, etc.

La programmation est une activité où rien ne remplacera jamais la pratique, c'est donc en développant que l'on devient développeur.

Bonne continuation !

CHAPITRE 10

Remerciements, historique et à faire

10.1 Remerciements

Un grand merci à :
– Thérèse Libourel pour m'avoir demandé de monter une formation d'une semaine à Python et aux participants pour l'avoir suivie (base de la version 1.0.0).
– Fabien Devaux pour ses remarques sur le module `re` et sur la spécification de l'encodage d'un fichier (prises en compte dans la version 1.0.1).
– Cécile Vénéziani et Jérémy Garrouste pour leurs remarques sur les typos et erreurs.

10.2 Historique

– *1.4.2, avril 2010* Le document est maintenant réalisé avec Sphinx, utilisation de doctests (sauf pour le chapitre 6), la partie concernant Postgres a été remplacée par une partie concernant Sqlite3, et passage en python 2.6.
– *1.3, septembre 2007* Correction de typos, d'erreurs et passage en Python 2.5.
– *1.2.1, mars 2007* Correction de typos et érreurs.
– *1.2.0, janvier 2007* Correction de typos et respect des règles d'écriture.
– *1.1.0, juillet 2005* Mise à jour des exemples et introduction de certaines spécificités de la version 2.4.
– *1.0.1* Améliorations et corrections d'erreurs sur le module `re` et sur l'encodage des fichiers.

10.3 À faire

– Définir une signalétique pour différencier ce qui est pour tous de ce qui est une fois Python bien compris.
– Faire un index.
– Refaire une passe sur l'ensemble afin d'améliorer les explications et rajouter, si besoin est, des exemples.

– Ajouter des chapitres sur les modules de tests, un ou plusieurs framework web, les moyens pour faire de la génération de code.
– Parler de SQLObject dans le chapitre sur la persistance ?

GNU Free Documentation License

Version 1.2, November 2002

11.1 Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful 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.

11.2 Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under

the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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 Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document 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, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. 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, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include 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, PostScript or PDF 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.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknow-**

ledgements", **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties : any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

11.3 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.

11.4 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. 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.

11.5 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 :

1. 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.
2. 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 fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. 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.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section Entitled "History", Preserve 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.
10. 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.
11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

12. 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.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

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.

11.6 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, and that you preserve all their Warranty Disclaimers.

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".

11.7 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.

11.8 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, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

11.9 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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

11.10 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.

11.11 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/.

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.

11.12 Addendum : 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 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this :

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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.