Langages Informatiques

# Programmation avec le langage *Python*

## Un outil commode au service de l'ingénieur

Xavier Dupré

# Avant-propos

Quelques années passées à initier les élèves à l'informatique avec différents langages me poussent à dire que le langage *Python* est sans doute un langage qui s'apprend plus vite sans pour autant être limité. Les ordinateurs ont progressé et il n'est plus souvent nécessaire d'utiliser des langages tels que le *C* certes plus rapide et plus connu mais aussi plus difficile à maîtriser. A partir de 2004, le langage *Python* a été utilisé à l'ENSAE [1] pour initier les élèves à la programmation et, depuis cette date, j'ai pu observer un regain d'intérêt des élèves pour cette matière qui s'appuyait auparavant sur le langage *C*. Certains étudiants avouent même s'être amusés ; leur projet de programmation n'est plus centré sur un langage mais sur les résultats qu'il permet d'obtenir.

Le langage *Python* suffit pour la plupart des besoins d'un ingénieur et permet de réaliser beaucoup plus rapidement des programmes informatiques qui fonctionnent sur les principaux systèmes d'exploitation. Internet fournit également un panel très large de librairies pour faire du calcul scientifique, des jeux ou encore des sites Internet. Il suffit parfois de quelques minutes pour télécharger une extension et s'en servir. Malgré ces avantages, il existe peu de livres de langue française sur ce langage, beaucoup moins que ceux écrits en langue anglaise dont quelques uns seulement sont traduits.

Ce livre n'aborde pas des sujets comme la programmation web ou l'utilisation conjointe des langages *Python* et *SQL*. La description du langage couvre les besoins qui sont ceux d'un ingénieur, c'est-à-dire une personne dont la programmation ne constitue pas l'essentiel de son travail. Chaque notion est illustrée par des exemples [2]. Ceux-ci sont particulièrement présents lors de l'introduction de notions pas toujours évidentes à comprendre comme les threads, les interfaces graphiques et l'utilisation de *C++*. Le livre se ferme sur plusieurs chapitres ne contenant que des exercices et leurs corrections. Ces exercices sont de deux types : des travaux dirigés et des problèmes courts. Ces derniers ont été posés lors des examens à l'ENSAE et ont été conçus pour préparer les élèves à répondre aux questions techniques lors des entretiens d'embauche.

Le langage *Python* évolue assez vite, il est de plus en plus utilisé ce qui explique la multitude d'informations qu'on peut trouver sur Internet. Le site officiel [3] recense la plupart des extensions (plus d'une centaine) accessibles gratuitement. Il existe également de nombreux sites d'aide comme *WikiPython* [4] qui recense des exemples de programmes sur des sujets variés. L'anglais est la langue d'échange par excellence

1. Ecole Nationale de la Statistique et de l'Administration Economique, http://www.ensae.fr/
2. L'ensemble de ces exemples est accessible depuis le site http://www.xavierdupre.fr.
3. http://www.python.org/
4. http://wikipython.flibuste.net/

dans le monde informatique et le langage *Python* n'y échappe pas même si les moteurs de recherches retournent presque toujours des résultats en français pour des requêtes concernant *Python*, à condition toutefois d'y insérer un mot de syntaxe exclusivement française.

Il n'est pas utile de lire tous les chapitres pour savoir programmer. Le chapitre 1 décrit l'installation du langage *Python* et son utilisation via des éditeurs de texte. Les chapitres 2 et 3 introduisent les premiers concepts, à partir de là, il est déjà possible d'écrire des programmes conséquents et notamment de lire les chapitres 11 à 12 pour s'exercer.

Les chapitres 4 et 5 s'intéressent à la programmation objet, qui n'est pas indispensable dans un premier temps. La programmation est utilisée en tant qu'outil pour atteindre un objectif, réaliser une étude ou tester un modèle. Nombreux sont les élèves qui choisissent de contourner les classes et les objets sans pour autant nuire à la qualité de leur travail ; les programmes qu'ils conçoivent sont sans doute moins lisibles mais ils parviennent à l'objectif fixé.

Le chapitre 6 présente comment créer ses propres extensions. Les derniers paragraphes ne concernent que des utilisateurs experts puisqu'ils parlent de l'utilisation conjointe des langages *Python* et *C++* à partir d'un exemple conçu à cet effet et utilisable en peu de temps [5]. Cette technique est intéressante dans des domaines où les calculs doivent être impérativement très rapides comme en finance, matière à laquelle sont formés les étudiants de l'ENSAE.

Le chapitre 7 introduit les fichiers où comment conservées des résultats sur disque dur. Le chapitre 8 présente les interfaces graphiques et le chapitre 9 les threads. Les chapitres 10 à 12 ne contiennent que des exercices et problèmes sur lesquels plusieurs promotions de l'ENSAE ont été évaluées.

5. Concernant ce point, cet exemple téléchargeable permet de construire entièrement un module *Python* écrit en *C++*, l'ajout d'une fonctionnalité ne prenant que quelques minutes. Cet exemple fonctionne sous *Microsoft Windows* avec *Microsoft Visual C++*, il utilise la librairie *Boost Python*. *Microsoft Windows* est le système d'exploitation le plus courant dans les sociétés.

# Table des matières

Première partie

# LE LANGAGE PYTHON

Les programmes informatiques sont souvent l'aboutissement d'un raisonnement, d'une construction, d'une idée, parfois imprécise mais dont le principe général est compris. Et pour valider cette idée, on utilise un langage de programmation qui ne tolère jamais l'imprécision, qui refuse de comprendre au moindre signe de ponctuation oublié, qui est d'une syntaxe si rigide.

On reste parfois indécis devant une erreur qui se produit avec un exemple qu'on a pourtant recopié sans ajout. On compare les deux versions sans faire attention aux petits détails qui ne changent rien pour le concepteur et beaucoup pour le langage. C'est un espace en plus ou en moins, un symbole : oublié... Il est fréquent de ne pas vérifier ces petits détails lorsqu'on commence à programmer. Et ils découragent souvent. On s'habitue peu à peu au fait qu'il ne faut pas confondre parenthèses et crochets, qu'il manque des guillemets, qu'il y a une lettre en plus, ou en moins.

Lorsque sa syntaxe n'est pas respectée, le langage de programmation ne cherche jamais à comprendre l'intention du programmeur. Il faut penser à la vérifier en premier lorsqu'une erreur se produit. Les messages d'erreur, obscurs au premier abord, donnent néanmoins un bon indice pour corriger un programme. Et si cela ne suffisait pas, il ne faut pas hésiter à recopier ce message dans un moteur de recherche sur Internet pour y trouver une indication dès les premiers résultats, et le plus souvent en français.

Cette première partie est consacrée à la description du langage *Python*. Il est parfois utile de reproduire un des exemples, de le faire fonctionner, de le modifier, d'y introduire des erreurs puis de les corriger. Plus tard, dans de plus longs programmes, les mêmes erreurs seront plus difficiles à déceler et l'expérience montre qu'on fait presque toujours les mêmes erreurs.

# Chapitre 1

# Introduction

---

Ce chapitre s'intéresse tout d'abord à l'installation du langage *Python* et à la réalisation d'un premier programme avec des instructions dont le sens est intuitif. Les derniers paragraphes présentent de nombreuses extensions disponibles sur Internet. Elles rendent le langage *Python* très attractif dans des domaines variés. Ces extensions témoignent que ce langage emporte l'adhésion de nombreux informaticiens qui en retour assurent sa pérennité. Il permet de relier facilement différents éléments, différentes applications. C'est une des raisons de son succès.

## 1.1 Ordinateur et langages

Il est rare aujourd'hui de ne pas avoir déjà entendu ou lu les termes informatiques définis ci-dessous. C'est un rapide rappel.

### 1.1.1 L'ordinateur

On peut considérer simplement qu'un ordinateur est composé de trois ensembles : le microprocesseur, la mémoire, les périphériques. Cette description n'a pas varié en cinquante ans depuis qu'un scientifique du nom de von Neumann l'a imaginée.

Le microprocesseur est le cœur de l'ordinateur, il suit les instructions qu'on lui donne et ne peut travailler qu'avec un très petit nombre d'informations. Sa vitesse se mesure en GigaHertz (GHz) qui correspondent au nombre d'opérations qu'il est capable d'effectuer en une seconde et en nombre de cœurs qui détermine le nombre d'opérations en parallèle qu'il est capable d'exécuter. On lui adjoint une mémoire avec laquelle il échange sans arrêt des données. Sa capacité se mesure en octets (kilo-octets, mégaoctets, gigaoctets ou leurs abréviations Ko, Mo, Go [1]). Ces échanges entre processeur et mémoire sont rapides.

Les périphériques regroupent tout le reste (écran, clavier, souris, disque dur, imprimante...). Ils sont principalement de deux types : les périphériques de stockages (disque dur, DVD) et ceux qui nous permettent de dialoguer avec l'ordinateur, que ce soit pour afficher, sonoriser (écran, enceintes) ou pour recevoir (souris, clavier, webcam, micro...).

1. Un kilo-octets équivaut à $1024 = 2^{10}$ octets, un Mo à 1024 Ko et un Go à 1024 Mo.

### 1.1.2 Termes informatiques

Certains termes reviendront fréquemment dans le livre. Ils sont souvent utilisés comme si tout le monde les connaissait comme la notion d'algorithme qui paraît plutôt abstraite si on se réfère à sa définition dans le dictionnaire. Dans la pratique, on confond souvent algorithme avec programme informatique.

> **Définition 1.1 : algorithme**
> Un algorithme est une suite finie de règles à appliquer dans un ordre déterminé à un nombre fini de données pour arriver avec certitude, en un nombre fini d'étapes, à un certain résultat et cela, indépendamment des données.

Leur écriture est indépendante du langage choisi, qu'il soit écrit en *Basic*, en *Pascal*, en *C*, en *Perl*, en *PHP*, en *Python*, en français, un algorithme reste le même. Pour les algorithmes simples (un tri par exemple), le passage d'un langage à l'autre consiste souvent à traduire mot-à-mot, ce qui favorise l'apprentissage d'un nouveau langage informatique lorsqu'on en connaît déjà un. Les différences entre ces langages résident dans leur spécialisation, le *Visual Basic* permet de piloter les applications *Microsoft Office*, le *PHP*, le *JavaScript* sont dédiés à la programmation Internet. Le langage *Python* n'est pas spécialisé. Il est souvent moins efficace que chaque langage appliqué à son domaine de prédilection mais il peut éviter l'apprentissage d'une syntaxe différente.

> **Définition 1.2 : programme**
> Un programme informatique est une suite d'instructions ou séquence d'instructions. C'est la réalisation informatique d'un ou plusieurs algorithmes. Il dépend du langage.

> **Définition 1.3 : compilateur et compilation**
> Le compilateur est un programme qui traduit un code écrit dans un langage de programmation en langage dit "machine", compréhensible par l'ordinateur. La compilation est le fait de traduire un programme afin que l'ordinateur le comprenne.

> **Définition 1.4 : langage interprété**
> Un langage interprété est converti en instructions propres à la machine au fur et à mesure de son exécution.

Le langage *Python* n'est pas un langage compilé car un programme *Python* n'est pas traduit en langage machine, il est un langage interprété. Entre son écriture et son exécution, il n'y a pas d'étape intermédiaire telle que la compilation et on peut ainsi tester un programme plus rapidement même si son exécution est alors plus lente.

> **Définition 1.5 : mot-clé**
> Un mot-clé est une composante du langage et fait partie de sa grammaire qui comprend également les opérateurs numériques.

La table 3.1 (page 58) regroupe les mots-clés du langage *Python*. Elle contient peu de

mots-clés : son concepteur s'est attaché à créer un langage objet avec la grammaire la plus simple possible.

> **Définition 1.6 : instruction**
> Ce terme est assez vague et dépend en général du langage. On peut considérer qu'une instruction est une expression syntaxiquement correcte pour un langage donné.

Les instructions sont très courtes et tiennent sur une ligne mais si on n'en prend qu'une partie, elle perd son sens. Ce serait comme considérer une phrase avec un verbe transitif mais sans son complément d'objet direct.

## 1.2 Présentation du langage *Python*

### 1.2.1 Histoire résumée

*Python* est un langage objet interprété de haut niveau, il a été créé au début des années quatre-vingt-dix par Guido van Rossum. Entre 1995 et 2001, Rossum a changé plusieurs fois de travail tout en continuant l'élaboration du langage *Python*. En 2001, la PSF (Python Software Foundation) est créée. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de *Python*. Il est depuis distribué sous forme de logiciel libre. *Python* est couvert par sa propre licence [2]. Toutes les versions depuis la 2.0.1 sont compatibles avec la licence GPL [3]. Selon Wikipedia [4], Rossum travaillerait maintenant pour l'entreprise *Google* qui propose une plate-forme de développement de sites web en *Python* [5] et qui héberge de nombreux projets en langage *Python* via son projet *Google Code* [6].

### 1.2.2 Le langage en quelques points

On distingue plusieurs classes parmi les langages informatiques selon la syntaxe qu'ils proposent ou les possibilités qu'ils offrent. *Python* est un langage : interprété, orienté objet, de haut niveau, modulaire, à syntaxe positionnelle, au typage dynamique.

Le langage *Python* est dit *interprété* car il est directement exécuté sans passer par une phase de compilation qui traduit le programme en langage machine, comme c'est le cas pour le langage *C*. En quelque sorte, il fonctionne autant comme une calculatrice que comme un langage de programmation. Afin d'accélérer l'exécution d'un programme *Python*, il est traduit dans un langage intermédiaire [7] qui est ensuite interprété par une machine virtuelle *Python*. Ce mécanisme est semblable à celui propre au langage *Java*.

Le langage est orienté objet car il intègre le concept de classe ou d'objet. Un objet

2. Voir le site http://www.python.org/psf/license/, cette licence autorise les usages commerciaux.
3. Ou GNU Public Licence (voir le site http://www.gnu.org/copyleft/gpl.html).
4. http://fr.wikipedia.org/wiki/Python_(langage)
5. http://code.google.com/appengine/
6. http://code.google.com/
7. Celui-ci est appelé *bytecode*. Ce mot ne signifie pas grand chose excepté que ce *bytecode* est seulement compréhensible par la machine virtuelle *Python*.

regroupe un ensemble de données et un ensemble de fonctionnalités attachées à ces données. Ce concept relie la description des données et les algorithmes qui leur sont appliqués comme un tout indissociable. Un objet est en quelque sorte une entité autonome avec laquelle on peut communiquer via une interface.

On considère que le langage *Python* est de haut niveau car il propose des fonctionnalités avancées et automatiques comme le *garbage collecting*. Cette tâche correspond à la destruction automatique des objets lorsqu'ils ne sont plus utilisés. Cette fonctionnalité est proposée par la plupart des langages interprétés ou encore *Java* mais pas *C++*. Il propose également des structures de données complexes éloignées des types numériques standards telles que des dictionnaires.

Le langage *Python* est modulaire. La définition du langage est très succincte et autour de ce noyau concis, de nombreuses librairies ou modules ont été développées. *Python* est assez intuitif. Etre à l'aise avec ce langage revient à connaître tout autant sa syntaxe que les nombreux modules disponibles. Comme il est assez simple de construire un pont entre *Python* et *C++*, de nombreux projets open source antérieurs à *Python* sont maintenant accessibles dans ce langage et sous toutes les plateformes.

Le langage *Python* est à syntaxe positionnelle en ce sens que l'indentation fait partie du langage. Le point virgule permet de séparer les instructions en langage *C*, l'accolade permet de commencer un bloc d'instructions. En *Python*, seule l'indentation permet de marquer le début et la fin d'un tel bloc, ce procédé consiste à décaler les lignes vers la droite pour signifier qu'elles appartiennent au même bloc d'instructions et qu'elles doivent être exécutées ensemble. Cette contrainte rend les programmes *Python* souvent plus faciles à lire.

Le langage *Python* est à typage dynamique. Le type d'une variable, d'une donnée est définie lors de l'exécution du programme. Chaque information est désignée par un identificateur qui est manipulé tout au long du programme. Le fait que telle ou telle opération soit valide est vérifiée au moment où elle doit être réalisée et non au moment où le programme est traduit en langage machine, c'est-à-dire compilé.

### 1.2.3 Avantages et inconvénients du langage *Python*

Alors qu'il y a quelques années, le langage *C* puis *C++* s'imposaient souvent comme langage de programmation, il existe dorénavant une profusion de langages (*Java*, *JavaScript*, *PHP*, *Visual Basic*, *C#*, *Perl*, *PHP*, ...). Il est souvent possible de transposer les mêmes algorithmes d'un langage à un autre. Le choix approprié est bien souvent celui qui offre la plus grande simplicité lors de la mise en œuvre d'un programme même si cette simplicité s'acquiert au détriment de la vitesse d'exécution. Le langage *PHP* est par exemple très utilisé pour la conception de sites Internet car il propose beaucoup de fonctions très utiles dans ce contexte. Dans ce domaine, le langage *Python* est une solution alternative intéressante.

Comme la plupart des langages, le langage *Python* est tout d'abord portable puisqu'un même programme peut être exécuté sur un grand nombre de systèmes d'exploitation comme *Linux*, *Microsoft Windows*, *Mac OS X*... *Python* possède également l'avantage d'être entièrement gratuit tout en proposant la possibilité de pouvoir réaliser des applications commerciales à l'aide de ce langage.

Si le langage *C* reste le langage de prédilection pour l'implémentation d'algorithmes complexes et gourmands en temps de calcul ou en capacités de stockage, un langage tel que *Python* suffit dans la plupart des cas. De plus, lorsque ce dernier ne convient pas, il offre toujours la possibilité, pour une grande exigence de rapidité, d'intégrer un code écrit dans un autre langage tel que le *C/C++* [8] ou *Java*, et ce, d'une manière assez simple. La question du choix du langage pour l'ensemble d'un projet est souvent superflue, il est courant aujourd'hui d'utiliser plusieurs langages et de les assembler. Le langage *C* reste incontournable pour concevoir des applications rapides, en revanche, il est de plus en plus fréquent d'"habiller" un programme avec une interface graphique programmée dans un langage tel que *Python*.

En résumé, l'utilisation du langage *Python* n'est pas restreinte à un domaine : elle comprend le calcul scientifique, les interfaces graphiques, la programmation Internet. Sa gratuité, la richesse des extensions disponibles sur Internet le rendent séduisant dans des environnements universitaire et professionnel. Le langage est vivant et l'intérêt qu'on lui porte ne décroît pas. C'est un critère important lorsqu'on choisit un outil parmi la myriade de projets open source. De plus, il existe différentes solutions pour améliorer son principal point faible qui est la vitesse d'exécution.

## 1.3 Installation du langage *Python*

### 1.3.1 Installation du langage *Python*

*Python* a l'avantage d'être disponible sur de nombreuses plates-formes comme *Microsoft Windows*, *Linux* ou *Mac OS X*. L'installation sous *Windows* est simple. Le langage *Python* est déjà intégré aux systèmes d'exploitation *Linux* et *Mac OS X*.

Sous *Microsoft Windows*, il suffit d'exécuter le fichier *python-2.6.5.msi* [9] ou tout autre version plus récente. En règle générale, il est conseillé de télécharger la dernière version stable du langage, celle avec laquelle le plus grande nombre d'extensions seront compatibles, en particulier toutes celles dont vous avez besoin. Les options d'installation choisies sont celles par défaut, le répertoire d'installation est par défaut C:/Python26. A la fin de cette installation apparaît un menu supplémentaire dans le menu *Démarrer* (ou *Start*) de *Microsoft Windows* comme le montre la figure 1.1. Ce menu contient les intitulés suivant :

| | |
|---|---|
| IDLE (Python GUI) | éditeur de texte, pour programmer |
| Module Docs | pour rechercher des informations dans la documentation |
| Python (command line) | ligne de commande *Python* |
| Python Manuals | documentation à propos du langage *Python* |
| Uninstall Python | pour désinstaller *Python* |

La documentation (de langue anglaise [10]) décrit en détail le langage *Python*, elle inclut également un tutoriel [11] qui permet de le découvrir. La ligne de commande (voir

8. Ce point est d'ailleurs abordé au paragraphe 6.5, page 158.
9. Cette version est disponible à l'adresse http://www.python.org/download/. C'est la version utilisée pour développer les exemples présents dans ce livre.
10. http://www.python.org/doc/
11. La requête *python français* de nombreux sites de documentation sur *Python* en français.

**Figure 1.1 :** *Menu ajouté à Microsoft Windows.*

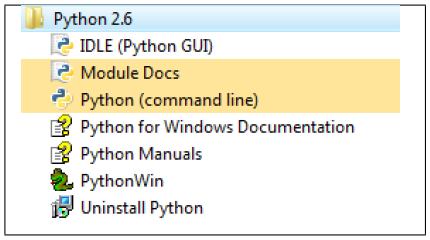



figure 1.2) permet d'exécuter des instructions en langage *Python*. Elle est pratique pour effectuer des calculs mais il est nécessaire d'utiliser un éditeur de texte pour écrire un programme, de le sauvegarder, et de ne l'exécuter qu'une fois terminé au lieu que chaque ligne de celui-ci ne soit interprétée immédiatement après qu'elle a été écrite comme c'est le cas pour une ligne de commande.

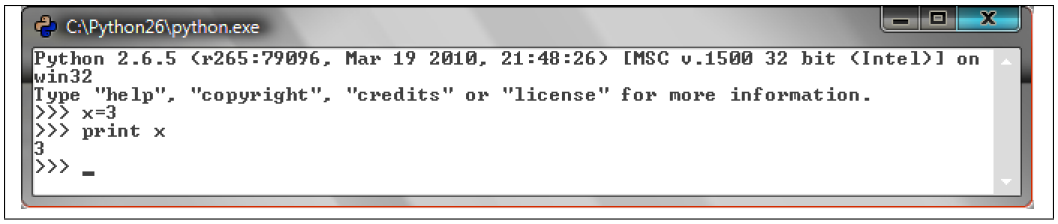



**Figure 1.2 :** *Ligne de commande, la première ligne affecte la valeur 3 à la variable* `x`*, la seconde ligne l'affiche.*

### 1.3.2 Particularité de *Mac OS X* et *Linux*

Le langage *Python* est déjà présent sur les ordinateurs d'*Apple* si ceux-ci sont équipés du système d'exploitation *Mac OS X*. Il est également présent dans les distributions *Linux*. L'installation de *Python* est parfois rendue nécessaire si on désire utiliser la dernière version du langage. L'installation des extensions du langage est similaire à celle d'autres extensions *Linux*.

Il ne faut pas oublier de vérifier la version du langage *Python* installée sur l'ordinateur *Linux* et *Mac OS X*. Il suffit pour cela de taper la ligne `help()` pour en prendre connaissance. Cela signifie que toutes les extensions qui doivent être installées doivent l'être pour cette version qu'il est néanmoins possible de mettre à jour [12].

### 1.3.3 Utilisation de l'éditeur de texte

La figure 1.1 montre le menu installé par *Python* dans le menu "Démarrer" de *Microsoft Windows*. En choisissant l'intitulé "IDLE (Python GUI)", on active la fenêtre de commande de *Python*(voir figure 1.3). Les instructions sont interprétées

12. voir la page http://www.python.org/download/mac/ ou http://www.python.org/download/linux/

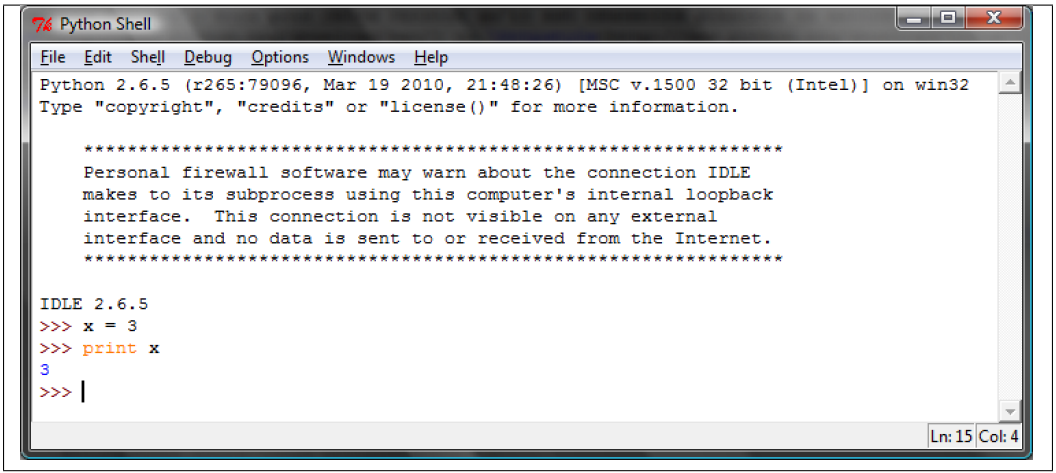



**Figure 1.3 :** *Fenêtre de commande fournie avec le langage Python.*

au fur et à mesure qu'elles sont tapées au clavier. Après chaque ligne, cette fenêtre de commande conserve la mémoire de tout ce qui a été exécuté. Par exemple :

```
>>> x = 3
>>> y = 6
>>> z = x * y
>>> print z
18
>>>
```

Après l'exécution de ces quelques lignes, les variables [13] `x`, `y`, `z` existent toujours. La ligne de commande ressemble à une calculatrice améliorée. Pour effacer toutes les variables créées, il suffit de redémarrer l'interpréteur par l'intermédiaire du menu `Shell->Restart Shell`. Les trois variables précédentes auront disparu.

Il n'est pas possible de conserver le texte qui a été saisi au clavier, il est seulement possible de rappeler une instruction déjà exécutée par l'intermédiaire de la combinaison de touches `ALT + p`, pressée une ou plusieurs fois. La combinaison `ALT + n` permet de revenir à l'instruction suivante. Pour écrire un programme et ainsi conserver toutes les instructions, il faut d'actionner le menu `File->New Window` qui ouvre une seconde fenêtre qui fonctionne comme un éditeur de texte (voir figure 1.4).

Après que le programme a été saisi, le menu `Run→Run Module` exécute le programme. Il demande au préalable s'il faut enregistrer le programme et réinitialise l'interpréteur *Python* pour effacer les traces des exécutions précédentes. Le résultat apparaît dans la première fenêtre (celle de la figure 1.3). La pression des touches "Ctrl + C" permet d'arrêter le programme avant qu'il n'arrive à sa fin.

**Remarque 1.7 : fenêtre intempestive**
Si on veut se débarrasser de cette fenêtre intempestive qui demande confirma-

13. Les variables sont définies au chapitre 2. En résumé, on peut considérer une variable comme une lettre, un mot qui désigne une information. Dans cet exemple, les trois variables désignent des informations numériques qu'on peut manipuler comme si elles étaient effectivement des nombres. Les variables supprimées, les informations qu'elles désignaient ne plus accessibles et sont perdues également.

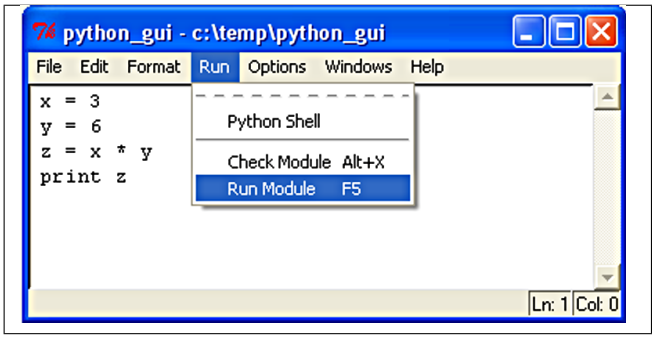



**Figure 1.4 :** *Fenêtre de programme, le menu* `Run→Run Module` *permet de lancer l'exécution du programme. L'interpréteur Python est réinitialisé au début de l'exécution et ne conserve aucune trace des travaux précédents.*

tion pour enregistrer les dernières modifications, il suffit d'aller dans le menu `Options->Configure IDLE` et de choisir `No prompt` sur la quatrième ligne dans la rubrique `General` (voir figure 1.5). Ce désagrément apparaît sur la plupart des éditeurs de texte et se résout de la même manière.

Cette description succincte permet néanmoins de réaliser puis d'exécuter des programmes. Les autres fonctionnalités sont celles d'un éditeur de texte classique, notamment la touche `F1` qui débouche sur l'aide associée au langage *Python*. Il est possible d'ouvrir autant de fenêtres qu'il y a de fichiers à modifier simultanément.

Néanmoins, il est préférable d'utiliser d'autres éditeurs plus riches comme celui proposé au paragraphe 1.4.1. Ils proposent des fonctions d'édition plus évoluées que l'éditeur installé avec *Python*.

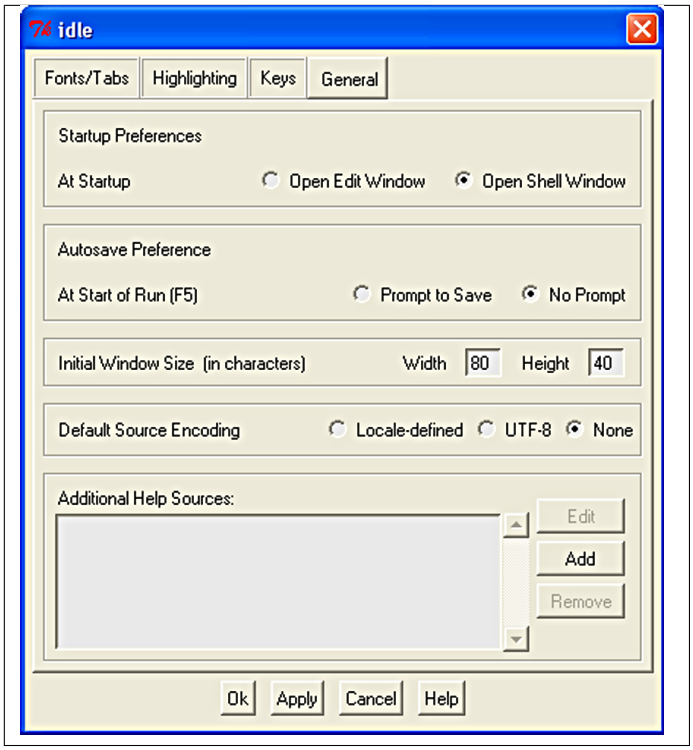



**Figure 1.5 :** *Cliquer sur* `General` *puis sur* `Noprompt` *pour se débarrasser de la fenêtre intempestive qui demande à l'utilisateur de confirmer la sauvegarde des dernières modifications. D'autres éditeurs de texte se comportent de la même manière et disposent d'une option similaire.*

## 1.4 Installation d'un éditeur de texte

### 1.4.1 Editeur SciTe

Il existe de nombreux éditeurs de texte, payants ou gratuits. Le premier proposé dans ce livre est *SciTe*, il est gratuit et fonctionne avec de nombreux compilateurs de langages différents (voir figure 1.6). L'éditeur *SciTe*[14] est léger et son utilisation ne nécessite pas de droits spécifiques[15]. La gestion des erreurs est également plus pratique puisqu'il suffit de cliquer sur une erreur pour que l'éditeur positionne automatiquement le curseur sur la ligne ayant généré cette erreur.

Une fois installé, il faut configurer l'éditeur de manière à ce qu'il puisse utiliser le compilateur *Python*. Il faut cliquer sur le menu `Option` et choisir la rubrique `Open python.properties` (voir figure 1.6).

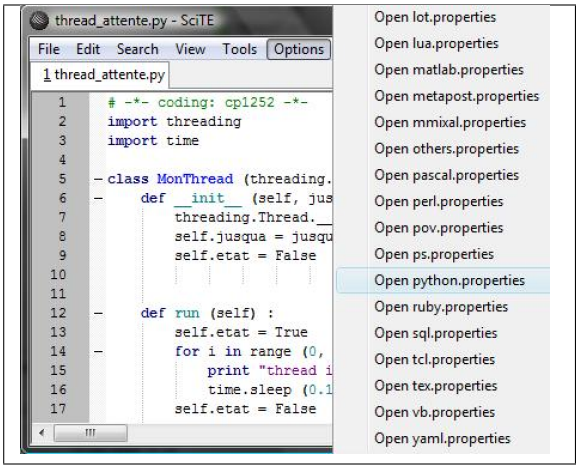



**Figure 1.6 :** *Configuration de l'éditeur de texte SciTe. Il faut cliquer sur le menu* `Option`*, puis sur* `Open python.properties` *pour ouvrir les paramètres de configuration associées au langage Python.*

Après avoir sélectionné ce menu, un fichier texte s'ouvre. Vers la fin, on trouve les lignes suivantes qui contiennent la ligne de commande qui permet d'exécuter un programme *Python* :

```
if PLAT_WIN
    command.go.*.py=python -u "$(FileNameExt)"
    command.go.subsystem.*.py=0
    command.go.*.pyw=pythonw -u "$(FileNameExt)"
    command.go.subsystem.*.pyw=1
```

Il suffit de préciser le répertoire où se trouve l'interpréteur qui devrait être `c : /python26` sur *Microsoft Windows* si le répertoire d'installation par défaut du langage *Python* n'a pas été modifié.

14. Cet éditeur est accessible depuis l'adresse http://www.scintilla.org/SciTE.html et téléchargeable depuis l'adresse http://scintilla.sourceforge.net/SciTEDownload.html.

15. Il arrive fréquemment que des universités, des sociétés interdisent à leurs utilisateurs l'installation de nouveaux logiciels sur leurs machines pour des raisons de sécurité. Il faut des droits spécifiques pour contourner cette interdiction.

```
if PLAT_WIN
    command.go.*.py=c:\Python26\python -u "$(FileNameExt)"
    command.go.subsystem.*.py=0
    command.go.*.pyw=c:\Python26\pythonw -u "$(FileNameExt)"
    command.go.subsystem.*.pyw=1
```

Une fois ce fichier enregistré, il ne reste plus qu'à écrire un programme *Python*. Une fois terminé, la touche `F5` lance son exécution. L'exemple de la figure 1.7 montre un cas où l'exécution a été interrompue par une erreur.

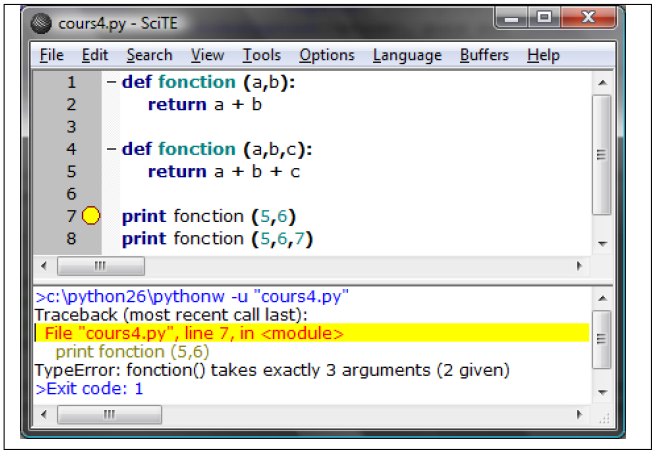



**Figure 1.7 :** *Exécution d'un programme interrompue par une erreur. Pour cet éditeur, il suffit de cliquer sur le message d'erreur pour placer le curseur sur la ligne erronée.*

Le site Internet de *SciTe* propose quelques trucs et astuces pour configurer l'éditeur à votre convenance. Comme à peu près tous les logiciels, il dispose d'une rubrique FAQ[16] (Frequently Asked Questions) qui recense les questions et les réponses les plus couramment posées. Cette rubrique existe pour la plupart des éditeurs et plus généralement pour la plupart des applications disponibles sur Internet.

### 1.4.2 Python Scripter

Le second éditeur proposé dans ce livre est *Python Scripter*[17] (voir figure 1.8). Il est plus lourd mais offre plus de fonctionnalités notamment un débugger qui permet d'exécuter pas à pas un programme afin de vérifier chaque ligne de son exécution. Lorsqu'un programme produit de mauvais résultats, la première solution consiste à afficher de nombreux résultats intermédiaires mais cela nécessite d'ajouter des instructions un peu partout dans le programme. La seconde méthode est l'utilisation du débugger. Sans modifier le code informatique, il est possible d'arrêter temporairement l'exécution à une ligne donnée pour inspecter le contenu des variables à cet instant de l'exécution. Ce logiciel fonctionne sans autre manipulation supplémentaire autre que son installation.

Cet éditeur, contrairement à *SciTe*, ne fonctionne qu'avec *Python* et ne peut être utilisé avec d'autres langages. Son apparence est adaptée au développement. Il per-

16. http://scintilla.sourceforge.net/SciTEFAQ.html
17. http://mmm-experts.com/

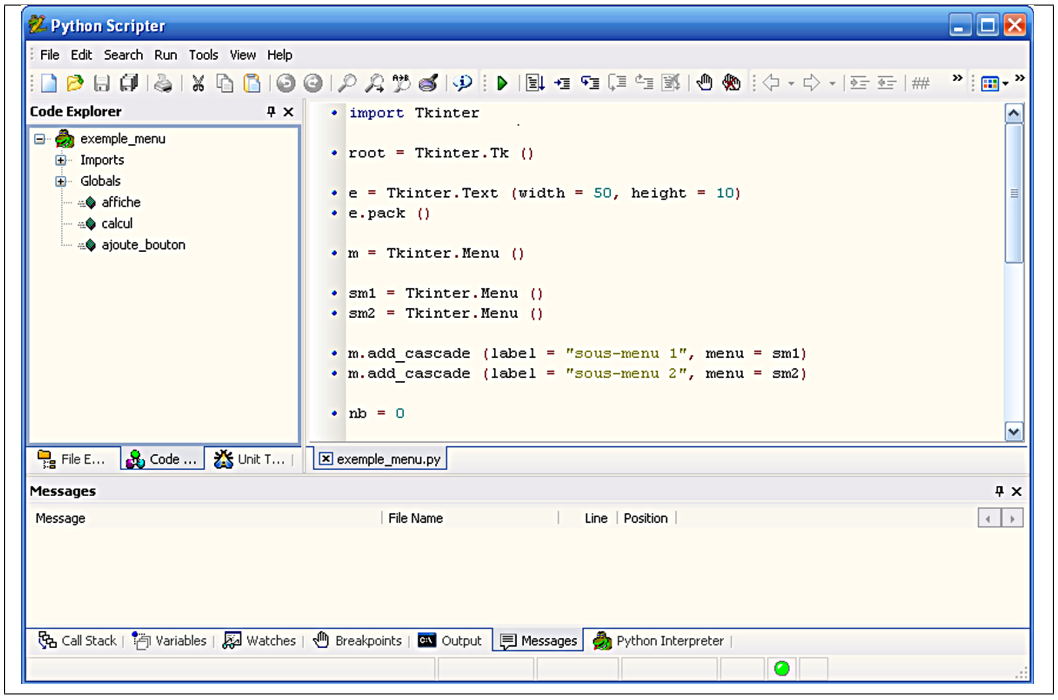



**Figure 1.8 :** *Copie d'écran de l'éditeur Python Scripter.*

met également de voir la liste des variables, fonctions, classes, présentes dans un programme.

### 1.4.3 Eclipse et PyDev

*Eclipse*[18] est un environnement complet de développement adapté à une multitude de langages dont *Python* via une extension appelé *PyDev*[19]. Cette solution s'adresse plus à des gens ayant déjà une expérience en informatique ou souhaitant construire un projet conséquent. En constante évolution, *Eclipse* est un éditeur très complet mais son utilisation nécessite un apprentissage non négligeable qui n'est pas indispensable pour des projets de petite taille.

### 1.4.4 Côté *Mac OS X* : *TextWrangler*

Un éditeur possible est *TextWrangler*[20] qui est gratuit. Lorsqu'on programme à la fois sous *Microsoft Windows* et *Linux* ou *Mac OS X*, les fichiers contiennent des codes de fins de lignes différents et l'éditeur vous le fait remarquer à chaque exécution. Pour éviter ce désagrément, il convient d'écraser le fichier contenant le programme en le sauvegardant à nouveau en prenant soin de choisir des fins de lignes identiques à celles du monde *Linux*. *Linux* fait également la différence entre les minuscules et les majuscules en ce qui concerne les noms de fichiers contrairement

18. http://www.eclipse.org/
19. http://pydev.sourceforge.net/
20. http://www.barebones.com/products/textwrangler/

à *Microsoft Windows*[21]. L'environnement de développement `XCode` est aussi une solution.

### 1.4.5 Autres éditeurs

Certains éditeurs de texte sont eux-mêmes écrits en *Python*, ce qui leur permet de fonctionner sur toutes les plates-formes. Trois sont assez connues, *DrPython*[22], *Boa Constructor*[23], *The Eric Python IDE*[24]. Ce dernier s'appuie sur une interface graphique de type `pyQt`[25], son interface est riche, presque trop lorsqu'on commence à programmer. *Boa Constructor* s'appuie sur `wxPython` et offre des outils permettant de dessiner des interfaces graphiques. *DrPython* est le plus simple des trois.

Certains éditeurs connus du monde *Linux* ont été déclinés pour le monde *Windows*, c'est le cas de *Emacs*[26]. *SciTe* est souvent utilisé car c'est le plus simple. Les autres offrent des fonctionnalités variées dont l'utilité n'apparaît que pour des programmes conséquents. Le site officiel[27] répertorie d'autres éditeurs de texte dont certains sont de véritables environnements de développements. Il n'est pas aberrant d'en utiliser plus d'un car chacun d'eux est plus approprié dans telle situation ou pour tel type de programme.

### 1.4.6 Tabulations

De nombreux éditeurs utilisent les tabulations pour mettre en forme un programme et tenir compte des décalages (indentation) d'un bloc par rapport à un autre. Il est conseillé de remplacer ces tabulations par des espaces, cela évite les confusions et les mauvaises interprétations lorsque tabulations et espaces sont mélangés. La taille d'une tabulation peut varier d'un éditeur à l'autre contrairement aux espaces.

## 1.5 Premier programme

Après avoir installé le langage *Python* ainsi qu'un éditeur, il suffit d'écrire quelques lignes pour vérifier que cette première étape s'est achevée correctement. Tous les programmes *Python* porte l'extension `.py`.

### 1.5.1 Afficher le résultat d'un calcul

Le programme suivant, qu'on peut appeler `premier.py`, affiche le message `premier message` une fois qu'il est exécuté. Le mot-clé `print` précédant le message - toujours entre guillemets ou entre apostrophes sauf si ce sont des nombres - stipule qu'il faut afficher quelque chose.

21. L'usage de la commande `dos2unix` peut se révéler utile.
22. http://drpython.sourceforge.net/
23. http://boa-constructor.sourceforge.net/
24. http://www.die-offenbachs.de/eric/index.html
25. Voir le paragraphe 1.6.5 sur les interfaces graphiques.
26. http://www.gnu.org/software/emacs/windows/
27. plus précisément à l'adresse http://wiki.python.org/moin/PythonEditors ou encore à l'adresse http://wiki.python.org/moin/IntegratedDevelopmentEnvironments

```
print "premier message"
```

Un programme ne barbouille pas l'écran de texte indésirable, il n'affiche que ce que le programmeur lui demande d'afficher et cette intention se matérialise grâce à l'instruction `print`. On peut ensuite écrire quelques lignes qu'on pourrait également taper sur une calculatrice :

```
print 3.141592 * 5*5
print 3.141592 * 5**2
```

Ces deux lignes affichent 78.5398 qui est la surface d'un cercle de 5 mètres de côté. Le symbole $**$ correspond à l'opération puissance. Il existe une constante PI ou $\pi$ mais qui ne fait pas partie du langage lui-même : elle est définie dans une extension interne[28] du langage qui contient bon nombre de fonctions mathématiques telles que les cosinus, sinus...

```
from math import *  # permet d'utiliser les extensions mathématiques de Python
print pi * 5**2
```

**Remarque 1.8 :** `print` ***Python* 3.0**
A partir de la version 3.0 du langage *Python*, la syntaxe de `print` change et il faudra dorénavant utiliser des parenthèses. Il est suggéré d'utiliser cette syntaxe qui est valide quelque soit la version du lanage.

```
print (3.141592 * 5*5)
```

### 1.5.2 Les accents

Le langage *Python* est conçu pour le monde anglophone et l'utilisation des accents ne va pas de soi. Le programme suivant qui demande d'afficher un message contenant un accent provoque l'apparition d'une erreur :

```
print "accentué"
```

L'erreur est la suivante :

```
  File "essai.py", line 1
SyntaxError: Non-ASCII character '\xe9' in file i.py on line 1,
             but no encoding declared;
             see http://www.python.org/peps/pep-0263.html for details
```

Dans ce cas, il faut ajouter une ligne placée en première position qui précise que des accents pourront être utilisés. L'exécution du programme qui suit ne soulève aucune erreur.

28. Cette extension est installée automatiquement avec *Python* contrairement à une extension externe.

```
# coding: latin-1
print "accentué"
```

Cette ligne précise en fait que l'interpréteur *Python* doit utiliser un jeu de caractères spécial. Ces jeux de caractères ont parfois leur importance, les navigateurs n'aiment pas trop les accents au sein des adresses Internet : il est parfois préférable de ne pas utiliser d'accents sur les sites de réservations d'hôtels, de trains ou d'avions.

Par défaut, l'interpréteur *Python* utilise le jeu de caractères le plus réduit, il faut lui préciser qu'il doit en utiliser un plus étendu incluant les accents. On trouve également sous *Microsoft Windows* la syntaxe équivalente suivante :

```
# coding: cp1252
print "accentué"
```

Enfin, s'il fallait utiliser des caractères autres que latins, il faudrait alors choisir le jeu de caractères `utf − 8` qui s'appliquent aussi aux langues orientales.

```
# coding: utf-8
print "accentué"
```

A cause des accents, la plupart des exemples cités dans ce livre ne fonctionnent pas sans cette première ligne qui a parfois été enlevée lors de l'impression pour des questions de lisibilité. Il faut penser à l'ajouter pour reproduire les exemples. Le paragraphe 7.8 page 191 est plus complet à ce sujet.

**Remarque 1.9 :** #!`/usr/local/bin/python`
Sur Internet, on trouve de nombreux exemples commençant par la ligne suivante :

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

On rencontre souvent cette ligne pour un programme écrit sur une machine *Linux*, elle indique l'emplacement de l'interpréteur *Python* à utiliser. Cette ligne s'appelle un *shebang*.

## 1.6 Installation d'extensions (ou modules externes)

Les extensions ou modules ou packages sont des bibliothèques de fonctions non standard. Ce sont souvent des programmes existant conçus dans un autre langage tel que *C++* et qui ont été "coiffés" d'une interface permettant de les utiliser sous *Python*. Les paragraphes qui suivent décrivent quelques extensions couramment utilisées. Certains besoins reviennent fréquemment comme dessiner un graphe, charger une image, faire du calcul matriciel et *Python* seul ne permet pas de répondre à ce problème.

La première information à connaître avant d'installer un package est la version du langage *Python* installée sur votre ordinateur. Il faut impérativement utiliser la version du module élaborée pour la même version [29]. Il est nécessaire de connaître également le système d'exploitation de votre ordinateur.

29. L'instruction `help`() saisie sur la ligne de commande *Python* retourne entre autres le numéro de version.

Par conséquent, il ne faut pas installer le module `scipy` (calcul scientifique) élaboré pour *Python* 2.5 alors que sur votre ordinateur est installée la version 2.4, il y a de fortes chances que cela ne fonctionne pas. Certains modules dépendent d'autres, lors de leur installation, il faut vérifier que les modules dont ils dépendent ont été préalablement installés.

Le site officiel [30] recense une longue liste de modules existants (plusieurs centaines). Il recense également les dernières mises à jour. Avant de commencer à programmer, il est conseillé d'aller chercher sur ce site ou d'utiliser un moteur de recherche pour savoir si un module ne ferait pas déjà une partie du travail.

Il est parfois intéressant de consulter des forums d'échanges entre programmeurs autour de *Python*. Il est parfois plus intéressant de jeter un œil sur de tels sites [31] avant de se lancer dans la recherche d'un module spécifique. Internet fournit également de nombreux exemples de programmes dont on peut s'inspirer.

Les modules cités dans ce paragraphe ne sont pas installés automatiquement avec le langage *Python* mais sont parfois très utiles dans le cadre d'un travail d'ingénieur. Ils existent depuis plusieurs années et sont pour la plupart régulièrement maintenus à jour. Ils sont tous facilement accessibles depuis un moteur de recherche.

### 1.6.1 Graphes

`matplotlib` [32] : ce module reprend la même logique que le logiciel *MatLab* ou *Octave*. Pour ceux que la finance intéresse, il comporte également des fonctions qui permettent de récupérer des historiques quotidiens de cours d'actions depuis le site *Yahoo ! Finance* [33].

`biggles` [34] : ce module permet de dessiner des courbes (2D, 3D) et de convertir des graphiques sous forme d'images de tout format. Ce module nécessite l'installation de fichiers supplémentaires (libpng, zlib, plotutils) [35].

`pydot` [36] : ce module propose le dessin de graphes dans une image (arbre généalogique, réseau de distribution, ...). Le module `pydot` permet de dessiner tout type de graphe. Ce module nécessite un autre module `pyparsing` [37] et une application *Graphviz* [38]. Avec cet outil, on peut réaliser des graphes comme celui de la figure 1.9. Ce module est utilisé lors de l'exercice 4 page 261.

`pyx` [39] : ce module est aussi une librairie qui permet de tracer des graphes avec la possibilité de convertir directement ceux-ci en fichiers PDF ou EPS.

30. http://pypi.python.org/pypi
31. On peut citer http://wikipython.flibuste.net/ ou http://www.afpy.org/, deux sites français qui regroupent des informations intéressantes sur de nouveaux modules ou des exemples d'utilisation de modules courants.
32. http://matplotlib.sourceforge.net/
33. Avec la fonction `fetch_historical_yahoo`.
34. http://biggles.sourceforge.net/
35. Accessibles depuis l'adresse http://www.gnu.org/software/plotutils/, ces informations sont disponibles depuis la page de téléchargement de `biggles`.
36. http://code.google.com/p/pydot/
37. http://pyparsing.wikispaces.com/
38. http://www.graphviz.org/
39. http://pyx.sourceforge.net/

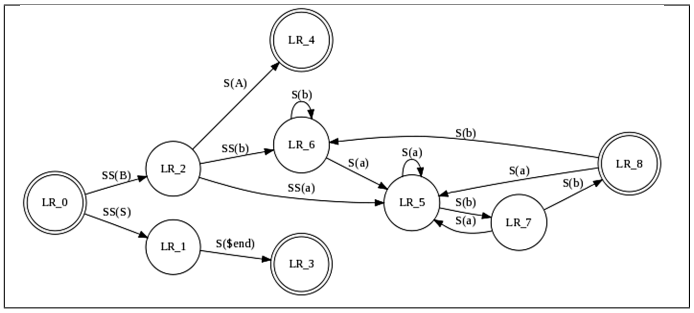



**Figure 1.9 :** *Image d'un graphe, extraite du site http ://www.graphviz.org/ où est disponible l'outil Graphviz. Il permet de représenter des graphes sur une page en minimisant les croisements d'arcs ce qui est un problème complexe.*

Sans utiliser nécessairement une extension, il existe des outils comme *GNUPlot*[40] venu du monde *Linux* qui s'exécutent facilement en ligne de commande. Cet outil propose son propre langage de programmation dédié à la création de graphiques. Une fois ce langage maîtrisé, il est facile de construire un programme qui permet de l'exécuter en ligne de commande pour récupérer l'image du graphique.

**Remarque 1.10 : sortie texte et graphique**
Un des moyens simples de construire un rapport mélangeant résultats numériques et graphiques est de construire un fichier au format HTML[41] puis de l'afficher grâce à un navigateur Internet. Il est même possible de construire un fichier au format *PDF* en utilisant l'outil *Latex*[42].

### 1.6.2 Calculs scientifiques

`numpy`[43] : ce module manipule efficacement les tableaux de grandes dimensions, intègre le calcul matriciel, la transformée de Fourier.

`scipy`[44] : ce module propose de nombreux algorithmes, algorithmes de classification, intégration, optimisation, génération de nombres aléatoires, traitement du signal, interpolation, algorithmes génétiques, algèbre linéaire, transformée de Fourier, dessin de courbes 2D, 3D. Son installation requiert d'abord celle du module `numpy`.

`orange`[45] : c'est un module qui permet de faire ce qu'on appelle du *datamining* ou fouille de données. Il propose de nombreux algorithmes de classifications, d'analyse des données qui ont trait au *machine learning*.

`Rpy`[46] : le langage *R*[47] est un langage dédié aux calculs statistiques. Il est optimisé pour la manipulation de grandes matrices et des calculs statistiques et matriciels rapides. Il dispose également de nombreux graphiques. Il faut au préalable installer *Python* et *R* puis le module `Rpy` fera la liaison entre les deux.

40. http://www.gnuplot.info/
41. Il est facile de trouver des pages Internet décrivant cette syntaxe via un moteur de recherche. L'exercice 7, page 267 en dira plus à ce sujet.
42. voir l'exercice 7, page 268
43. http://numpy.scipy.org/
44. http://www.scipy.org/
45. http://www.ailab.si/orange/
46. http://rpy.sourceforge.net/
47. http://www.r-project.org/

On pourrait également citer le module `mdp`[48] qui offre quelques fonctionnalités relatives à l'analyse des données tels que l'analyse en composante principale (ACP) ou encore le module `cvxopt`[49] qui propose des algorithmes d'optimisation quadratique sous contraintes.

### 1.6.3 Images

`PythonImageLibrary(PIL)`[50] : ce module permet d'utiliser les images, quelque soit leur format. Elle propose également quelques traitements d'images simples, rotation, zoom, histogrammes.

### 1.6.4 Base de données

**SQLObject**[51] : ce module offre la possibilité de se connecter à la plupart des serveurs de bases de données et d'y accéder via une syntaxe objet plutôt qu'au travers de requêtes *SQL*.

**SQLAlchemy**[52] : ce module propose les mêmes fonctionnalités que le précédent. C'est le plus riche des deux. Il est souvent utilisé en association avec le module `elixir`[53].

### 1.6.5 Interfaces graphiques

Le langage *Python* propose un module interne permettant de réaliser des interfaces graphiques : `Tkinter`. Il suffit pour des applications simples qui ne nécessitent que quelques fenêtres. Pour des applications plus ambitieuses qui doivent afficher des données sous forme de matrices par exemple ou pour obtenir un aspect plus professionnel, il est impératif d'utiliser un module externe. Il est néanmoins conseillé de savoir utiliser `Tkinter` et de comprendre la programmation événementielle avant de se lancer dans l'apprentissage de `wxPython`, `pyQt` ou `pyGTK`. Ces trois modules proposent des fonctionnalités équivalentes, le choix porte surtout sur des besoins précis qu'un module couvre et sur des questions d'esthétisme.

`wxPython`[54] : il propose des interfaces graphiques plus complètes que le module standard `Tkinter` comme une fenêtre tableur. Utilisé par de nombreuses applications, il est régulièrement mis à jour. Il est entièrement gratuit et peut servir à des applications commerciales. Il s'inspire des *MFC* ou *Microsoft Fundation Class* issues de *Microsoft* mais a l'avantage d'être portable. Aujourd'hui, c'est avant tout un projet open source en constante évolution.

`pyQt`[55] : il est moins répandu que `wxPython` sans doute car il est produit par une

48. http://mdp-toolkit.sourceforge.net/
49. http://abel.ee.ucla.edu/cvxopt/
50. http://www.pythonware.com/products/pil/
51. http://www.sqlobject.org/
52. http://www.sqlalchemy.org/
53. http://elixir.ematia.de/trac/
54. http://www.wxpython.org/
55. http://www.riverbankcomputing.co.uk/software/pyqt/intro

société et n'est pas open source. Il est réputé pour être mieux structuré. Il existe des restrictions pour la conception d'applications commerciales.

`pyGTK` [56] : il est plus proche de `Tkinter` que les deux autres modules. Venu du monde *Linux*, l'aspect de l'interface lui est plus ressemblant qu'à celui du monde *Windows*.

Lorsqu'on souhaite réaliser un jeu, il est nécessaire d'avoir une interface graphique plus conciliante avec la gestion des images, des dessins, des sons voire des vidéos. C'est le cas du module qui suit.

`pygame` [57] : ce module est prévu pour concevoir des jeux en *Python*. Il permet entre autres d'écouter les morceaux enregistrés sur un CD, d'afficher un film et également d'afficher des images, de récupérer les touches pressées au clavier, les mouvements de la souris, ceux de la manette de jeux. Il suffit d'une heure ou deux pour écrire ses premiers programmes avec `pygame`. Il n'est pas prévu pour réaliser des jeux en 3D mais suffit pour bon nombre de jeux de plateau, jeux de réflexion.

Pour la réalisation d'images et d'animations 3D, on peut utiliser le module `pyOpenGL` [58] qui s'interface avec `pygame`, `wxPython` ou `pyQt`. C'est un module dont la maîtrise ne se compte plus en heures mais en dizaine d'heures tout comme le module `directpython` [59] qui est l'équivalent de `pyOpenGL` mais uniquement sous *Microsoft Windows*.

### 1.6.6 *Microsoft Windows*, *Microsoft Excel*, PDF

`win32com` [60] : ce module permet de communiquer avec les applications *Microsoft Windows* tels que *Microsoft Excel*. Il est ainsi possible de créer une feuille *Excel* depuis un programme *Python*, de l'imprimer, de la sauvegarder sans avoir à ouvrir *Excel*. Un moteur de recherche permettra de trouver de nombreux petits exemples utilisant ce module [61].

`reportlab` [62] : ce module permet de manipuler les fichiers PDF, autant pour lire les informations qu'il contient que pour les modifier.

### 1.6.7 Accélération de *Python*

*Python* est un langage interprété et pour cette raison, il est lent ce qui ne veut pas dire qu'il n'existe pas des solutions pour contourner ce problème.

`psyco` : ce module permet de multiplier la vitesse d'exécution d'un programme par trois ou quatre. Il suffit d'ajouter au programme les instructions suivantes :

```
import psyco
psyco.full ()
```

56. http://www.pygtk.org/
57. http://www.pygame.org/
58. http://pyopengl.sourceforge.net/
59. http://sourceforge.net/projects/directpython/
60. http://python.net/crew/mhammond/win32/Downloads.html
61. comme http://wikipython.flibuste.net/CodeWindows#ExempleavecExcel
62. http://www.reportlab.com/software/opensource/rl-toolkit/

Dans les faits, le programme s'exécute plus vite mais pas uniformément plus vite : les calculs sont le plus favorisés. Toutefois, même avec ce module, le langage *Python* reste très lent par rapport à un langage compilé comme le *C++*. L'outil *Pyrex*[63] propose une solution en permettant l'insertion de code *C++* au sein d'un programme *Python*. *Pyrex* n'est pas un module, c'est un interpréteur *Python* modifié pour prendre en compte ces insertions *C++*. Il est essentiel de connaître la syntaxe du langage *C++*. *Pyrex* propose des temps d'exécution tout à fait acceptable par rapport à ceux du *C++* et il accroît fortement la vitesse d'écriture des programmes. Son installation n'est pas toujours évidente car il faut le relier au compilateur *C++ GCC*[64]. Une approche semblable est développée par l'extension `Weave`[65] du module `scipy`[66]. *Pyrex* et *Weave* offre des performances bien supérieures à *Python* seul, proches de celle du langage *C*. *Weave* conviendra sans doute mieux à ceux qui ont déjà programmé en *C*.

L'accélération de l'exécution d'un programme *Python* suscite l'intérêt de nombreux informaticiens, c'est aussi l'objectif de l'outil *PyPy*[67] qui réécrit automatiquement certaines partie du code en langage *C* et l'intègre au programme existant.

## 1.7 Outils connexes

Ce paragraphe sort du cadre de ce livre. Néanmoins, on ne conçoit pas un programme d'envergure sans outil autre que *Python*. Lorsqu'on travaille à plusieurs, il faut pouvoir conserver les différentes versions d'un programme, être capable de fusionner des modifications effectuées par deux personnes, échanger de l'information. Ce sont ces tâches auxquelles les applications qui suivent tentent d'apporter des réponses. La liste des outils présentés dans ce livre est loin d'être exhaustive. Il existe de nombreuses applications *open source* qui améliorent l'usage qu'on peut avoir des ordinateurs. De nombreux internautes échangent autour de problèmes rencontrées, comparent différents logiciels. On fait parfoirs quelques trouvailles en y jetant un coup d'œil régulièrement.

### 1.7.1 Outils quotidiens

**SVN**[68] : c'est un logiciel de suivi de source, il permet de conserver différentes versions d'une arborescence de fichiers, de fusionner des modifications faites par plusieurs personnes. C'est très pratique pour travailler à plusieurs. Une interface graphique est également disponible : **TortoiseSVN**[69].

**MoinMoin**[70] : les wikis sont devenus des outils presque incontournables qui permettent de conserver tout type d'information. *Wikipedia* est le wiki le plus célèbre.

63. http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/
64. http://gcc.gnu.org/, ce compilateur présent sur toute machine *Linux* est porté également sur *Microsoft Windows* (voir le site http://www.mingw.org/).
65. http://www.scipy.org/Weave
66. http://www.scipy.org/
67. http://codespeak.net/pypy/dist/pypy/doc/home.html
68. http://subversion.tigris.org/
69. http://tortoisesvn.tigris.org/
70. http://moinmo.in/

Ces outils sont très utiles pour partager l'information au sein d'une entreprise, pour archiver les rapports de réunions par exemple. Ils intègrent souvent un moteur de recherche.

### 1.7.2 Outils occasionnels

**doxygen** [71] : associé au module `doxypy` [72], cet outil offre une alternative plus riche que *epydoc*, il peut utiliser également *Graphviz* [73] pour produire des graphes.

**Sphinx** [74] : c'est l'outil qui génère l'aide associée au langage *Python* telle qu'elle apparaît à l'adresse http://docs.python.org/. Il crée des pages HTML et des documents au format PDF pour tout projet *Python*.

**HTML Help Workshop** [75] : cet outil permet de convertir une documentation au format HTML en un format CHM propre à *Microsoft Windows*. L'outil *doxygen* propose une option qui facilite cette conversion via ce logiciel.

**InnoSetup** [76] : cette application permet de construire un installateur qui s'exécute automatiquement. Lorsque vous souhaitez transmettre votre travail sous *Python* à quelqu'un d'autre, celui-ci doit récupérer le langage, un éditeur, les fichiers *Python*, d'éventuelles données. L'installation permettra de regrouper l'ensemble de ces opérations sans que l'utilisateur final n'ait à se soucier de quoi que ce soit. Cette application ne fonctionne que sous *Microsoft Windows*.

### 1.7.3 Outils pour des besoins spécifiques

Le langage *Python* s'interface avec de nombreuses applications, les outils qui suivent peuvent aider des programmeurs confirmés à automatiser certaines tâches fastidieuses.

**BuildBot** [77] : cet utilitaire est écrit en *Python*, lorsqu'il est couplé avec *SVN*, il facilite la conception d'applications informatiques à plusieurs. A chaque mise à jour du code source, *BuildBot* recompile les sources, passe les tests de non-régression [78], avertit le ou les programmeurs concernés par des erreurs éventuellement introduites lors d'une mise à jour. Il permet de suivre en temps réel la stabilité d'une application. Le projet *PyBots* [79] regroupe plusieurs outils d'automatisation intéressants.

**MySQL** [80] : c'est un serveur de base de données. Les bases de données sont deve-

71. http://www.stack.nl/~dimitri/doxygen/
72. http://code.foosel.org/doxypy
73. http://www.graphviz.org/
74. http://sphinx.pocoo.org/
75. http://www.microsoft.com/downloads/ puis saisir *HTML Help Workshop* dans la barre de recherche
76. http://www.innosetup.com/
77. http://buildbot.net/trac
78. Les tests de non-régression ont pour objectif de limiter le nombre d'erreurs dans un programme. Pour une fonction retournant le résultat d'un calcul scientifique, un test de non-régression consiste à vérifier que ce calcul, pour des données bien précises, aboutit toujours au même résultat malgré les modifications apportées au programme. La détection d'une erreur débouche souvent sur un test de non-régression afin de s'assurer qu'elle ne se reproduira pas.
79. http://www.pybots.org/
80. http://www.mysql.fr/

nus incontournables, elles permettent de stocker et de consulter un grand nombre d'informations de façon beaucoup plus efficace que des fichiers quels qu'ils soient. Le langage *Python* permet facilement de dialoguer directement avec les logiciels de bases de données les plus connus dont *MySQL*. Le module `MySQLdb`[81] permet de s'y connecter depuis *Python*.

**SQLite**[82] : cette application traite également les bases de données mais sans le côté serveur. Les bases de données sont stockées sous forme de fichiers. Cet outil est intégré à la version 2.5 du langage *Python*. L'application *SQLiteSpy*[83] permet sous *Microsoft Windows* de visualiser le contenu de ces fichiers et de lancer des requêtes *SQL*.

**Remote Python Call (RPyC)**[84] : ce module facilite l'écriture de programmes *Python* qui doivent s'exécuter sur plusieurs machines avec tous les problèmes que cela comporte : synchronisation, partage d'informations, ...

**Mercurial**[85] : c'est un autre logiciel de suivi de source écrit en *Python*, son mode de fonctionnement - décentralisé - est différent de celui de *SVN*. Avec *Mercurial*, il n'y a plus une référence des fichiers sources d'un programme, *Mercurial* est capable d'assembler des versions d'où qu'elles viennent.

**Bazaar**[86] : encore un logiciel de suivi de source créé plus récemment qui paraît plus convivial.

### 1.7.4 Outils Internet

**django**[87] : cet outil facilite la conception de sites Internet qui ne sont pas réduits à de simples pages HTML mais incluent des parties écrites en *Python*. Il propose des exemples de pages, des accès à des bases de données. Il est souvent utilisé en combinaison avec l'outil suivant.

**CherryPy**[88] : il permet de réaliser des applications Internet. Il est suffisamment robuste pour être utilisé par de nombreux sites Internet. Il peut paraître fastidieux de s'y plonger mais ce genre d'outil offre un réel gain de temps lors de la conception de sites qui incluent de nombreux échanges avec les utilisateurs.

**TurboGears**[89] : cet outil en regroupe plusieurs dont *CherryPy*, c'est un environement complet pour créer des sites Internet.

**Zope**[90] : c'est encore un outil pour concevoir des sites Internet, il est plus vieux et beaucoup plus riche. Il est également plus complexe à appréhender. Il est plutôt destiné au développement de sites conséquents.

81. http://sourceforge.net/projects/mysql-python
82. http://www.sqlite.org/
83. http://www.yunqa.de/delphi/doku.php/products/sqlitespy/index
84. http://rpyc.wikispaces.com/
85. http://www.selenic.com/mercurial/wiki/
86. http://bazaar.canonical.com/
87. http://www.django-fr.org/
88. http://www.cherrypy.org/
89. http://turbogears.org/
90. http://www.zope.org/

# Chapitre 2

# Types et variables du langage *Python*

## 2.1 Variables

Il est impossible d'écrire un programme sans utiliser de variable. Ce terme désigne le fait d'attribuer un nom ou identificateur à des informations : en les nommant, on peut manipuler ces informations beaucoup plus facilement. L'autre avantage est de pouvoir écrire des programmes valables pour des valeurs qui varient : on peut changer la valeur des variables, le programme s'exécutera toujours de la même manière et fera les mêmes types de calculs quelles que soient les valeurs manipulées. Les variables jouent un rôle semblable aux inconnues dans une équation mathématique.

L'ordinateur ne sait pas faire l'addition de plus de deux nombres mais cela suffit à calculer la somme de $n$ premiers nombres entiers. Pour cela, il est nécessaire de créer une variable intermédiaire qu'on appellera par exemple `somme` de manière à conserver le résultat des sommes intermédiaires.

```
somme = 0                   # initialisation : la somme est nulle
for i in range(1,n) :       # pour tous les indices de 1 à n exclu
    somme = somme + i       # on ajoute le i ème élément à somme
```

**Définition 2.1 : variable**
Une variable est caractérisée par :

*un identificateur :* il peut contenir des lettres, des chiffres, des blancs soulignés mais il ne peut commencer par un chiffre. Minuscules et majuscules sont différenciées. Il est aussi unique.

*un type :* c'est une information sur le contenu de la variable qui indique à l'interpréteur *Python* la manière de manipuler cette information.

Comme le typage est dynamique en *Python*, le type n'est pas précisé explicitement, il est implicitement liée à l'information manipulée. Par exemple, en écrivant, `x = 3.4`, on ne précise pas le type de la variable `x` mais il est implicite car `x` reçoit une valeur réelle : `x` est de type réel ou `float` en *Python*. Pour leur première initialisation, une variable reçoit dans la plupart des cas une constante :

**Définition 2.2 : constante**
Les constantes sont le contraire des variables, ce sont toutes les valeurs numériques, chaînes de caractères, ..., tout ce qui n'est pas désigné par un nom. Les constantes possèdent un type mais pas d'identificateur.

Le langage *Python* possède un certain nombre de types de variables déjà définis ou types fondamentaux à partir desquels il sera possible de définir ses propres types (voir chapitre 4). Au contraire de langages tels que le *C*, il n'est pas besoin de déclarer une variable pour signifier qu'elle existe, il suffit de lui affecter une valeur. Le type de la variable sera défini par le type de la constante qui lui est affectée. Le type d'une variable peut changer, il correspond toujours au type de la dernière affectation.

```
x   = 3.5      # création d'une variable nombre réel appelée x initialisée à 3.5
               # 3.5 est un réel, la variable est de type "float"
sc = "chaîne"  # création d'une variable chaîne de caractères appelée str
               # initialisée à "chaîne", sc est de type "str"
```

**Remarque 2.3 : commentaires**
Pour tous les exemples qui suivront, le symbole # apparaîtra à maintes reprises. Il marque le début d'un commentaire que la fin de la ligne termine. Autrement dit, un commentaire est une information aidant à la compréhension du programme mais n'en faisant pas partie comme dans l'exemple qui suit.

```
x = 3          # affectation de la valeur entière 3 à la variable x
y = 3.0        # affectation de la valeur réelle 3.0 à la variable y
```

**Remarque 2.4 : instruction sur plusieurs lignes**
Le *Python* impose une instruction par ligne. Il n'est pas possible d'utiliser deux lignes pour écrire une affectation à moins de conclure chaque ligne qui n'est pas la dernière par le symbole \. L'exemple suivant est impossible.

```
x =
    5.5
```

Il devrait être rédigé comme suit :

```
x =  \
    5.5
```

Avec ce symbole, les longues instructions peuvent être écrites sur plusieurs lignes de manière plus lisibles, de sorte qu'elles apparaissent en entier à l'écran. Si le dernier caractère est une virgule, il est implicite.

Les paragraphes suivant énumèrent les types incontournables en *Python*. Ils sont classés le plus souvent en deux catégories : types *immuables* ou *modifiables*. Tous les types du langage *Python* sont également des objets, c'est pourquoi on retrouve dans ce chapitre certaines formes d'écriture similaires à celles présentées plus tard dans le chapitre concernant les classes (chapitre 4).

## 2.2 Types immuables (ou *immutable*)

> **Définition 2.5 : type immuable (ou immutable)**
> Une variable de type immuable ne peut être modifiée. Une opération sur une variable de ce type entraîne nécessairement la création d'une autre variable du même type, même si cette dernière est temporaire.

Autrement dit, la simple instruction `x+=3` qui consiste à ajouter à la variable `x` la valeur `3` crée une seconde variable dont la valeur est celle de `x` augmentée de `3` puis

à en recopier le contenu dans celui de la variable `x`. Les nombres sont des types immuables tout comme les chaînes de caractères et les `tuple` qui sont des tableaux d'objets. Il n'est pas possible de modifier une variable de ce type, il faut en recréer une autre du même type qui intègrera la modification.

### 2.2.1 Type "rien"

*Python* propose un type `None` pour signifier qu'une variable ne contient rien. La variable est de type `None` et est égale à `None`.

```
s = None
print s    # affiche None
```

Certaines fonctions [1] utilisent cette convention lorsqu'il leur est impossible de retourner un résultat. Ce n'est pas la seule option pour gérer cette impossibilité : il est possible de générer une exception [2], de retourner une valeur par défaut ou encore de retourner `None`. Il n'y a pas de choix meilleur, il suffit juste de préciser la convention choisie.

### 2.2.2 Nombres réels et entiers

Il existe deux types de nombres en *Python*, les nombres réels (`float`) et les nombres entiers (`int`). L'instruction `x=3` crée une variable de type `int` initialisée à `3` tandis que `y=3.0` crée une variable de type `float` initialisée à `3.0`. Le programme suivant permet de vérifier cela en affichant pour les variables `x` et `y`, leurs valeurs et leurs types respectifs grâce à la fonction `type`.

```
x = 3
y = 3.0
print "x =", x, type(x)
print "y =", y, type(y)
```

```
x = 3 <type 'int'>
y = 3.0 <type 'float'>
```

La liste des opérateurs qui s'appliquent aux nombres réels et entiers suit ce paragraphe. Les trois premiers résultats s'expliquent en utilisant la représentation en base deux. `8 « 1` s'écrit en base deux `100 « 1 = 1000`, ce qui vaut 16 en base décimale : les bits sont décalés vers la droite ce qui équivaut à multiplier par deux. De même, `7&2` s'écrit `1011&10 = 10`, qui vaut 2 en base décimale. Les opérateurs <<, >>, |, & sont des opérateurs bit à bit, ils se comprennent à partir de la représentation binaire des nombres entiers.

1. Les fonctions sont définies au paragraphe 3.4, plus simplement, ce sont des mini-programmes : elles permettent de découper un programme long en tâches plus petites. On les distingue des variables car leur nom est suivi d'une liste de constantes ou variables comprises entre parenthèses et séparées par une virgule.
2. voir chapitre 5

| opérateur | signification | exemple |
|---|---|---|
| << >> | décalage à gauche, à droite | `x = 8 << 1` (résultat = 16) |
| \| | opérateur logique *ou* bit à bit | `x = 8\|1` (résultat = 9) |
| & | opérateur logique *et* bit à bit | `x = 11&2` (résultat = 2) |
| + - | addition, soustraction | `x = y + z` |
| += -= | addition ou soustraction puis affectation | `x += 3` |
| * / | multiplication, division | `x = y * z` |
| // | division entière, le résultat est de type réel si l'un des nombres est réel | `x = y//3` |
| % | reste d'une division entière (modulo) | `x = y%3` |
| *= /= | multiplication ou division puis affectation | `x *= 3` |
| ** | puissance (entière ou non, racine carrée = ** 0.5) | `x = y**3` |

Les fonctions `int` et `float` permettent de convertir un nombre quelconque ou une chaîne de caractères respectivement en un entier (arrondi) et en un nombre réel.

```
x = int (3.5)
y = float (3)
z = int ("3")
print "x:", type(x), "   y:", type(y), "   z:", type(z)
# affiche   x: <type 'int'>    y: <type 'float'>    z: <type 'int'>
```

**Remarque 2.6 : conversion en nombre entier**
Il peut arriver que la conversion en un nombre entier ne soit pas directe. Dans l'exemple qui suit, on cherche à convertir une chaîne de caractères (voir paragraphe 2.2.4) en entier mais cette chaîne représente un réel. Il faut d'abord la convertir en réel puis en entier, c'est à ce moment que l'arrondi sera effectué.

```
i = int ("3.5")          # provoque une erreur
i = int (float ("3.5"))  # fonctionne
```

Les opérateurs listés par le tableau ci-dessus ont des priorités différentes, triés par ordre croissant [3]. Toutefois, il est conseillé d'avoir recours aux parenthèses pour enlever les doutes : $3 * 2 * *4 = 3 * (2 * *4)$.

**Remarque 2.7 : division entière**
Il existe un cas pour lequel cet opérateur cache un sens implicite : lorsque la division opère sur deux nombres entiers ainsi que le montre l'exemple suivant.

```
x = 11
y = 2
z = x / y      # le résultat est 5 et non 5.5 car la division est entière
```

Pour obtenir le résultat juste incluant la partie décimale, il faut convertir les entiers en réels.

3. La page http://docs.python.org/reference/expressions.html#operator-summary est plus complète à ce sujet.

```
x = float (11)
y = float (2)
z = x / y      # le résultat est 5.5 car c'est une division entre deux réels
```

Pour éviter d'écrire le type `float`, on peut également écrire `11.0/2` de façon à spécifier explicitement que la valeur 11.0 est réelle et non entière. L'opérateur // permet d'effectuer une division entière lorsque les deux nombres à diviser sont réels, le résultat est un entier mais la variable est de type réel si l'un des nombres est de type réel.

### 2.2.3 Booléen

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : `True` ou `False`. Voici la liste des opérateurs qui s'appliquent aux booléens.

| **opérateur** | **signification** | **exemple** |
|---|---|---|
| `and` `or` | et, ou logique | `x = True or False` (résultat = True) |
| `not` | négation logique | `x = not x` |

```
x = 4 < 5
print x         # affiche True
print not x     # affiche False
```

Voici la liste des opérateurs de comparaisons qui retournent des booléens. Ceux-ci s'applique à tout type, aux entiers, réels, chaînes de caractères, t-uples... Une comparaison entre un entier et une chaîne de caractères est syntaxiquement correcte même si le résultat a peu d'intérêt.

| **opérateur** | **signification** | **exemple** |
|---|---|---|
| < > | inférieur, supérieur | `x = 5 < 5` |
| <= >= | inférieur ou égal, supérieur ou égal | `x = 5 <= 5` |
| == != | égal, différent | `x = 5 == 5` |

A l'instar des nombres réels, il est préférable d'utiliser les parenthèses pour éviter les problèmes de priorités d'opérateurs dans des expressions comme : `3 < x and x < 7`. Toutefois, pour cet exemple, *Python* accepte l'écriture résumée qui enchaîne des comparaisons : `3 < x and x < 7` est équivalent à `3 < x < 7`. Il existe deux autres mots-clés qui retournent un résultat de type booléen :

| **opérateur** | **signification** |
|---|---|
| is | test d'identification |
| in | test d'appartenance |

Ces deux opérateurs seront utilisés ultérieurement, `in` avec les listes, les dictionnaires, les boucles (paragraphe 3.3.2), `is` lors de l'étude des listes (paragraphe 2.3.2) et des classes (chapitre 4). Bien souvent, les booléens sont utilisés de manière implicite lors de tests (paragraphe 3.2.1) ce qui n'empêche pas de les déclarer explicitement.

```
x = True
y = False
```

### 2.2.4 Chaîne de caractères

#### 2.2.4.1 Création d'une chaîne de caractères

> **Définition 2.8 : chaîne de caractères**
> Le terme "chaîne de caractères" ou *string* en anglais signifie une suite finie de caractères, autrement dit, du texte.

Ce texte est compris entre deux guillemets ou deux apostrophes, ces deux symboles sont interchangeables. L'exemple suivant montre comment créer une chaîne de caractères. Il ne faut pas confondre la partie entre guillemets ou apostrophes, qui est une constante, de la variable qui la contient.

```
t = "string = texte"
print type (t), t
t = 'string = texte, initialisation avec apostrophes'
print type (t), t

t = "morceau 1" \
    "morceau 2"    # second morceau ajouté au premier par l'ajout du symbole \,
                   # il ne doit rien y avoir après le symbole \,
                   # pas d'espace ni de commentaire
print t

t = """première ligne
seconde ligne"""   # chaîne de caractères qui s'étend sur deux lignes
print t
```

Le résultat de ce petit programme est le suivant :

```
<type 'str'> string = texte
<type 'str'> string = texte, initialisation avec apostrophes
morceau 1morceau 2
première ligne
seconde ligne
```

La troisième chaîne de caractères créée lors de ce programme s'étend sur deux lignes. Il est parfois plus commode d'écrire du texte sur deux lignes plutôt que de le laisser caché par les limites de fenêtres d'affichage. *Python* offre la possibilité de couper le texte en deux chaînes de caractères recollées à l'aide du symbole \ à condition que ce symbole soit le dernier de la ligne sur laquelle il apparaît. De même, lorsque le texte contient plusieurs lignes, il suffit de les encadrer entre deux symboles """ ou ''' pour que l'interpréteur *Python* considère l'ensemble comme une chaîne de caractères et non comme une série d'instructions.

Par défaut, le *Python* ne permet pas l'insertion de caractères tels que les accents dans les chaînes de caractères, le paragraphe 1.5.2 (page 22) explique comment résoudre ce problème. De même, pour insérer un guillemet dans une chaîne de caractères encadrée elle-même par des guillemets, il faut le faire précéder du symbole \. La séquence \" est appelée un extra-caractère (voir table 2.1).

**Remarque 2.9 : préfixe "r", chaîne de caractères**
Il peut être fastidieux d'avoir à doubler tous les symboles \ d'un nom de fichier.

| | |
|---|---|
| \" | guillemet |
| \' | apostrophe |
| \n | passage à la ligne |
| \\ | insertion du symbole \ |
| \% | pourcentage, ce symbole est aussi un caractère spécial |
| \t | tabulation |
| \r | retour à la ligne, peu usité, il a surtout son importance lorsqu'on passe d'un système *Windows* à *Linux* car *Windows* l'ajoute automatiquement à tous ses fichiers textes |

**Table 2.1 :** *Liste des extra-caractères les plus couramment utilisés à l'intérieur d'une chaîne de caractères (voir page http://docs.python.org/reference/lexical_analysis.html#strings).*

Il est plus simple dans ce cas de préfixer la chaîne de caractères par `r` de façon à éviter que l'utilisation du symbole \ ne désigne un caractère spécial. Les deux lignes suivantes sont équivalentes :

```
s = "C:\\Users\\Dupre\\exemple.txt"
s = r"C:\Users\Dupre\exemple.txt"
```

Sans la lettre "r", tous les \ doivent être doublés, dans le cas contraire, *Python* peut avoir des effets indésirables selon le caractère qui suit ce symbole.

#### 2.2.4.2 Manipulation d'une chaîne de caractères

Une chaîne de caractères est semblable à un tableau et certains opérateurs qui s'appliquent aux tableaux s'appliquent également aux chaînes de caractères. Ceux-ci sont regroupés dans la table 2.2. La fonction `str` permet de convertir un nombre, un tableau, un objet (voir chapitre 4) en chaîne de caractères afin de pouvoir l'afficher. La fonction `len` retourne la longueur de la chaîne de caractères.

```
x = 5.567
s = str (x)
print type(s), s   # <type 'str'> 5.567
print len(s)       # affiche 5
```

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères.

> **syntaxe 2.10 : fonctions et chaînes de caractères**
>
> `res = s.fonction (...)`
>
> Où `s` est une chaîne de caractères, `fonction` est le nom de l'opération que l'on veut appliquer à `s`, `res` est le résultat de cette manipulation.

La table 2.3 (page 38) présente une liste non exhaustive des fonctions disponibles dont un exemple d'utilisation suit. Cette syntaxe `variable.fonction(arguments)` est celle des classes[4].

4. voir chapitre 4

| opérateur | signification | exemple |
|---|---|---|
| + | concaténation de chaînes de caractères | t = "abc" + "def" |
| += | concaténation puis affectation | t += "abc" |
| in, not in | une chaîne en contient-elle une autre ? | "ed" in "med" |
| * | répétition d'une chaîne de caractères | t = "abc" * 4 |
| [n] | obtention du n$^{ième}$ caractère, le premier caractère a pour indice 0 | t = "abc"<br>print t[0] # donne a |
| [i : j] | obtention des caractères compris entre les indices i et j − 1 inclus, le premier caractère a pour indice 0 | t = "abc"<br>print t[0 : 2] # donne ab |

**Table 2.2 :** *Opérations applicables aux chaînes de caractères.*

| | |
|---|---|
| count( sub[, st [, end]]) | Retourne le nombre d'occurrences de la chaîne de caractères sub, les paramètres par défaut st et end permettent de réduire la recherche entre les caractères d'indice st et end exclu. Par défaut, st est nul tandis que end correspond à la fin de la chaîne de caractères. |
| find( sub[, st [, end]]) | Recherche une chaîne de caractères sub, les paramètres par défaut st et end ont la même signification que ceux de la fonction count. Cette fonction retourne -1 si la recherche n'a pas abouti. |
| isalpha() | Retourne True si tous les caractères sont des lettres, False sinon. |
| isdigit() | Retourne True si tous les caractères sont des chiffres, False sinon. |
| replace( old, new[, co]) | Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne old par new. Si le paramètre optionnel co est renseigné, alors seules les co premières occurrences seront remplacées. |
| split( [sep[, mspl]]) | Découpe la chaîne de caractères en se servant de la chaîne sep comme délimiteur. Si le paramètre mspl est renseigné, au plus mspl coupures seront effectuées. |
| strip([s]) | Supprime les espaces au début et en fin de chaîne. Si le paramètre s est renseigné, la fonction strip supprime tous les caractères qui font partie de s au début et en fin de chaîne. |
| upper() | Remplace les minuscules par des majuscules. |
| lower() | Remplace les majuscules par des minuscules. |
| join(words) | Fait la somme d'un tableau de chaînes de caractères (une liste ou un T-uple). La chaîne de caractères sert de séparateur qui doit être ajouté entre chaque élément du tableau words. Un exemple de cette fonction figure en page 39. |

**Table 2.3 :** *Quelques fonctions s'appliquant aux chaînes de caractères, l'aide associée au langage Python fournira la liste complète (voir page http://docs.python.org/library/stdtypes.html). Certains des paramètres sont encadrés par des crochets, ceci signifie qu'ils sont facultatifs.*

```
st = "langage python"
st = st.upper ()              # mise en lettres majuscules
i  = st.find ("PYTHON")       # on cherche "PYTHON" dans st
print i                       # affiche 8
print st.count ("PYTHON")     # affiche 1
print st.count ("PYTHON", 9)  # affiche 0
```

L'exemple suivant permet de retourner une chaîne de caractères contenant plusieurs éléments séparés par ";". La chaîne "`un;deux;trois`" doit devenir "`trois;deux;un`". On utilise pour cela les fonctionnalités `split` et `join` (voir table 2.3). L'exemple utilise également la fonctionnalité `reverse` des listes qui seront décrites plus loin dans ce chapitre. Il faut simplement retenir qu'une liste est un tableau. `reverse` retourne le tableau.

```
s    = "un;deux;trois"
mots = s.split (";")        # mots est égal à ['un', 'deux', 'trois']
mots.reverse ()             # retourne la liste, mots devient égal à
                            #                 ['trois', 'deux', 'un']
s2 = ";".join (mots)        # concaténation des éléments de mots séparés par ";"
print s2                    # affiche trois;deux;un
```

#### 2.2.4.3 Formatage d'une chaîne de caractères

*Python* offre une manière plus concise de former une chaîne de caractères à l'aide de plusieurs types d'informations en évitant la conversion explicite de ces informations (fonction `str`) et leur concaténation. Il est particulièrement intéressant pour les nombres réels qu'il est possible d'écrire en imposant un nombre de décimales fixe. Le format est le suivant :

**syntaxe 2.11 : chaîne de caractères, formatage**

```
"....  %c1  ....  %c2 " % (v1,v2)
```

`c1` est un code choisi parmi ceux de la table 2.4 (page 40). Il indique le format dans lequel la variable `v1` devra être transcrite. Il en est de même pour le code `c2` associé à la variable `v2`. Les codes insérés dans la chaîne de caractères seront remplacés par les variables citées entre parenthèses après le symbole % suivant la fin de la chaîne de caractères. Il doit y avoir autant de codes que de variables, qui peuvent aussi être des constantes.

Voici concrètement l'utilisation de cette syntaxe :

```
x = 5.5
d = 7
s = "caractères"
res =   "un nombre réel %f et un entier %d, une chaîne de %s, \n" \
        "un réel d'abord converti en chaîne de caractères %s" % (x,d,s, str(x+4))
print res
res =   "un nombre réel " + str (x) + " et un entier " + str (d) + \
        ", une chaîne de " + s + \
        ",\n un réel d'abord converti en chaîne de caractères " + str(x+4)
print res
```

La seconde affectation de la variable `res` propose une solution équivalente à la première en utilisant l'opérateur de concaténation +. Les deux solutions sont équivalentes, tout dépend des préférences de celui qui écrit le programme.

```
un nombre réel 5.500000 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
un nombre réel 5.5 et un entier 7, une chaîne de caractères,
un réel d'abord converti en chaîne de caractères 9.5
```

La première option permet néanmoins un formatage plus précis des nombres réels en imposant par exemple un nombre défini de décimal. Le format est le suivant :

**syntaxe 2.12 : chaîne de caractères, formatage des nombres**

```
"%n.df" % x
```

où `n` est le nombre de chiffres total et `d` est le nombre de décimales, `f` désigne un format réel indiqué par la présence du symbole %

```
x = 0.123456789
print x             # affiche 0.123456789
print "%1.2f" % x   # affiche 0.12
print "%06.2f" % x  # affiche 000.12
```

Il existe d'autres formats regroupés dans la table 2.4. L'aide reste encore le meilleur réflexe car le langage *Python* est susceptible d'évoluer et d'ajouter de nouveaux formats.

| | |
|---|---|
| d | entier relatif |
| e | nombre réel au format exponentiel |
| f | nombre réel au format décimal |
| g | nombre réel, format décimal ou exponentiel si la puissance est trop grande ou trop petite |
| s | chaîne de caractères |

**Table 2.4 :** *Liste non exhaustive des codes utilisés pour formater des informations dans une chaîne de caractères (voir page http://docs.python.org/library/stdtypes.html).*

### 2.2.5 T-uple

**Définition 2.13 : T-uple**
Les T-uple sont un tableau d'objets qui peuvent être de tout type. Ils ne sont pas modifiables.

Un T-uple apparaît comme une liste d'objets comprise entre parenthèses et séparés par des virgules. Leur création reprend le même format :

```
x = (4,5)               # création d'un T-uple composé de 2 entiers
x = ("un",1,"deux",2)   # création d'un T-uple composé de 2 chaînes de caractères
```

```
                        # et de 2 entiers, l'ordre d'écriture est important
x = (3,)                # création d'un T-uple d'un élément, sans la virgule,
                        # le résultat est un entier
```

Ces objets sont des vecteurs d'objets. Il est possible d'effectuer les opérations regroupées dans la table 2.5. Etant donné que les chaînes de caractères sont également des tableaux, ces opérations reprennent en partie celles de la table 2.2.

| | |
|---|---|
| `x in s` | vrai si `x` est un des éléments de `s` |
| `x not in s` | réciproque de la ligne précédente |
| `s + t` | concaténation de `s` et `t` |
| `s * n` | concatène `n` copies de `s` les unes à la suite des autres |
| `s[i]` | retourne le $i^{\text{ème}}$ élément de `s` |
| `s[i : j]` | retourne un T-uple contenant une copie des éléments de `s` d'indices $i$ à $j$ exclu. |
| `s[i : j : k]` | retourne un T-uple contenant une copie des éléments de `s` dont les indices sont compris entre $i$ et $j$ exclu, ces indices sont espacés de $k$ : $i, i+k, i+2k, i+3k, ...$ |
| `len(s)` | nombre d'éléments de `s` |
| `min(s)` | plus petit élément de `s`, résultat difficile à prévoir lorsque les types des éléments sont différents |
| `max(s)` | plus grand élément de `s` |
| `sum(s)` | retourne la somme de tous les éléments |

**Table 2.5 :** *Opérations disponibles sur les T-uples, on suppose que* `s` *et* `t` *sont des T-uples,* `x` *est quant à lui quelconque.*

**Remarque 2.14 : T-uple, opérateur []**
Les T-uples ne sont pas modifiables, cela signifie qu'il est impossible de modifier un de leurs éléments. Par conséquent, la ligne d'affectation suivante n'est pas correcte :

```
a     = (4,5)
a [0] = 3      # déclenche une erreur d'exécution
```

Le message d'erreur suivant apparaît :

```
Traceback (most recent call last):
  File "<pyshell#78>", line 1, in -toplevel-
    a[0]=3
TypeError: object doesn't support item assignment
```

Pour changer cet élément, il est possible de s'y prendre de la manière suivante :

```
a = (4,5)
a = (3,) + a[1:2]  # crée un T-uple d'un élément concaténé
                   # avec la partie inchangée de a
```

### 2.2.6 Autres types

Il existe d'autres types comme le type `complex` permettant de représenter les nombres complexes. Ce type numérique suit les mêmes règles et fonctionne avec les mêmes

opérateurs (excepté les opérateurs de comparaisons) que ceux présentés au paragraphe 2.2.2 et décrivant les nombres.

```
print complex (1,1)   # affiche (1+1j)
```

Le langage *Python* offre la possibilité de créer ses propres types immuables [5] mais ils seront définis à partir des types immuables présentés jusqu'ici.

## 2.3 Types modifiables (ou *mutable*)

Les types modifiables sont des conteneurs (ou containers en anglais) : ils contiennent d'autres objets, que ce soit des nombres, des chaînes de caractères ou des objets de type modifiable. Plutôt que d'avoir dix variables pour désigner dix objets, on en n'utilise qu'une seule qui désigne ces dix objets.

> **Définition 2.15 : type modifiable (ou mutable)**
> Une variable de type modifiable peut être modifiée, elle conserve le même type et le même identificateur. C'est uniquement son contenu qui évolue.

### 2.3.1 Liste

#### 2.3.1.1 Définition et fonctions

> **Définition 2.16 : liste**
> Les listes sont des collections d'objets qui peuvent être de tout type. Elles sont modifiables.

Une liste apparaît comme une succession d'objets compris entre crochets et séparés par des virgules. Leur création reprend le même format :

```
x = [4,5]               # création d'une liste composée de deux entiers
x = ["un",1,"deux",2]   # création d'une liste composée de
                        # deux chaînes de caractères
                        # et de deux entiers, l'ordre d'écriture est important
x = [3,]                # création d'une liste d'un élément, sans la virgule,
                        # le résultat reste une liste
x = [ ]                 # crée une liste vide
x = list ()             # crée une liste vide
y = x [0]               # accède au premier élément
y = x [-1]              # accède au dernier élément
```

Ces objets sont des listes chaînées d'autres objets de type quelconque (immuable ou modifiable). Il est possible d'effectuer les opérations regroupées dans la table 2.6. Ces opérations reprennent celles des T-uples (voir table 2.5) et incluent d'autres fonctionnalités puisque les listes sont modifiables (voir table 2.6). Il est donc possible

5. voir le paragraphe 4.7

d'insérer, de supprimer des éléments, de les trier. La syntaxe des opérations sur les listes est similaire à celle des opérations qui s'appliquent sur les chaînes de caractères, elles sont présentées par la table 2.7.

| | |
|---|---|
| `x in l` | vrai si `x` est un des éléments de `l` |
| `x not in l` | réciproque de la ligne précédente |
| `l + t` | concaténation de `l` et `t` |
| `l * n` | concatène `n` copies de `l` les unes à la suite des autres |
| `l[i]` | retourne l'élément i$^{\text{ème}}$ élément de `l`, à la différence des T-uples, l'instruction `l[i] = "a"` est valide, elle remplace l'élément `i` par "`a`". Un indice négatif correspond à la position `len(l) + i`. |
| `l[i : j]` | retourne une liste contenant les éléments de `l` d'indices $i$ à $j$ exclu. Il est possible de remplacer cette sous-liste par une autre en utilisant l'affectation `l[i : j] = l2` où `l2` est une autre liste (ou un T-uple) de dimension différente ou égale. |
| `l[i : j : k]` | retourne une liste contenant les éléments de `l` dont les indices sont compris entre $i$ et $j$ exclu, ces indices sont espacés de $k : i, i+k, i+2k, i+3k, ...$ Ici encore, il est possible d'écrire l'affectation suivante : `l[i : j : k] = l2` mais `l2` doit être une liste (ou un T-uple) de même dimension que `l[i : j : k]`. |
| `len(l)` | nombre d'éléments de `l` |
| `min(l)` | plus petit élément de `l`, résultat difficile à prévoir lorsque les types des éléments sont différents |
| `max(l)` | plus grand élément de `l` |
| `sum(l)` | retourne la somme de tous les éléments |
| `del l[i : j]` | supprime les éléments d'indices entre `i` et `j` exclu. Cette instruction est équivalente à `l[i : j] = []`. |
| `list(x)` | convertit `x` en une liste quand cela est possible |

**Table 2.6 :** *Opérations disponibles sur les listes, identiques à celles des T-uples, on suppose que* `l` *et* `t` *sont des listes,* `i` *et* `j` *sont des entiers.* `x` *est quant à lui quelconque.*

#### 2.3.1.2 Exemples

L'exemple suivant montre une utilisation de la méthode `sort`.

```
x = [9,0,3,5,4,7,8]          # définition d'une liste
print x                      # affiche cette liste
x.sort ()                    # trie la liste par ordre croissant
print x                      # affiche la liste triée
```

Voici les deux listes affichées par cet exemple :

```
[9, 0, 3, 5, 4, 7, 8]
[0, 3, 4, 5, 7, 8, 9]
```

Pour classer les objets contenus par la liste mais selon un ordre différent, il faut définir une fonction qui détermine un ordre entre deux éléments de la liste. C'est la fonction `compare` de l'exemple suivant.

| | |
|---|---|
| `l.count(x)` | Retourne le nombre d'occurrences de l'élément `x`. Cette notation suit la syntaxe des classes développée au chapitre 4. `count` est une méthode de la classe `list`. |
| `l.index(x)` | Retourne l'indice de la première occurrence de l'élément `x` dans la liste `l`. Si celle-ci n'existe pas, une exception est déclenchée (voir le paragraphe 5 ou l'exemple page 45). |
| `l.append(x)` | Ajoute l'élément `x` à la fin de la liste `l`. Si `x` est une liste, cette fonction ajoute la liste `x` en tant qu'élément, au final, la liste `l` ne contiendra qu'un élément de plus. |
| `l.extend(k)` | Ajoute tous les éléments de la liste `k` à la liste `l`. La liste `l` aura autant d'éléments supplémentaires qu'il y en a dans la liste `k`. |
| `l.insert(i,x)` | Insère l'élément `x` à la position `i` dans la liste `l`. |
| `l.remove(x)` | Supprime la première occurrence de l'élément `x` dans la liste `l`. S'il n'y a aucune occurrence de `x`, cette méthode déclenche une exception. |
| `l.pop([i])` | Retourne l'élément `l[i]` et le supprime de la liste. Le paramètre `i` est facultatif, s'il n'est pas précisé, c'est le dernier élément qui est retourné puis supprimé de la liste. |
| `l.reverse(x)` | Retourne la liste, le premier et dernier élément échange leurs places, le second et l'avant dernier, et ainsi de suite. |
| `l.sort([f,rev])` | Cette fonction trie la liste par ordre croissant. Le paramètre `f` est facultatif, il permet de préciser la fonction de comparaison qui doit être utilisée lors du tri. Cette fonction prend comme paramètre deux éléments `x` et `y` de la liste et retourne les valeurs -1,0,1 selon que `x < y`, `x == y` ou `x > y` (voir paragraphe 3.4). Si `rev` est `True`, alors le tri est décroissant. |

**Table 2.7 :** *Opérations permettant de modifier une liste on suppose que* `l` *est une liste,* `x` *est quant à lui quelconque.*

```
def compare (x,y):           # crée une fonction
    if   x >  y : return -1  # qui retourne -1 si x<y,
    elif x == y : return 0   # 0 si x == y
    else        : return 1   # 1 si x < y

x.sort (compare)             # trie la liste x à l'aide de la fonction compare
                             # cela revient à la trier par ordre décroissant
print x
```

Le programme affiche cette fois-ci la ligne :

```
[9, 8, 7, 5, 4, 3, 0]
```

L'exemple suivant illustre un exemple dans lequel on essaye d'accéder à l'indice d'un élément qui n'existe pas dans la liste :

```
x = [9,0,3,5,0]
print x.index (1) # cherche la position de l'élément 1
```

Comme cet élément n'existe pas, on déclenche ce qu'on appelle une exception qui se traduit par l'affichage d'un message d'erreur. Le message indique le nom de l'exception générée (`ValueError`) ainsi qu'un message d'information permettant en règle générale de connaître l'événement qui en est la cause.

```
Traceback (most recent call last):
  File "c:/temp/temp", line 2, in -toplevel-
    print x.index(1)
ValueError: list.index(x): x not in list
```

Pour éviter cela, on choisit d'intercepter l'exception (voir paragraphe 5).

```
x = [9,0,3,5,0]
try:               print x.index(1)
except ValueError: print "1 n'est pas présent dans la liste x"
else:              print "trouvé"
```

Ce programme a pour résultat :

```
1 n'est pas présent dans la liste x
```

#### 2.3.1.3 Fonctions `range`, `xrange`

Les listes sont souvent utilisées dans des boucles ou notamment par l'intermédiaire de la fonction `range`. Cette fonction retourne une liste d'entiers.

**syntaxe 2.17 :** `range`

```
range (debut, fin [,marche])
```

Retourne une liste incluant tous les entiers compris entre `debut` et `fin` exclu. Si le paramètre facultatif `marche` est renseigné, la liste contient tous les entiers `n` compris `debut` et `fin` exclu et tels que `n` – `debut` soit un multiple de `marche`.

Exemple :

```
print range (0,10,2)       # affiche [0, 2, 4, 6, 8]
```

La fonction `xrange` est d'un usage équivalent à `range`. Elle permet de parcourir une liste d'entiers sans la créer vraiment. Elle est plus rapide.

```
print xrange (0,10,2)      # affiche xrange(0,10,2)
```

Cette fonction est souvent utilisée lors de boucles[6] pour parcourir tous les éléments d'un T-uple, d'une liste, d'un dictionnaire... Le programme suivant permet par exemple de calculer la somme de tous les entiers impairs compris entre 1 et 20 exclu.

```
s = 0
for n in range (1,20,2) :  # ce programme est équivalent à
    s += n                 # s = sum (range(1,20,2))
```

6. voir paragraphe 3.3.2

Le programme suivant permet d'afficher tous les éléments d'une liste.

```
x = ["un", 1, "deux", 2, "trois", 3]
for n in range (0, len(x)) :
    print "x [%d] = %s" % (n, x [n])

# le résultat est présenté à droite
```

```
x [0] = un
x [1] = 1
x [2] = deux
x [3] = 2
x [4] = trois
x [5] = 3
```

#### 2.3.1.4 Boucles et listes

Il est possible aussi de ne pas se servir des indices comme intermédiaires pour accéder aux éléments d'une liste quand il s'agit d'effectuer un même traitement pour tous les éléments de la liste `x`.

```
x = ["un", 1, "deux", 2]
for el in x :
    print "la liste inclut : ", el
```

L'instruction `for el in x :` se traduit littéralement par : *pour tous les éléments de la liste, faire...* Ce programme a pour résultat :

```
la liste inclut :  un
la liste inclut :  1
la liste inclut :  deux
la liste inclut :  2
```

Il existe également des notations abrégées lorsqu'on cherche à construire une liste à partir d'une autre. Le programme suivant construit la liste des entiers de 1 à 5 à partir du résultat retourné par la fonction `range`.

```
y = list ()
for i in range(0,5) : y.append (i+1)
print y                                 # affiche [1,2,3,4,5]
```

Le langage *Python* offre la possibilité de résumer cette écriture en une seule ligne. Cette syntaxe sera reprise au paragraphe 3.3.2.2.

```
y = [ i+1 for i in range (0,5)]
print y                                 # affiche [1,2,3,4,5]
```

Cette définition de liste peut également inclure des tests ou des boucles imbriquées.

```
y = [ i for i in range(0,5) if i % 2 == 0]   # sélection les éléments pairs
print y                                      # affiche [0,2,4]
z = [ i+j for i in range(0,5) \
          for j in range(0,5)]      # construit tous les nombres i+j possibles
print z                             # affiche [0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 2,
                                    # 3, 4, 5, 6, 3, 4, 5, 6, 7, 4, 5, 6, 7, 8]
```

#### 2.3.1.5 Collage de séquences, fonction `zip`

On suppose qu'on dispose de $n$ séquences d'éléments (T-uple, liste), toutes de longueur $l$. La fonction `zip` permet de construire une liste de T-uples qui est la juxtaposition de toutes ces séquences. Le $i^{\text{ième}}$ T-uple de la liste résultante contiendra tous les $i^{\text{ième}}$ éléments des séquences juxtaposées. Si les longueurs des séquences sont différentes, la liste résultante aura même taille que la plus courte des séquences.

```
a = (1,0,7,0,0,0)
b = [2,2,3,5,5,5]
c = [ "un", "deux", "trois", "quatre" ]
d = zip (a,b,c)
print d           # affiche [(1, 2, 'un'),    (0, 2, 'deux'),
                  #          (7, 3, 'trois'), (0, 5, 'quatre')]
```

#### 2.3.1.6 Concaténation de chaîne de caractères

Il arrive fréquemment de constuire une chaîne de caractères petits bouts par petits bouts comme le montre le premier exemple ci-dessous. Cette construction peut s'avérer très lente lorsque le résultat est long. Dans ce cas, il est nettement plus rapide d'ajouter chaque morceau dans une liste puis de les concaténer en une seule fois grâce à la méthode `join`.

```
s = ""
while <condition> : s += ...
```

```
s = []
while <condition> : s.append ( ... )
s = "".join (s)
```

### 2.3.2 Copie

A l'inverse des objets de type immuable, une affectation ne signifie pas une copie. Afin d'éviter certaines opérations superflues et parfois coûteuses en temps de traitement, on doit distinguer la variable de son contenu. Une variable désigne une liste avec un mot (ou identificateur), une affection permet de créer un second mot pour désigner la même liste. Il est alors équivalent de faire des opérations avec le premier mot ou le second comme le montre l'exemple suivant avec les listes `l` et `l2`.

```
l  = [4,5,6]
l2 = l
print l            # affiche [4,5,6]
print l2           # affiche [4,5,6]
l2 [1] = "modif"
print l            # affiche [4, 'modif', 6]
print l2           # affiche [4, 'modif', 6]
```

Dans cet exemple, il n'est pas utile de créer une seconde variable, dans le suivant, cela permet quelques raccourcis.

```
l      = [[0,1], [2,3]]
l1     = l [0]
l1 [0] = "modif" # ligne équivalente à : l [0][0] = "modif"
```

Par conséquent, lorsqu'on affecte une liste à une variable, celle-ci n'est pas recopiée, la liste reçoit seulement un nom de variable. L'affectation est en fait l'association d'un nom avec un objet (voir paragraphe 4.6). Pour copier une liste, il faut utiliser la fonction `copy` du module `copy`[7]. Ce point sera rappelé au paragraphe 4.6.3 (page 118).

```
import copy
l  = [4,5,6]
l2 = copy.copy(l)
print l            # affiche [4,5,6]
print l2           # affiche [4,5,6]
l2 [1] = "modif"
print l            # affiche [4,5,6]
print l2           # affiche [4, 'modif', 6]
```

L'opérateur == permet de savoir si deux listes sont égales même si l'une est une copie de l'autre. Le mot-clé `is` permet de vérifier si deux variables font référence à la même liste ou si l'une est une copie de l'autre comme le montre l'exemple suivant :

```
import copy
l  = [1,2,3]
l2 = copy.copy (l)
l3 = l

print l == l2  # affiche True
print l is l2  # affiche False
print l is l3  # affiche True
```

**Remarque 2.18 : fonction `copy` et `deepcopy`**
Le comportement de la fonction `copy` peut surprendre dans le cas où une liste contient d'autres listes. Pour être sûr que chaque élément d'une liste a été correctement recopiée, il faut utiliser la fonction `deepcopy`. La fonction est plus longue mais elle recopie toutes les listes que ce soit une liste incluse dans une liste elle-même incluse dans une autre liste elle-même incluse...

```
import copy
l  = [[1,2,3],[4,5,6]]
l2 = copy.copy (l)
l3 = copy.deepcopy (l)
l [0][0] = 1111
print l                # affiche [[1111, 2, 3], [4, 5, 6]]
print l2               # affiche [[1111, 2, 3], [4, 5, 6]]
print l3               # affiche [[1, 2, 3], [4, 5, 6]]
print l is l2          # affiche False
print l [0] is l2 [0]  # affiche True
print l [0] is l3 [0]  # affiche False
```

La fonction `deepcopy` est plus lente à exécuter car elle prend en compte les références récursives comme celles de l'exemple suivant où deux listes se contiennent l'une l'autre.

7. Le module `copy` est une extension interne. C'est une librairie de fonctions dont la fonction `copy`. Cette syntaxe sera vue au chapitre 6.

```
l     = [1,"a"]
ll    = [l,3]   # ll contient l
l [0] = ll      # l contient ll
print l         # affiche [[[...], 3], 'a']
print ll        # affiche [[[...], 'a'], 3]

import copy
z = copy.deepcopy (l)
print z         # affiche [[[...], 3], 'a']
```

### 2.3.3 Dictionnaire

Les dictionnaires sont des tableaux plus souples que les listes. Une liste référence les éléments en leur donnant une position : la liste associe à chaque élément une position entière comprise entre 0 et $n-1$ si $n$ est la longueur de la liste. Un dictionnaire permet d'associer à un élément autre chose qu'une position entière : ce peut être un entier, un réel, une chaîne de caractères, un T-uple contenant des objets immuables. D'une manière générale, un dictionnaire associe à une valeur ce qu'on appelle une clé de type immuable. Cette clé permettra de retrouver la valeur associée.

L'avantage principal des dictionnaires est la recherche optimisée des clés. Par exemple, on recense les noms des employés d'une entreprise dans une liste. On souhaite ensuite savoir si une personne ayant un nom précisé à l'avance appartient à cette liste. Il faudra alors parcourir la liste jusqu'à trouver ce nom ou parcourir toute la liste si jamais celui-ci ne s'y trouve pas[8]. Dans le cas d'un dictionnaire, cette recherche du nom sera beaucoup plus rapide à écrire et à exécuter.

#### 2.3.3.1 Définition et fonctions

> **Définition 2.19 : dictionnaire**
> Les dictionnaires sont des listes de couples. Chaque couple contient une clé et une valeur. Chaque valeur est indicée par sa clé. La valeur peut-être de tout type, la clé doit être de type immuable, ce ne peut donc être ni une liste, ni un dictionnaire. Chaque clé comme chaque valeur peut avoir un type différent des autres clés ou valeurs.

Un dictionnaire apparaît comme une succession de couples d'objets comprise entre accolades et séparés par des virgules. La clé et sa valeur sont séparées par le symbole :. Leur création reprend le même format :

```
x = { "cle1":"valeur1", "cle2":"valeur2" }
y = { }         # crée un dictionnaire vide
z = dict ()     # crée aussi un dictionnaire vide
```

Les indices ne sont plus entiers mais des chaînes de caractères pour cet exemple. Pour associer la valeur à la clé "cle1", il suffit d'écrire :

8. voir également le paragraphe 3.7.1, page 88

```
print x ["cle1"]
```

La plupart des fonctions disponibles pour les listes sont interdites pour les dictionnaires comme la concaténation ou l'opération de multiplication (∗). Il n'existe plus non plus d'indices entiers pour repérer les éléments, le seul repère est leur clé. La table 2.8 dresse la liste des opérations simples sur les dictionnaires tandis que la table 2.9 dresse la liste des méthodes plus complexes.

| | |
|---|---|
| `x in d` | vrai si `x` est une des clés de `d` |
| `x not in d` | réciproque de la ligne précédente |
| `d[i]` | retourne l'élément associé à la clé `i` |
| `len(d)` | nombre d'éléments de `d` |
| `min(d)` | plus petite clé |
| `max(d)` | plus grande clé |
| `del d[i]` | supprime l'élément associé à la clé `i` |
| `list(d)` | retourne une liste contenant toutes les clés du dictionnaire `d`. |
| `dict(x)` | convertit `x` en un dictionnaire si cela est possible, `d` est alors égal à `dict(d.items())` |

**Table 2.8 :** *Opérations disponibles sur les dictionnaires,* `d` *est un dictionnaire,* `x` *est quant à lui quelconque.*

| | |
|---|---|
| `d.copy()` | Retourne une copie de `d`. |
| `d.has_key(x)` | Retourne `True` si `x` est une clé de `d`. |
| `d.items()` | Retourne une liste contenant tous les couples (clé, valeur) inclus dans le dictionnaire. |
| `d.keys()` | Retourne une liste contenant toutes les clés du dictionnaire `d`. |
| `d.values()` | Retourne une liste contenant toutes les valeurs du dictionnaire `d`. |
| `d.iteritems()` | Retourne un itérateur sur les couples (clé, valeur). |
| `d.iterkeys()` | Retourne un itérateur sur les clés. |
| `d.itervalues()` | Retourne un itérateur sur les valeurs. |
| `d.get(k[,x])` | Retourne `d[k]`, si la clé `k` est manquante, alors la valeur `None` est retournée à moins que le paramètre optionnel `x` soit renseigné, auquel cas, ce sera ce paramètre qui sera retourné. |
| `d.clear()` | Supprime tous les éléments du dictionnaire. |
| `d.update(d2)` | Le dictionnaire `d` reçoit le contenu de `d2`. |
| `d.setdefault(k[,x])` | Définit `d[k]` si la clé `k` existe, sinon, lui affecte `x` à `d[k]`. |
| `d.popitem()` | Retourne un élément et le supprime du dictionnaire. |

**Table 2.9 :** *Méthodes associées aux dictionnaires,* `d`, `d2` *sont des dictionnaires,* `x` *est quant à lui quelconque.*

Contrairement à une liste, un dictionnaire ne peut être trié car sa structure interne est optimisée pour effectuer des recherches rapides parmi les éléments. On peut aussi se demander quel est l'intérêt de la méthode `popitem` qui retourne un élément puis le

supprime alors qu'il existe le mot-clé `del`. Cette méthode est simplement plus rapide car elle choisit à chaque fois l'élément le moins coûteux à supprimer, surtout lorsque le dictionnaire est volumineux.

Les itérateurs sont des objets qui permettent de parcourir rapidement un dictionnaire, ils seront décrits en détail au chapitre 4 sur les classes. Un exemple de leur utilisation est présenté dans le paragraphe suivant.

#### 2.3.3.2 Exemples

Il n'est pas possible de trier un dictionnaire. L'exemple suivant permet néanmoins d'afficher tous les éléments d'un dictionnaire selon un ordre croissant des clés. Ces exemples font appel aux paragraphes sur les boucles (voir chapitre 3).

```
d = { "un":1, "zéro":0, "deux":2, "trois":3, "quatre":4, "cinq":5, \
       "six":6, "sept":1, "huit":8, "neuf":9, "dix":10 }
key = d.keys ()
key.sort ()
for k in key:
    print k,d [k]
```

L'exemple suivant montre un exemple d'utilisation des itérateurs. Il s'agit de construire un dictionnaire inversé pour lequel les valeurs seront les clés et réciproquement.

```
d = { "un":1,   "zero":0, "deux":2, "trois":3, "quatre":4, "cinq":5, \
      "six":6,  "sept":1, "huit":8, "neuf":9,  "dix":10 }

dinv = { }                      # création d'un dictionnaire vide, on parcout
for key,value in d.items ()  :  # les éléments du dictionnaire comme si
                                # c'était une liste de 2-uple (clé,valeur)
    dinv [value] = key          # on retourne le dictionnaire

print dinv                      # affiche {0: 'zero', 1: 'un', 2: 'deux',
                                # 3: 'trois', 4: 'quatre', 5: 'cinq', 6: 'six',
                                # 8: 'huit', 9: 'neuf', 10: 'dix'}
```

Pour être plus efficace, on peut remplacer la ligne `for key, value in d.items()` : par : `for key, value in d.iteritems()` :. De cette manière, on parcourt les éléments du dictionnaire sans créer de liste intermédiaire. Il est équivalent d'utiliser l'une ou l'autre au sein d'une boucle même si le programme suivant montre une différence.

```
d = { "un":1, "zero":0, "deux":2, "trois":3, "quatre":4, "cinq":5, \
       "six":6, "sept":1, "huit":8, "neuf":9, "dix":10 }
print d.items ()
print d.iteritems ()
```

Il a pour résultat :

```
[('trois', 3), ('sept', 1), ('neuf', 9), ('six', 6), ('zero', 0),
 ('un', 1), ('dix', 10), ('deux', 2), ('huit', 8), ('quatre', 4),
                                                    ('cinq', 5)]

<dictionary-itemiterator object at 0x0115DC40>
```

**Remarque 2.20 : modification d'un dictionnaire dans une boucle**
D'une manière générale, il faut éviter d'ajouter ou de supprimer un élément dans une liste ou un dictionnaire qu'on est en train de parcourir au sein d'une boucle `for` ou `while`. Cela peut marcher mais cela peut aussi aboutir à des résultats imprévisibles surtout avec l'utilisation d'itérateurs (fonction `iteritems`, `itervalues`, `iterkeys`). Il est préférable de terminer le parcours de la liste ou du dictionnaire puis de faire les modifications désirées une fois la boucle terminée. Dans le meilleur des cas, l'erreur suivante survient :

```
  File "essai.py", line 6, in <module>
    for k in d :
RuntimeError: dictionary changed size during iteration
```

#### 2.3.3.3 Copie

A l'instar des listes (voir paragraphe 2.3.2), les dictionnaires sont des objets et une affectation n'est pas équivalente à une copie comme le montre le programme suivant.

```
d  = {4:4,5:5,6:6}
d2 = d
print d            # affiche {4: 4, 5: 5, 6: 6}
print d2           # affiche {4: 4, 5: 5, 6: 6}
d2 [5] = "modif"
print d            # affiche {4: 4, 5: 'modif', 6: 6}
print d2           # affiche {4: 4, 5: 'modif', 6: 6}
```

Lorsqu'on affecte une liste à une variable, celle-ci n'est pas recopiée, la liste reçoit seulement un nom de variable. L'affectation est en fait l'association d'un nom avec un objet (voir paragraphe 4.6). Pour copier une liste, il faut utiliser la fonction `copy` du module `copy`.

```
d  = {4:4,5:5,6:6}
import copy
d2 = copy.copy(l)
print d            # affiche {4: 4, 5: 5, 6: 6}
print d2           # affiche {4: 4, 5: 5, 6: 6}
d2 [5] = "modif"
print d            # affiche {4: 4, 5: 5, 6: 6}
print d2           # affiche {4: 4, 5: 'modif', 6: 6}
```

Le mot-clé `is` a la même signification pour les dictionnaires que pour les listes, l'exemple du paragraphe 2.3.2 (page 47) est aussi valable pour les dictionnaires. Il en est de même pour la remarque concernant la fonction `deepcopy`. Cette fonction recopie les listes et les dictionnaires.

#### 2.3.3.4 Clés de type modifiable

Ce paragraphe concerne davantage des utilisateurs avertis qui souhaitent malgré tout utiliser des clés de type modifiable. Dans l'exemple qui suit, la clé d'un dictionnaire est également un dictionnaire et cela provoque une erreur. Il en serait de même si la variable `k` utilisée comme clé était une liste.

```
k = { 1:1}
d = { }
d [k] = 0
```

```
Traceback (most recent call last):
  File "cledict.py", line 3, in <module>
    d [k] = 0
TypeError: dict objects are unhashable
```

Cela ne veut pas dire qu'il faille renoncer à utiliser un dictionnaire ou une liste comme clé. La fonction `id` permet d'obtenir un entier qui identifie de manière unique tout objet. Le code suivant est parfaitement correct.

```
k = { 1:1}
d = { }
d [id (k)] = 0
```

Toutefois, ce n'est pas parce que deux dictionnaires auront des contenus identiques que leurs identifiants retournés par la fonction `id` seront égaux. C'est ce qui explique l'erreur que provoque la dernière ligne du programme suivant.

```
k = {1:1}
d = { }
d [id (k)] = 0
b = k
print d [id(b)]  # affiche bien zéro
c = {1:1}
print d [id(c)]  # provoque une erreur car même si k et c ont des contenus égaux,
                 # ils sont distincts, la clé id(c) n'existe pas dans d
```

Il existe un cas où on peut se passer de la fonction `id` mais il inclut la notion de classe définie au chapitre 4. L'exemple suivant utilise directement l'instance `k` comme clé. En affichant le dictionnaire `d`, on vérifie que la clé est liée au résultat de l'instruction `id(k)` même si ce n'est pas la clé.

```
class A : pass

k = A ()
d = { }
d [k] = 0
print d                    # affiche {<__main__.A object at 0x0120DB90>: 0}
print id (k), hex(id(k))   # affiche 18930576, 0x120db90
print d [id(k)]            # provoque une erreur
```

La fonction `hex` convertit un entier en notation hexadécimale. Les nombres affichés changent à chaque exécution. Pour conclure, ce dernier exemple montre comment se passer de la fonction `id` dans le cas d'une clé de type dictionnaire.

```
class A (dict):
    def __hash__(self):
        return id(self)

k = A ()
k ["t"]= 4
d = { }
d [k] = 0
print d         # affiche {{'t': 4}: 0}
```

## 2.4 Extensions

### 2.4.1 Mot-clé `print`, `repr`, conversion en chaîne de caractères

Le mot-clé `print` est déjà apparu dans les exemples présentés ci-dessus, il permet d'afficher une ou plusieurs variables préalablement définies, séparées par des virgules. Les paragraphes qui suivent donnent quelques exemples d'utilisation. La fonction `print` permet d'afficher n'importe quelle variable ou objet à l'écran, cet affichage suppose la conversion de cette variable ou objet en une chaîne de caractères. Deux fonctions permettent d'effectuer cette étape sans toutefois afficher le résultat à l'écran.

La fonction `str` (voir paragraphe 2.2.4.2) permet de convertir toute variable en chaîne de caractères. Il existe cependant une autre fonction `repr`, qui effectue cette conversion. Dans ce cas, le résultat peut être interprété par la fonction `eval` (voir paragraphe 2.4.2) qui se charge de la conversion inverse. Pour les types simples comme ceux présentés dans ce chapitre, ces deux fonctions retournent des résultats identiques. Pour l'exemple, `x` désigne n'importe quelle variable.

```
x == eval (repr(x)) # est toujours vrai (True)
x == eval (str (x)) # n'est pas toujours vrai
```

### 2.4.2 Fonction `eval`

Comme le suggère le paragraphe précédent, la fonction `eval` permet d'évaluer une chaîne de caractères ou plutôt de l'interpréter comme si c'était une instruction en *Python*. Le petit exemple suivant permet de tester toutes les opérations de calcul possibles entre deux entiers.

```
x  = 32
y  = 9
op = "+ - * / % // & | and or << >>".split ()
for o in op :
    s = str (x) + " " + o + "  " + str (y)
    print s, " = ", eval (s)
```

Ceci aboutit au résultat suivant :

```
32 +  9  =  41
32 -  9  =  23
32 *  9  =  288
32 /  9  =  3
32 %  9  =  5
32 //  9  =  3
32 &  9  =  0
32 |  9  =  41
32 and  9  =  9
32 or  9  =  32
32 <<  9  =  16384
32 >>  9  =  0
```

Le programme va créer une chaîne de caractères pour chacune des opérations et celle-ci sera évaluée grâce à la fonction `eval` comme si c'était une expression numérique.

Il faut bien sûr que les variables que l'expression mentionne existent durant son évaluation.

### 2.4.3 Informations fournies par *Python*

Bien que les fonctions ne soient définies que plus tard (paragraphe 3.4), il peut être intéressant de mentionner la fonction `dir` qui retourne la liste de toutes les variables créées et accessibles à cet instant du programme. L'exemple suivant :

```
x = 3
print dir ()
```

Retourne le résultat suivant :

```
['__builtins__', '__doc__', '__file__', '__name__', 'x']
```

Certaines variables - des chaînes des caractères - existent déjà avant même la première instruction. Elles contiennent différentes informations concernant l'environnement dans lequel est exécuté le programme *Python* :

| | |
|---|---|
| `__builtins__` | Ce module contient tous les éléments présents dès le début d'un programme *Python*, il contient entre autres les types présentés dans ce chapitre et des fonctions simples comme `range`. |
| `__doc__` | C'est une chaîne commentant le fichier, c'est une chaîne de caractères insérée aux premières lignes du fichiers et souvent entourée des symboles """ (voir chapitre 6). |
| `__file__` | Contient le nom du fichier où est écrit ce programme. |
| `__name__` | Contient le nom du module. |

La fonction `dir` est également pratique pour afficher toutes les fonctions d'un module. L'instruction `dir(sys)` affiche la liste des fonctions du module `sys` (voir chapitre 6).

**Remarque 2.21 : fonction `dir`**
La fonction `dir` appelée sans argument donne la liste des fonctions et variables définies à cet endroit du programme. Ce résultat peut varier selon qu'on se trouver dans une fonction, une méthode de classe ou à l'extérieur du programme. L'instruction `dir([])` donne la liste des méthodes qui s'appliquent à une liste.

De la même manière, la fonction `type` retourne une information concernant le type d'une variable.

```
x = 3
print x, type(x)     # affiche 3 <type 'int'>
x = 3.5
print x, type(x)     # affiche 3.5 <type 'float'>
```

### 2.4.4 Affectations multiples

Il est possible d'effectuer en *Python* plusieurs affectations simultanément.

```
x = 5       # affecte 5 à x
y = 6       # affecte 6 à y
x,y = 5,6   # affecte en une seule instruction 5 à x et 6 à y
```

Cette particularité reviendra lorsque les fonctions seront décrites puisqu'il est possible qu'une fonction retourne plusieurs résultats comme la fonction `divmod` illustrée par le programme suivant.

```
x,y = divmod (17,5)
print x,y                          # affiche 3 2
print "17 / 5 = 5 * ", x, " + ",y  # affiche 17 / 5 = 5 *  3  +  2
```

Le langage *Python* offre la possibilité d'effectuer plusieurs affectations sur la même ligne. Dans l'exemple qui suit, le couple $(5, 5)$ est affecté à la variable `point`, puis le couple `x`, `y` reçoit les deux valeurs du T-uple `point`.

```
x,y = point = 5,5
```

### 2.4.5 Type `long`

Le type `long` permet d'écrire des entiers aussi grands que l'on veut. Le langage *Python* passe automatiquement du type `int` à `long` lorsque le nombre considéré devient trop grand. Ils se comportent de la même manière excepté que les opérations sur des types `long` sont plus longues en temps d'exécution [9].

```
i = int(2**28)
for k in range (0,4) :
    i *= int(2)
    print type(i),i
```

```
<type 'int'> 536870912
<type 'int'> 1073741824
<type 'long'> 2147483648
<type 'long'> 4294967296
```

### 2.4.6 Ensemble

Le langage *Python* définit également ce qu'on appelle un ensemble. Il est défini par les classes `set` de type modifiable et la classe `frozenset` de type immuable. Ils n'acceptent que des types identiques et offrent la plupart des opérations liées aux ensembles comme l'intersection, l'union. D'un usage moins fréquent, ils ne seront pas plus détaillés [10].

```
print set ( (1,2,3) ) & set ( (2,3,5) )
          # construit l'intersection qui est set([2, 3])
```

Ce chapitre a présenté les différents types de variables définis par le langage *Python* pour manipuler des données ainsi que les opérations possibles avec ces types de données. Le chapitre suivant va présenter les tests, les boucles et les fonctions qui permettent de réaliser la plupart des programmes informatiques.

9. La différence dépend des opérations effectuées.
10. La page http://docs.python.org/library/stdtypes.html#set décrit l'ensemble des fonctionnalités qui leur sont attachées.

# Chapitre 3

# Syntaxe du langage *Python* (boucles, tests, fonctions)

---

Avec les variables, les boucles et les fonctions, on connaît suffisamment d'éléments pour écrire des programmes. Le plus difficile n'est pas forcément de les comprendre mais plutôt d'arriver à découper un algorithme complexe en utilisant ces briques élémentaires. C'est l'objectif des chapitres centrés autour des exercices. Toutefois, même si ce chapitre présente les composants élémentaires du langage, l'aisance qu'on peut acquérir en programmation vient à la fois de la connaissance du langage mais aussi de la connaissance d'algorithmes standards comme celui du tri ou d'une recherche dichotomique. C'est cette connaissance tout autant que la maîtrise d'un langage de programmation qui constitue l'expérience en programmation.

## 3.1 Les trois concepts des algorithmes

Les algorithmes sont composés d'instructions, ce sont des opérations élémentaires que le processeur exécute selon trois schémas :

| | |
|---|---|
| la séquence | enchaînement des instructions les unes à la suite des autres : passage d'une instruction à la suivante |
| le saut | passage d'une instruction à une autre qui n'est pas forcément la suivante (c'est une rupture de séquence) |
| le test | choix entre deux instructions |

Le saut n'apparaît plus de manière explicite dans les langages évolués car il est une source fréquente d'erreurs. Il intervient dorénavant de manière implicite au travers des boucles qui combinent un saut et un test comme le montre l'exemple suivant :

**Version avec boucles :**

```
initialisation de la variable moy à 0
faire pour i allant de 1 à N
      moy reçoit moy + ni
moy reçoit moy / N
```

**Version équivalente avec sauts :**

```
ligne 1 : initialisation de la variable moy à 0
ligne 2 : initialisation de la variable i à 1
ligne 3 : moy reçoit moy + ni
ligne 4 : i reçoit i + 1
```

```
ligne 5 : si i est inférieur ou égal à N alors aller à la ligne 3
ligne 6 : moy reçoit moy / N
```

Tout programme peut se résumer à ces trois concepts. Chaque langage les met en place avec sa propre syntaxe et parfois quelques nuances mais il est souvent facile de les reconnaître même dans des langages inconnus. Le calcul d'une somme décrit plus haut et écrit en *Python* correspond à l'exemple suivant :

```
moy = 0
for i in range(1,N+1):    # de 1 à N+1 exclu --> de 1 à N inclus
    moy += n [i]
moy /= N
```

Le premier élément de cette syntaxe est constitué de ses mots-clés (`for` et `in`) (voir également table 3.1) et des symboles (=, `+=`, `/=`, [, ], (, ), :) (voir table 3.2) La fonction `iskeyword` du module `keyword` [1] permet de savoir si un mot-clé donné fait partie du langage *Python*.

```
import keyword
print keyword.iskeyword("for")     # affiche True
print keyword.iskeyword("until")   # affiche False
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| and | as | assert | break | class | continue | def | del |
| elif | else | except | exec | finally | for | from | global |
| if | import | in | is | lambda | not | or | pass |
| print | raise | return | try | while | with | yield | |

**Table 3.1 :** *Mots-clés du langage Python, voir également la page http://docs.python.org/reference/lexical_analysis.html#identifiers.*

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + | − | * | ** | / | // | % | << | >> | & | \| | ^ | \ |
| ∼ | < | > | <= | >= | == | != | <> | ( | ) | [ | ] | |
| { | } | " | , | : | . | ’ | = | ; | += | -= | *= | |
| /= | //= | %= | &= | \|= | ^= | >>= | <<= | **= | """ | ’’’ | # | |

**Table 3.2 :** *Symbole du langage Python, certains ont plusieurs usages comme : qui est utilisé à chaque test ou boucle et qui permet aussi de déterminer une plage d'indices dans un tableau. L'opérateur <> est identique à ! = mais est déclaré obsolète (deprecated) : il est susceptible de disparaître dans une prochaine version de Python.*

**Remarque 3.1 : syntaxe et espaces**
Les espaces entre les mots-clés et les symboles ont peu d'importance, il peut n'y en avoir aucun comme dix. Les espaces servent à séparer un mot-clé, un nombre d'une variable.

1. Les modules seront décrits au chapitre 6. Dans ce cas-ci, un module est une librairie de fonctionnalités. La syntaxe `keyword.iskeyword` permet d'exécuter la fonctionnalité `iskeyword` du module `keyword`.

## 3.2 Tests

### 3.2.1 Définition et syntaxe

**Définition 3.2 : test**
Les tests permettent d'exécuter des instructions différentes selon la valeur d'une condition logique.

**syntaxe 3.3 : test**

```
if condition1 :
   instruction1
   instruction2
   ...
else :
   instruction3
   instruction4
   ...
```

La clause `else` est facultative. Lorsque la condition `condition1` est fausse et qu'il n'y a aucune instruction à exécuter dans ce cas, la clause `else` est inutile. La syntaxe du test devient :

```
if condition1 :
   instruction1
   instruction2
   ...
```

S'il est nécessaire d'enchaîner plusieurs tests d'affilée, il est possible de condenser l'écriture avec le mot-clé `elif` :

```
if condition1 :
   instruction1
   instruction2
   ...
elif condition2 :
   instruction3
   instruction4
   ...
elif condition3 :
   instruction5
   instruction6
   ...
else :
   instruction7
   instruction8
   ...
```

**Remarque 3.4 : indentation**
Le décalage des instructions par rapport aux lignes contenant les mots-clés `if`, `elif`, `else` est très important : il fait partie de la syntaxe du langage et s'appelle l'*indentation* (voir paragraphe 3.5). Celle-ci permet de grouper les instructions ensemble. Le programme suivant est syntaxiquement correct même si le résultat n'est pas celui désiré.

```
x = 1
if x > 0 :
   signe = 1
   print "le nombre est positif"
else :
   signe = -1
print "le nombre est négatif"  # ligne mal indentée
print "signe = ", signe
```

Une ligne est mal indentée (`print "le nombre est négatif"`). Elle ne devrait être exécutée que si la condition `x > 0` n'est pas vérifiée. Le fait qu'elle soit alignée avec les premières instructions du programme fait que son exécution n'a plus rien à voir avec cette condition. La programme répond de manière erronée :

```
le nombre est positif
le nombre est négatif
signe = 1
```

Dans certains cas, l'interpréteur *Python* ne sait pas à quel bloc attacher une instruction, c'est le cas de l'exemple suivant, la même ligne a été décalée de deux espaces, ce qui est différent de la ligne qui précède et de la ligne qui suit.

```
x = 1
if x > 0 :
    signe = 1
    print "le nombre est positif"
else :
    signe = -1
  print "le nombre est négatif"     # ligne mal indentée
print "signe = ", signe
```

L'interpréteur retourne l'erreur suivante :

```
  File "test.py", line 7
    print "le nombre est négatif"
                                 ^
IndentationError: unindent does not match any outer indentation level
```

### 3.2.2 Comparaisons possibles

Les comparaisons possibles entre deux entités sont avant tout numériques mais ces opérateurs peuvent être définis pour tout type (voir chapitre 4), notamment sur les chaînes de caractères pour lesquelles les opérateurs de comparaison transcrivent l'ordre alphabétique.

| | |
|---|---|
| < , > | inférieur, supérieur |
| <= , >= | inférieur ou égal, supérieur ou égal |
| == , ! = | égal, différent |
| `is` , `not is` | `x is y` vérifie que `x` et `y` sont égaux, `not is`, différents, l'opérateur `is` est différent de l'opérateur ==, il est expliqué au paragraphe 2.3.2 (page 47). |
| `in`, `not in` | appartient, n'appartient pas |

### 3.2.3 Opérateurs logiques

Il existe trois opérateurs logiques qui combinent entre eux les conditions.

| | | | | | |
|---|---|---|---|---|---|
| `not` | négation | `and` | et logique | `or` | ou logique |

**Remarque 3.5 : priorité des opérations**
La priorité des opérations numériques est identique à celle rencontrée en mathématiques. L'opérateur puissance vient en premier, la multiplication/division ensuite puis l'addition/soustraction. Ces opérations sont prioritaires sur les opérateurs de comparaisons (>, <, ==, ...) qui sont eux-mêmes sur les opérateurs logiques `not`, `and`, `or`. Il est tout de même conseillé d'ajouter des parenthèses en cas de doute.

### 3.2.4 Ecriture condensée

Il existe deux écritures condensées de tests. La première consiste à écrire un test et l'unique instruction qui en dépend sur une seule ligne.

```
if condition :
    instruction1
else :
    instruction2
```

Ce code peut tenir en deux lignes :

```
if condition : instruction1
else : instruction2
```

Le second cas d'écriture condensée concerne les comparaisons enchaînées. Le test `if` $3 <$ `x and x` $< 5$ : `instruction` peut être condensé par `if` $3 <$ `x` $< 5$ : `instruction`. Il est ainsi possible de juxtaposer autant de comparaisons que nécessaire : `if` $3 <$ `x` $<$ `y` $< 5$ : `instruction`.

Le mot-clé `in` permet également de condenser certains tests lorsque la variable à tester est entière. `if x` $==$ `1 or x` $==$ `6 or x` $==$ `50` : peut être résumé simplement par `if x in` $(1, 6, 50)$ :.

### 3.2.5 Exemple

L'exemple suivant associe à la variable `signe` le signe de la variable `x`.

```
x = -5
if x < 0 :
   signe = -1
elif x == 0 :
   signe = 0
else :
   signe = 1
```

Son écriture condensée lorsqu'il n'y a qu'une instruction à exécuter :

```
x = -5
if x < 0 : signe = -1
elif x == 0 : signe = 0
else : signe = 1
```

Le programme suivant saisit une ligne au clavier et dit si c'est "oui" ou "non" qui a été saisi.

```
s = raw_input ("dites oui : ")    # voir remarque suivante
if s == "oui" or s [0:1] == "o" or s [0:1] == "O" or s == "1" :
       print "oui"
else : print "non"
```

**Remarque 3.6 : fonction `raw_input`**
La fonction `raw_input` invite l'utilisateur d'un programme à saisir une réponse lors de l'exécution du programme. Tant que la touche entrée n'a pas été pressée, l'exécution du programme ne peut continuer. Cette fonction est en réalité peu utilisée. Les interfaces graphiques sont faciles d'accès en *Python*, on préfère donc saisir une réponse via une fenêtre plutôt qu'en ligne de commande. L'exemple suivant montre comment remplacer cette fonction à l'aide d'une fenêtre graphique, le résultat apparaît dans la figure 3.1. On peut améliorer la fonction `question` en précisant une valeur par défaut par exemple, le chapitre 8 sera plus explicite à ce sujet.

```
import Tkinter
def question (legende) :
    reponse = [""]
    root = Tkinter.Tk ()
    root.title ("pseudo raw_input")
    Tkinter.Label (text = legende).pack (side = Tkinter.LEFT)
    s = Tkinter.Entry (text= "def", width=80)
    s.pack (side = Tkinter.LEFT)
    def rget () :
        reponse [0] = s.get ()
        root.destroy ()
    Tkinter.Button (text = "ok", command = rget).pack (side = Tkinter.LEFT)
    root.mainloop ()
    return reponse [0]

print "reponse ", question ("texte de la question")
```

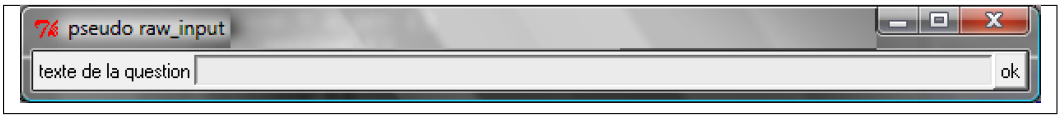



**Figure 3.1 :** *Fenêtre apparaissant lors de l'exécution du programme page 62 proposant une version graphique de la fonction* `raw_input`.

**Remarque 3.7 : `if None`, `if 0`**
L'écriture de certains tests peut encore être réduite lorsqu'on cherche à comparer une variable entière, booléenne ou `None` comme le précise le tableau 3.2.5.

**Table 3.3 :** *Ce tableau précise le sens de certains tests lorsque leur écriture est tronquée. On suppose que* `v` *est une variable.*

| type | test | test équivalent |
|---|---|---|
| `bool` | `if v :` | `if v == True :` |
| `bool` | `if not v :` | `if v == False :` |
| `int` | `if v :` | `if v != 0 :` |
| `int` | `if not v :` | `if v == 0 :` |
| `float` | `if v :` | `if v != 0.0 :` |
| `float` | `if not v :` | `if v == 0.0 :` |
| `None` | `if v :` | `if v != None :` |
| `None` | `if not v :` | `if v == None :` |

### 3.2.6 Passer, instruction `pass`

Dans certains cas, aucune instruction ne doit être exécutée même si un test est validé. En *Python*, le corps d'un test ne peut être vide, il faut utiliser l'instruction `pass`. Lorsque celle-ci est manquante, *Python* affiche un message d'erreur.

```
signe = 0
x = 0
if x < 0 : signe = -1
elif x == 0:
   pass          # signe est déjà égal à 0
else : signe = 1
```

Dans ce cas précis, si l'instruction `pass` est oubliée, l'interpréteur *Python* génère l'erreur suivante :

```
File "nopass.py", line 6
    else :
    ^
IndentationError: expected an indented block
```

## 3.3 Boucles

> **Définition 3.8 : boucle**
> Les boucles permettent de répéter une séquence d'instructions tant qu'une certaine condition est vérifiée.

Le langage *Python* propose deux types de boucles. La boucle `while` suit scrupuleusement la définition 3.8. La boucle `for` est une boucle `while` déguisée (voir paragraphe 3.3.2), elle propose une écriture simplifiée pour répéter la même séquence d'instructions pour tous les éléments d'un ensemble.

### 3.3.1 Boucle `while`

#### 3.3.1.1 Syntaxe et exemples

L'implémentation d'une boucle de type `while` suit le schéma d'écriture suivant :

> **syntaxe 3.9 : boucle `while`**
>
> ```
> while cond :
>     instruction 1
>     ...
>     instruction n
> ```
>
> Où `cond` est une condition qui détermine la poursuite de la répétition des instructions incluses dans la boucle. Tant que celle-ci est vraie, les instructions 1 à $n$ sont exécutées.

Tout comme les tests, la remarque 3.4 concernant l'indentation reste vraie. Le décalage des lignes d'un cran vers la droite par rapport à l'instruction `while` permet de les inclure dans la boucle comme le montre l'exemple suivant.

```
n = 0
while n < 3:
   print "à l'intérieur ", n
   n += 1
print "à l'extérieur ", n
```

Résultat :

```
à l'intérieur  0
à l'intérieur  1
à l'intérieur  2
à l'extérieur  3
```

#### 3.3.1.2 Condition

Les conditions suivent la même syntaxe que celles définies lors des tests (voir paragraphes 3.2.2 et 3.2.3). A moins d'inclure l'instruction `break`[2] qui permet de sortir prématurément d'une boucle, la condition qui régit cette boucle doit nécessairement être modifiée à l'intérieur de celle-ci. Dans le cas contraire, on appelle une telle boucle une *boucle infinie* puisqu'il est impossible d'en sortir. L'exemple suivant contient une boucle infinie car le symbole = est manquant dans la dernière instruction. La variable `n` n'est jamais modifiée et la condition `n < 3` toujours vraie.

```
n = 0
while n < 3 :
   print n
   n + 1        # n n'est jamais modifié, l'instruction correcte serait n += 1
```

### 3.3.2 Boucle `for`

#### 3.3.2.1 Syntaxe et exemples

L'implémentation d'une boucle de type `for` suit le schéma d'écriture suivant :

**syntaxe 3.10 : boucle `for`**

```
for x in set :
    instruction 1
    ...
    instruction n
```

Où `x` est un élément de l'ensemble `set`. Les instructions 1 à $n$ sont exécutées pour chaque élément `x` de l'ensemble `set`. Cet ensemble peut être une chaîne de caractères, un T-uple, une liste, un dictionnaire ou tout autre type incluant des itérateurs qui sont présentés au chapitre 4 concernant les classes (page 106).

Tout comme les tests, la remarque 3.4 concernant l'indentation reste vraie. L'exemple suivant affiche tous les éléments d'un T-uple à l'aide d'une boucle `for`.

```
t = (1,2,3,4)
for x in t:       # affiche les nombres 1,2,3,4
    print x       # chacun sur une ligne différente
```

2. L'instruction `break` est décrite au paragraphe 3.3.4.2, page 68.

Lors de l'affichage d'un dictionnaire, les éléments n'apparaissent pas triés ni dans l'ordre dans lequel ils y ont été insérés. L'exemple suivant montre comment afficher les clés et valeurs d'un dictionnaire dans l'ordre croissant des clés.

```
d = { 1:2, 3:4, 5:6, 7:-1, 8:-2 }
print d                # affiche le dictionnaire {8: -2, 1: 2, 3: 4, 5: 6, 7: -1}
k = d.keys ()
print k                # affiche les clés [8, 1, 3, 5, 7]
k.sort ()
print k                # affiche les clés triées [1, 3, 5, 7, 8]
for x in k:            # affiche les éléments du dictionnaire
    print x,":",d [x]  # triés par clés croissantes
```

Le langage *Python* propose néanmoins la fonction `sorted` qui réduit l'exemple suivant en trois lignes :

```
d = { 1:2, 3:4, 5:6, 7:-1, 8:-2 }
for x in sorted(d):    # pour les clés dans l'ordre croissant
    print x,":",d [x]
```

La boucle la plus répandue est celle qui parcourt des indices entiers compris entre 0 et $n-1$. On utilise pour cela la boucle `for` et la fonction `range` ou `xrange` comme dans l'exemple qui suit. La différence entre ces deux fonctions a déjà été présentée au paragraphe 2.3.1.3 (page 45).

```
sum = 0
N   = 10
for n in range(0,N):     # ou for n in xrange(0,N):  (plus rapide)
   sum += n              # additionne tous les entiers compris entre 0 et N-1
```

Ou encore pour une liste quelconque :

```
l   = [ 4, 5, 3, -6, 7, 9]
sum = 0
for n in range(0,len (l)):  # ou for n in xrange(0,len (l)) :  (plus rapide)
   sum += l [n]             # additionne tous les éléments de l
```

#### 3.3.2.2 Listes et boucle `for`

Le paragraphe 2.3.1.4 a montré comment le mot-clé `for` peut être utilisé pour simplifier la création d'une liste à partir d'une autre. La syntaxe suit le schéma suivant :

> **syntaxe 3.11 : boucle `for`**
>
> `[ expression for x in ensemble ]`
>
> Où `expression` est une expression numérique incluant ou non `x`, la variable de la boucle, `ensemble` est un ensemble d'éléments, T-uple, liste, dictionnaire.

Cette syntaxe permet de résumer en une ligne la création de la séquence `y` du programme suivant.

```
y = list ()
for i in range(0,5) :
```

```
   y.append (i+1)
print y             # affiche [1,2,3,4,5]
```

Les trois lignes du milieu sont maintenant résumées en une seule :

```
y = [ i+1 for i in range(0,5)] # résume trois lignes du programme précédent
print y                       # affiche [1,2,3,4,5]
```

Un autre exemple de cette syntaxe réduite a été présenté au paragraphe 2.3.1.4 page 46. Cette écriture condensée est bien souvent plus lisible même si tout dépend des préférences de celui qui programme.

#### 3.3.2.3 Itérateurs

Toute boucle `for` peut s'appliquer sur un objet muni d'un itérateur tels que les chaînes de caractères, T-uples, les listes, les dictionnaires.

```
d = ["un", "deux", "trois"]
for x in d:
   print x          # affichage de tous les éléments de d
```

Cette syntaxe réduite a déjà été introduite pour les listes et les dictionnaires au chapitre précédent. Il existe une version équivalente avec la boucle `while` utilisant de façon explicite les itérateurs. Il peut être utile de lire le chapitre suivant sur les classes et le chapitre 5 sur les exceptions avant de revenir sur la suite de cette section qui n'est de toutes façons pas essentielle. L'exemple précédent est convertible en une boucle `while` en faisant apparaître explicitement les itérateurs (voir paragraphe 4.4.2). Un itérateur est un objet qui permet de parcourir aisément un ensemble. La fonction `it = iter(e)` permet d'obtenir un itérateur `it` sur l'ensemble `e`. L'appel à l'instruction `it.next()` parcourt du premier élément jusqu'au dernier en retournant la valeur de chacun d'entre eux. Lorsqu'il n'existe plus d'élément, l'exception `StopIteration` est déclenchée (voir paragraphe 5). Il suffit de l'intercepter pour mettre fin au parcours.

```
d = ["un", "deux", "trois"]
it = iter (d)                     # obtient un itérateur sur d
while True:
    try: x = it.next ()           # obtient l'élément suivant, s'il n'existe pas
    except StopIteration: break   # déclenche une exception
    print x                       # affichage de tous les éléments de d
```

#### 3.3.2.4 Plusieurs variables de boucles

Jusqu'à présent, la boucle `for` n'a été utilisée qu'avec une seule variable de boucle, comme dans l'exemple suivant où on parcourt une liste de T-uple pour les afficher.

```
d = [ (1,0,0), (0,1,0), (0,0,1) ]
for v in d: print v
```

Lorsque les éléments d'un ensemble sont des T-uples, des listes ou des dictionnaires composés de taille fixe, il est possible d'utiliser une notation qui rappelle les affectations multiples (voir paragraphe 2.4.4). L'exemple précédent devient dans ce cas :

```
d = [ (1,0,0), (0,1,0), (0,0,1) ]
for x,y,z in d: print x,y,z
```

Cette écriture n'est valable que parce que chaque élément de la liste `d` est un T-uple composé de trois nombres. Lorsqu'un des éléments est de taille différente à celle des autres, comme dans l'exemple suivant, une erreur survient.

```
d = [ (1,0,0), (0,1,0,6), (0,0,1) ]  # un élément de taille quatre
for x,y,z in d: print x,y,z
```

Ce programme génère l'erreur suivante :

```
Traceback (most recent call last):
  File "c:\temp\delete.py", line 2, in -toplevel-
    for x,y,z in d: print x,y,z
ValueError: unpack tuple of wrong size
```

Cette syntaxe est très pratique associée à la fonction `zip` (voir paragraphe 2.3.1.5). Il est alors possible de parcourir plusieurs séquences (T-uple, liste, dictionnaire) simultanément.

```
a = range(0,5)
b = [x**2 for x in a]
for x,y in zip (a,b):
    print y, " est le carré de ", x
    # affichage à droite
```

```
0  est le carré de  0
1  est le carré de  1
4  est le carré de  2
9  est le carré de  3
16  est le carré de  4
```

### 3.3.3 Ecriture condensée

Comme pour les tests, lorsque les boucles ne contiennent qu'une seule instruction, il est possible de l'écrire sur la même ligne que celle de la déclaration de la boucle `for` ou `while`.

```
d = ["un", "deux", "trois"]
for x in d: print x          # une seule instruction
```

Il existe peu de cas où la boucle `while` s'écrit sur une ligne car elle inclut nécessairement une instruction permettant de modifier la condition d'arrêt.

```
d = ["un", "deux", "trois"]
i = 0
while d [i] != "trois" : i += 1
print "trois a pour position ", i
```

### 3.3.4 Pilotage d'une boucle

#### 3.3.4.1 Passer à l'itération suivante : `continue`

Pour certains éléments d'une boucle, lorsqu'il n'est pas nécessaire d'exécuter toutes les instructions, il est possible de passer directement à l'élément suivant ou l'itération suivante. Le programme suivant utilise la crible d'Eratosthène [3] pour dénicher tous les nombres premiers compris entre 1 et 99.

```
d = dict ()
for i in range(1,100):            # d [i] est vrai si i est un nombre premier
    d [i] = True                  # au début, comme on ne sait pas, on suppose
                                  # que tous les nombres sont premiers
for i in range(2,100):
                                  # si d [i] est faux,
   if not d [i]: continue         # les multiples de i ont déjà été cochés
                                  # et peut passer à l'entier suivant
   for j in range (2,100):
       if i*j < 100:
           d [i*j] = False        # d [i*j] est faux pour tous les multiples de i
                                  # inférieurs à 100
print "liste des nombres premiers"
for i in d:
    if d [i]: print i
```

Ce programme est équivalent au suivant :

```
d = dict ()
for i in range(1,100): d [i] = True

for i in range(2,100):
   if d [i]:
       for j in range (2,100):
           if i*j < 100 :
               d [i*j] = False

print "liste des nombres premiers"
for i in d:
    if d [i]: print i
```

Le mot-clé `continue` évite de trop nombreuses indentations et rend les programmes plus lisibles.

#### 3.3.4.2 Sortie prématurée : `break`

Lors de l'écriture d'une boucle `while`, il n'est pas toujours adéquat de résumer en une seule condition toutes les raisons pour lesquelles il est nécessaire d'arrêter l'exécution de cette boucle. De même, pour une boucle `for`, il n'est pas toujours utile de

3. Le crible d'Eratosthène est un algorithme permettant de déterminer les nombres premiers. Pour un nombre premier $p$, il paraît plus simple de considérer tous les entiers de $p-1$ à 1 pour savoir si l'un d'eux divise $p$. C'est ce qu'on fait lorsqu'on doit vérifier le caractère premier d'un seul nombre. Pour plusieurs nombres à la fois, le crible d'Eratosthène est plus efficace : au lieu de s'intéresser aux diviseurs, on s'intéresse aux multiples d'un nombre. Pour un nombre $i$, on sait que $2i$, $3i$, ... ne sont pas premiers. On les raye de la liste. On continue avec $i+1$, $2(i+1)$, $3(i+1)$...

visiter tous les éléments de l'ensemble à parcourir. C'est le cas par exemple lorsqu'on recherche un élément, une fois qu'il a été trouvé, il n'est pas nécessaire d'aller plus loin. L'instruction `break` permet de quitter l'exécution d'une boucle.

```
l = [6,7,5,4,3]
n = 0
c = 5
for x in l:
   if x == c: break   # l'élément a été trouvé, on sort de la boucle
   n += 1             # si l'élément a été trouvé, cette instruction
                      # n'est pas exécutée
print "l'élément ",c, " est en position ",
print n               # affiche l'élément 5 est en position 2
```

Si deux boucles sont imbriquées, l'instruction `break` ne sort que de la boucle dans laquelle elle est insérée. L'exemple suivant vérifie si un entier est la somme des carrés de deux entiers compris entre 1 et 20.

```
set = range (1,21)
n = 53
for x in set:
    for y in set:
        c = x*x + y*y
        if c == n: break
    if c == n: break   # cette seconde instruction break est nécessaire
                       # pour sortir de la seconde boucle
                       # lorsque la solution a été trouvée
if c == n:
    # le symbole \ permet de passer à la ligne sans changer d'instruction
    print n, " est la somme des carrés de deux entiers :", \
          x, "*", x, "+", y, "*", y, "=", n
else:
    print n, " n'est pas la somme des carrés de deux entiers"
```

Le programme affiche :

```
53  est la somme des carrés de deux entiers : 2 * 2 + 7 * 7 = 53
```

#### 3.3.4.3 Fin normale d'une boucle : `else`

Le mot-clé `else` existe aussi pour les boucles et s'utilise en association avec le mot-clé `break`. L'instruction `else` est placée à la fin d'une boucle, indentée au même niveau que `for` ou `while`. Les lignes qui suivent le mot-clé `else` ne sont exécutées que si aucune instruction `break` n'a été rencontrée dans le corps de la boucle. On reprend l'exemple du paragraphe précédent. On recherche cette fois-ci la valeur 1 qui ne se trouve pas dans la liste `L`. Les lignes suivant le test `if x == c` ne seront jamais exécutées au contraire de la dernière.

```
L = [6,7,5,4,3]
n = 0
c = 1
for x in L :
   if x == c :
       print "l'élément ", c, " est en position ", n
```

```
        break
    n += 1
else:
    print "aucun élément ", c, " trouvé"  # affiche aucun élément  1  trouvé
```

Les lignes dépendant de la clause `else` seront exécutées dans tous les cas où l'exécution de la boucle n'est pas interrompue par une instruction `break` ou une instruction `return` [4] ou par la levée d'une exception [5].

#### 3.3.4.4 Suppression ou ajout d'éléments lors d'une boucle

En parcourant la liste en se servant des indices, il est possible de supprimer une partie de cette liste. Il faut néanmoins faire attention à ce que le code ne produise pas d'erreur comme c'est le cas pour le suivant. La boucle `for` parcourt la liste `range(0,len(li))` qui n'est pas modifiée en même temps que l'instruction `del li[i:i+2]`.

```
li = range (0,10)
print li                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in range (0, len (li)):
    if i == 5 :
        del li [i:i+2]
    print li [i]        # affiche successivement 0, 1, 2, 3, 4, 7, 8, 9 et
                        # produit une erreur
print li
```

Le programme suivant marche parfaitement puisque cette fois-ci la boucle parcourt la liste `li`. En revanche, pour la suppression d'une partie de celle-ci, il est nécessaire de conserver en mémoire l'indice de l'élément visité. C'est le rôle de la variable `i`.

```
li = range (0,10)
print li                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
i = 0
for t in li :
    if i == 5 : del li [i:i+2]
    i = i+1
    print t             # affiche successivement 0, 1, 2, 3, 4, 5, 8, 9
print li                # affiche [0, 1, 2, 3, 4, 7, 8, 9]
```

Le langage *Python* offre la possibilité de supprimer des éléments d'une liste alors même qu'on est en train de la parcourir. Le programme qui suit ne marche pas puisque l'instruction `del i` ne supprime pas un élément de la liste mais l'identificateur `i` qui prendra une nouvelle valeur lors du passage suivant dans la boucle.

```
li = range (0,10)
print li                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in li :
    if i == 5 : del i
print li                # affiche [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4. voir les fonctions au paragraphe 3.4
5. voir paragraphe 5

On pourrait construire des exemples similaires dans le cadre de l'ajout d'un élément à la liste. Il est en règle générale déconseillé de modifier une liste, un dictionnaire pendant qu'on le parcourt. Malgré tout, si cela s'avérait indispensable, il convient de faire plus attention dans ce genre de situations.

## 3.4 Fonctions

Les fonctions sont des petits programmes qui effectuent des tâches plus précises que le programme entier. On peut effectivement écrire un programme sans fonction mais ils sont en général illisibles. Utiliser des fonctions implique de découper un algorithme en tâches élémentaires. Le programme final est ainsi plus facile à comprendre. Un autre avantage est de pouvoir plus facilement isoler une erreur s'il s'en produit une : il suffit de tester une à une les fonctions pour déterminer laquelle retourne un mauvais résultat. L'avantage le plus important intervient lorsqu'on doit effectuer la même chose à deux endroits différentes d'un programme : une seule fonction suffit et elle sera appelée à ces deux endroits [6].

### 3.4.1 Définition, syntaxe

**Définition 3.12 : fonction**
Une fonction est une partie d'un programme - ou sous-programme - qui fonctionne indépendamment du reste du programme. Elle reçoit une liste de paramètres et retourne un résultat. Le corps de la fonction désigne toute instruction du programme qui est exécutée si la fonction est appelée.

Lorsqu'on écrit ses premiers programme, on écrit souvent des fonctions plutôt longues avant de s'apercevoir que certains parties sont identiques ailleurs. On extrait donc la partie répétée pour en faire une fonction. Avec l'habitude, on finit par écrire des fonctions plus petites et réutilisables.

**syntaxe 3.13 : fonction, définition**

```
def fonction_nom (par_1, ..., par_n) :
    instruction_1
    ...
    instruction_n
    return res_1, ..., res_n
```

`fonction_nom` est le nom de la fonction, il suit les mêmes règles que le nom des variables. `par_1` à `par_n` sont les noms des paramètres et `res_1` à `res_n` sont les résultats retournés par la fonction. Les instructions associées à une fonction doivent être indentées par rapport au mot-clé `def`.

S'il n'y a aucun résultat, l'instruction `return` est facultative ou peut être utilisée seule

6. Pour les utilisateurs experts : en langage *Python*, les fonctions sont également des variables, elles ont un identificateur et une valeur qui est dans ce cas un morceau de code. Cette précision explique certaines syntaxes du chapitre 8 sur les interfaces graphiques ou celle introduite en fin de chapitre au paragraphe 3.7.10.

sans être suivie par une valeur ou une variable. Cette instruction peut apparaître plusieurs fois dans le code de la fonction mais une seule d'entre elles sera exécutée. A partir de ce moment, toute autre instruction de la fonction sera ignorée. Pour exécuter une fonction ainsi définie, il suffit de suivre la syntaxe suivante :

**syntaxe 3.14 : fonction, appel**

```
x_1, ..., x_n = fonction_nom (valeur_1, valeur_2, ..., valeur_n)
```

Où `fonction_nom` est le nom de la fonction, `valeur_1` à `valeur_n` sont les noms des paramètres, `x_1` à `x_n` reçoivent les résultats retournés par la fonction. Cette affectation est facultative. Si on ne souhaite pas conserver les résultats, on peut donc appeler la fonction comme suit :

```
fonction_nom (valeur_1, valeur_2, ..., valeur_n)
```

Lorsqu'on commence à programmer, il arrive parfois qu'on confonde le rôle des mots-clés `print` et `return`. Dans ce cas, il faut se reporter à la remarque 10.3 page 255.

### 3.4.2 Exemple

Le programme suivant utilise deux fonctions. La première convertit des coordonnées cartésiennes en coordonnées polaires. Elle prend deux réels en paramètres et retourne deux autres réels. La seconde fonction affiche les résultats de la première pour tout couple de valeurs $(x, y)$. Elle ne retourne aucun résultat.

```
import math
def coordonnees_polaires (x,y):
    rho     = math.sqrt(x*x+y*y)   # calcul la racine carrée de x*x+y*y
    theta   = math.atan2 (y,x)     # calcule l'arc tangente de y/x en tenant
                                   # compte des signes de x et y
    return rho, theta

def affichage (x,y):
    r,t = coordonnees_polaires(x,y)
    print "cartésien (%f,%f) --> polaire (%f,%f degrés)" \
                % (x,y,r,math.degrees(t))

affichage (1,1)
affichage (0.5,1)
affichage (-0.5,1)
affichage (-0.5,-1)
affichage (0.5,-1)
```

Le programme affiche les lignes suivantes :

```
cartésien (1.000000,1.000000) --> polaire (1.414214,45.000000 degrés)
cartésien (0.500000,1.000000) --> polaire (1.118034,63.434949 degrés)
cartésien (-0.500000,1.000000) --> polaire (1.118034,116.565051 degrés)
cartésien (-0.500000,-1.000000) --> polaire (1.118034,-116.565051 degrés)
cartésien (0.500000,-1.000000) --> polaire (1.118034,-63.434949 degrés)
```

### 3.4.3 Paramètres avec des valeurs par défaut

Lorsqu'une fonction est souvent appelée avec les mêmes valeurs pour ses paramètres, il est possible de spécifier pour ceux-ci une valeur par défaut.

**syntaxe 3.15 : fonction, valeur par défaut**

```
def fonction_nom (param_1, param_2 = valeur_2, ..., param_n = valeur_n):
    ...
```

Où `fonction_nom` est le nom de la fonction. `param_1` à `param_n` sont les noms des paramètres, `valeur_2` à `valeur_n` sont les valeurs par défaut des paramètres `param_2` à `param_n`. La seule contrainte lors de cette définition est que si une valeur par défaut est spécifiée pour un paramètre, alors tous ceux qui suivent devront eux aussi avoir une valeur par défaut.

Exemple :

```
def commander_carte_orange (nom, prenom, paiement = "carte", nombre = 1, zone = 2):
    print "nom : ", nom
    print "prénom : ", prenom
    print "paiement : ", paiement
    print "nombre : ", nombre
    print "zone :", zone

commander_carte_orange ("Dupré", "Xavier", "chèque")
        # les autres paramètres nombre et zone auront pour valeur
        # leurs valeurs par défaut
```

Il est impossible qu'un paramètre sans valeur par défaut associée se situe après un paramètre dont une valeur par défaut est précisée. Le programme suivant ne pourra être exécuté.

```
def commander_carte_orange (nom, prenom, paiement = "carte", nombre = 1, zone):
    print "nom : ", nom
    # ...
```

Il déclenche l'erreur suivante :

```
  File "problem_zone.py", line 1
    def commander_carte_orange (nom, prenom, paiement = "carte", nombre = 1, zone):
SyntaxError: non-default argument follows default argument
```

**Remarque 3.16 : valeurs par défaut de type modifiable**
Les valeurs par défaut de type modifiable (liste, dictionnaire, classes) peuvent introduire des erreurs inattendues dans les programmes comme le montre l'exemple suivant :

```
def fonction (l = [0,0]) :
    l [0] += 1
    return l

print fonction ()         # affiche [1,0] : résultat attendu
print fonction ()         # affiche [2,0] : résultat surprenant
print fonction ( [0,0])   # affiche [1,0] : résultat attendu
```

L'explication provient du fait que la valeur par défaut est une liste qui n'est pas recréée à chaque appel : c'est la même liste à chaque fois que la fonction est appelée sans paramètre. Pour remédier à cela, il faudrait écrire :

```
import copy
def fonction (l = [0,0]) :
    l = copy.copy (l)
    l [0] += 1
    return l
```

L'exercice 12.4.7 (page 316) propose un exemple plus complet, voire retors.

### 3.4.4 Ordre des paramètres

Le paragraphe 3.4.1 a présenté la syntaxe d'appel a une fonction. Lors de l'appel, le nom des paramètres n'intervient plus, supposant que chaque paramètre reçoit pour valeur celle qui a la même position que lui lors de l'appel à la fonction. Il est toutefois possible de changer cet ordre en précisant quel paramètre doit recevoir quelle valeur.

**syntaxe 3.17 : fonction, ordre des paramètres**

```
x_1, ..., x_n = fonction_nom (param_1 = valeur_1, ..., param_n = valeur_n)
```

Où `fonction_nom` est le nom de la fonction, `param_1` à `param_n` sont les noms des paramètres, `valeur_1` à `valeur_n` sont les valeurs que reçoivent ces paramètres. Avec cette syntaxe, l'ordre d'écriture n'importe pas. La valeur `valeur_i` sera toujours attribuée à `param_i`. Les variables `x_1` à `x_n` reçoivent les résultats retournés par la fonction. L'ordre des résultats ne peut pas être changé. S'il y a plusieurs résultats retournés, il est impossible de choisir lesquels conserver : soit tous, soit aucun.

Exemple :

```
def identite (nom, prenom):
    print "nom : ", nom, " prénom : ", prenom

identite("Xavier", "Dupré")                 # nom :  Xavier prénom :  Dupré
identite(prenom = "Xavier", nom = "Dupré")  # nom :  Dupré  prénom :  Xavier
```

Cette possibilité est intéressante surtout lorsqu'il y a de nombreux paramètres par défaut et que seule la valeur d'un des derniers paramètres doit être changée.

```
def commander_carte_orange (paiement = "carte", nombre = 1, zone = 2):
    print "paiement : ", paiement
    print "nombre : ", nombre
    print "zone :", zone

commander_carte_orange (zone = 5)  # seule la valeur par défaut
                                   # du paramètre zone sera changée
```

### 3.4.5 Surcharge de fonction

Contrairement à d'autres langages, *Python* n'autorise pas la surcharge de fonction. Autrement dit, il n'est pas possible que plusieurs fonctions portent le même nom même si chacune d'entre elles a un nombre différent de paramètres.

```
def fonction (a,b):
    return a + b

def fonction (a,b,c):
    return a + b + c

print fonction (5,6)
print fonction (5,6,7)
```

Le petit programme précédent est syntaxiquement correct mais son exécution génère une erreur parce que la seconde définition de la fonction `fonction` efface la première.

```
Traceback (most recent call last):
  File "cours4.py", line 7, in ?
    print fonction (5,6)
TypeError: fonction() takes exactly 3 arguments (2 given)
```

### 3.4.6 Commentaires

Le langage *Python* propose une fonction `help` qui retourne pour chaque fonction un commentaire ou mode d'emploi qui indique comment se servir de cette fonction. L'exemple suivant affiche le commentaire associé à la fonction `round`.

```
>>> help (round)
```

```
Help on built-in function round:

round(...)
    round(number[, ndigits]) -> floating point number

    Round a number to a given precision in decimal digits (default 0 digits).
    This always returns a floating point number.  Precision may be negative.
```

Lorsqu'on utilise cette fonction `help` sur la fonction `coordonnees_polaires` définie dans l'exemple du paragraphe précédent 3.4.2, le message affiché n'est pas des plus explicites.

```
>>> help (coordonnees_polaires)
```

```
Help on function coordonnees_polaires in module __main__:

coordonnees_polaires(x, y)
```

Pour changer ce message, il suffit d'ajouter en première ligne du code de la fonction une chaîne de caractères.

```
import math
def coordonnees_polaires (x,y):
    """convertit des coordonnées cartésiennes en coordonnées polaires
    (x,y) --> (pho,theta)"""
    rho     = math.sqrt(x*x+y*y)
    theta   = math.atan2 (y,x)
    return rho, theta
help (coordonnees_polaires)
```

Le programme affiche alors le message d'aide suivant nettement plus explicite :

```
Help on function coordonnees_polaires in module __main__:

coordonnees_polaires(x, y)
    convertit des coordonnées cartésiennes en coordonnées polaires
    (x,y) --> (pho,theta)
```

Il est conseillé d'écrire ce commentaire pour toute nouvelle fonction avant même que son corps ne soit écrit. L'expérience montre qu'on oublie souvent de l'écrire après.

### 3.4.7 Paramètres modifiables

Les paramètres de types immuables et modifiables se comportent de manières différentes à l'intérieur d'une fonction. Ces paramètres sont manipulés dans le corps de la fonction, voire modifiés parfois. Selon le type du paramètre, ces modifications ont des répercussions à l'extérieur de la fonction.

Les types immuables ne peuvent être modifiés et cela reste vrai. Lorsqu'une fonction accepte un paramètre de type immuable, elle ne reçoit qu'une copie de sa valeur. Elle peut donc modifier ce paramètre sans que la variable ou la valeur utilisée lors de l'appel de la fonction n'en soit affectée. On appelle ceci un passage de paramètre par valeur. A l'opposé, toute modification d'une variable d'un type modifiable à l'intérieur d'une fonction est répercutée à la variable qui a été utilisée lors de l'appel de cette fonction. On appelle ce second type de passage un passage par adresse.

L'exemple suivant utilise une fonction `somme_n_premier_terme` qui modifie ces deux paramètres. Le premier, `n`, est immuable, sa modification n'a aucune incidence sur la variable `nb`. En revanche, le premier élément du paramètre `liste` reçoit la valeur 0. Le premier élément de la liste `l` n'a plus la même valeur après l'appel de la fonction `somme_n_premier_terme` que celle qu'il avait avant.

```
def somme_n_premier_terme(n,liste):
    """calcul la somme des n premiers termes d'une liste"""
    somme = 0
    for i in liste:
        somme += i
        n -= 1              # modification de n (type immuable)
        if n <= 0: break
    liste[0] = 0            # modification de liste (type modifiable)
    return somme
```

```
l = [1,2,3,4]
nb = 3
print "avant la fonction ",nb,l   # affiche   avant la fonction  3 [1, 2, 3, 4]
s = somme_n_premier_terme (nb,l)
print "après la fonction ",nb,l   # affiche   après la fonction  3 [0, 2, 3, 4]
print "somme : ", s               # affiche   somme :  6
```

La liste `l` est modifiée à l'intérieur de la fonction `somme_n_premier_terme` comme l'affichage suivant le montre. En fait, à l'intérieur de la fonction, la liste `l` est désignée par l'identificateur `liste`, c'est la même liste. La variable `nb` est d'un type immuable. Sa valeur a été recopiée dans le paramètre `n` de la fonction `somme_n_premier_terme`. Toute modification de `n` à l'intérieur de cette fonction n'a aucune répercussion à l'extérieur de la fonction.

**Remarque 3.18 : passage par adresse**
Dans l'exemple précédent, il faut faire distinguer le fait que la liste passée en paramètre ne soit que modifiée et non changée. L'exemple suivant inclut une fonction qui affecte une nouvelle valeur au paramètre `liste` sans pour autant modifier la liste envoyée en paramètre.

```
def fonction (liste):
    liste = []

liste = [0,1,2]
print liste       # affiche [0,1,2]
fonction (liste)
print liste       # affiche [0,1,2]
```

Il faut considérer dans ce programme que la fonction `fonction` reçoit un paramètre appelé `liste` mais utilise tout de suite cet identificateur pour l'associer à un contenu différent. L'identificateur `liste` est en quelque sorte passé du statut de paramètre à celui de variable locale. La fonction associe une valeur à `liste` - ici, une liste vide - sans toucher à la valeur que cet identificateur désignait précédemment.

Le programme qui suit est différent du précédent mais produit les mêmes effets. Ceci s'explique par le fait que le mot-clé `del` ne supprime pas le contenu d'une variable mais seulement son identificateur. Le langage *Python* détecte ensuite qu'un objet n'est plus désigné par aucun identificateur pour le supprimer. Cette remarque est à rapprocher de celles du paragraphe 4.6.

```
def fonction (liste):
    del liste

liste = [0,1,2]
print liste       # affiche [0,1,2]
fonction (liste)
print liste       # affiche [0,1,2]
```

Le programme qui suit permet cette fois-ci de vider la liste `liste` passée en paramètre à la fonction `fonction`. La seule instruction de cette fonction modifie vraiment le contenu désigné par l'identificateur `liste` et cela se vérifie après l'exécution de cette fonction.

```
def fonction (liste):
    del liste[0:len(liste)]  # on peut aussi écrire : liste[:] = []
```

```
liste = [0,1,2]
print liste       # affiche [0,1,2]
fonction (liste)
print liste       # affiche []
```

### 3.4.8 Fonction récursive

> **Définition 3.19 : fonction récursive**
> Une fonction récursive est une fonction qui s'appelle elle-même.

La fonction récursive la plus fréquemment citée en exemple est la fonction factorielle. Celle-ci met en évidence les deux composantes d'une fonction récursive, la récursion proprement dite et la condition d'arrêt.

```
def factorielle(n):
    if n == 0 : return 1
    else : return n * factorielle(n-1)
```

La dernière ligne de la fonction `factorielle` est la récursion tandis que la précédente est la condition d'arrêt, sans laquelle la fonction ne cesserait de s'appeler, empêchant le programme de terminer son exécution. Si celle-ci est mal spécifiée ou absente, l'interpréteur *Python* affiche une suite ininterrompue de messages. *Python* n'autorise pas plus de 1000 appels récursifs : `factorielle(999)` provoque nécessairement une erreur d'exécution même si la condition d'arrêt est bien spécifiée.

```
Traceback (most recent call last):
  File "fact.py", line 5, in <module>
    factorielle(999)
  File "fact.py", line 3, in factorielle
    else : return n * factorielle(n-1)
  File "fact.py", line 3, in factorielle
    else : return n * factorielle(n-1)
  ...
```

La liste des messages d'erreurs est aussi longue qu'il y a eu d'appels à la fonction récursive. Dans ce cas, il faut transformer cette fonction en une fonction non récursive équivalente, ce qui est toujours possible.

```
def factorielle_non_recursive (n) :
    r = 1
    for i in range (2, n+1) :
        r *= i
    return r
```

### 3.4.9 Portée

#### 3.4.9.1 Portée des variables, des paramètres

Lorsqu'on définit une variable, elle n'est pas utilisable partout dans le programme. Par exemple, elle n'est pas utilisable avant d'avoir été déclarée au moyen d'une affectation. Le court programme suivant déclenche une erreur.

```
print x   # déclenche une erreur
```

```
Traceback (most recent call last):
  File "pas_declaree.py", line 1, in <module>
    print x
NameError: name 'x' is not defined
```

Il est également impossible d'utiliser une variable à l'extérieur d'une fonction où elle a été déclarée. Plusieurs fonctions peuvent ainsi utiliser le même nom de variable sans qu'à aucun moment, il n'y ait confusion. Le programme suivant déclenche une erreur identique à celle reproduite ci-dessus.

```
def portee_variable(x):
    var = x
    print var

portee_variable(3)
print var           # déclenche une erreur car var est déclarée dans
                    # la fonction portee_variable
```

> **Définition 3.20 : portée d'une variable**
> La portée d'une variable associée à un identificateur recouvre la portion du programme à l'intérieur de laquelle ce même identificateur la désigne. Ceci implique que, dans cette portion de code, aucune autre variable, aucune autre fonction, aucune autre classe, ne peut porter le même identificateur.

Une variable n'a donc d'existence que dans la fonction dans laquelle elle est déclarée. On appelle ce type de variable une variable locale. Par défaut, toute variable utilisée dans une fonction est une variable locale.

> **Définition 3.21 : variable locale**
> Une variable locale est une variable dont la portée est réduite à une fonction.

Par opposition aux variables locales, on définit les variables globales qui sont déclarées à l'extérieur de toute fonction.

> **Définition 3.22 : variable globale**
> Une variable globale est une variable dont la portée est l'ensemble du programme.

L'exemple suivant mélange variable locale et variable globale. L'identificateur `n` est utilisé à la fois pour désigner une variable globale égale à 1 et une variable locale égale à 1. A l'intérieur de la fonction, `n` désigne la variable locale égale à 2. A l'extérieur de la fonction, `n` désigne la variable globale égale à 1.

```
n = 1                   # déclaration d'une variable globale
def locale_globale():
    n = 2               # déclaration d'une variable locale
    print n             # affiche le contenu de la variable locale

print n                 # affiche 1
locale_globale()        # affiche 2
print n                 # affiche 1
```

Il est possible de faire référence aux variables globales dans une fonction par l'intermédiaire du mot-clé `global`. Celui-ci indique à la fonction que l'identificateur `n` n'est plus une variable locale mais désigne une variable globale déjà déclarée.

```
n = 1                   # déclaration d'une variable globale
def locale_globale():
    global n            # cette ligne indique que n désigne la variable globale
    n = 2               # change le contenu de la variable globale
    print n             # affiche le contenu de la variable globale

print n                 # affiche 1
locale_globale()        # affiche 2
print n                 # affiche 2
```

Cette possibilité est à éviter le plus possible car on peut considérer que `locale_globale` est en fait une fonction avec un paramètre caché. La fonction `locale_globale` n'est plus indépendante des autres fonctions puisqu'elle modifie une des données du programme.

#### 3.4.9.2 Portée des fonctions

Le langage *Python* considère les fonctions également comme des variables d'un type particulier. La portée des fonctions obéit aux mêmes règles que celles des variables. Une fonction ne peut être appelée que si elle a été définie avant son appel.

```
print type(factorielle)  # affiche <type 'function'>
```

Comme il est possible de déclarer des variables locales, il est également possible de définir des fonctions locales ou fonctions imbriquées. Une fonction locale n'est appelable qu'à l'intérieur de la fonction dans laquelle elle est définie. Dans l'exemple suivant, la fonction `affiche_pair` inclut une fonction locale qui n'est appelable que par cette fonction `affiche_pair`.

```
def affiche_pair():
    def fonction_locale(i):            # fonction locale ou imbriquée
        if i % 2 == 0 : return True
        else : return False
    for i in range(0,10):
        if fonction_locale(i):
```

```
            print i

affiche_pair()
fonction_locale(5)      # l'appel à cette fonction locale
                        # déclenche une erreur d'exécution
```

A l'intérieur d'une fonction locale, le mot-clé `global` désigne toujours les variables globales du programme et non les variables de la fonction dans laquelle cette sous-fonction est définie.

### 3.4.10 Nombre de paramètres variable

Il est possible de définir des fonctions qui prennent un nombre indéterminé de paramètres, lorsque celui-ci n'est pas connu à l'avance. Hormis les paramètres transmis selon le mode présenté dans les paragraphes précédents, des informations peuvent être ajoutées à cette liste lors de l'appel de la fonction, ces informations sont regroupées soit dans une liste de valeurs, soit dans une liste de couples (identificateur, valeur). La déclaration d'une telle fonction obéit à la syntaxe suivante :

**syntaxe 3.23 : fonction, nombre variable de paramètres**

```
def fonction (param_1, ..., param_n, *liste, **dictionnaire) :
```

Où `fonction` est un nom de fonction, `param_1` à `param_n` sont des paramètres de la fonction, `liste` est le nom de la liste qui doit recevoir la liste des valeurs seules envoyées à la fonction et qui suivent les paramètres (plus précisément, c'est un tuple), `dictionnaire` reçoit la liste des couples (identificateur, valeur).

L'appel à cette fonction suit quant à lui la syntaxe suivante :

**syntaxe 3.24 : fonction, nombre variable de paramètres (appel)**

```
fonction (valeur_1, ..., valeur_n, \
          liste_valeur_1, ..., liste_valeur_p, \
          nom_1 = v_1, ..., nom_q = v_q)
```

Où `fonction` est un nom de fonction, `valeur_1` à `valeur_n` sont les valeurs associées aux paramètres `param_1` à `param_n`, `liste_valeur_1` à `liste_valeur_p` formeront la liste `liste`, les couples `nom_1` : `v_1` à `nom_q` : `v_q` formeront le dictionnaire `dictionnaire`.

Exemple :

```
def fonction(p,*l,**d):
    print "p = ",p
    print "liste (tuple) l :", l
    print "dictionnaire d :", d

fonction (1,2,3,a=5,b=6) # 1 est associé au paramètre p
                         # 2 et 3 sont insérés dans la liste l
                         # a=5 et b=6 sont insérés dans le dictionnaire d
```

Ce programme affiche :

```
p =  1
liste l : (2, 3)
dictionnaire d : {'a': 5, 'b': 6}
```

A l'instar des paramètres par défaut, la seule contrainte de cette écriture est la nécessité de respecter l'ordre dans lequel les informations doivent apparaître. Lors de l'appel, les valeurs sans précision de nom de paramètre seront placées dans une liste (ici le tuple `l`). Les valeurs associées à un nom de paramètre seront placées dans un dictionnaire (ici `d`). Les valeurs par défaut sont obligatoirement placées après les paramètres non nommés explicitement.

Une fonction qui accepte des paramètres en nombre variable peut à son tour appeler une autre fonction acceptant des paramètres en nombre variable. Il faut pour cela se servir du symbole $*$ afin de transmettre à `fonction` les valeurs reçues par `fonction2`.

```
def fonction(p,*l,**d):
    print "p = ",p
    print "liste l :", l
    print "dictionnaire d :", d

def fonction2 (p, *l, **d) :
    l += (4,)              # on ajoute une valeur au tuple
    d ["c"] = 5            # on ajoute un couple (paramètre,valeur)
    fonction (p, *l, **d)  # ne pas oublier le symbole *

fonction2 (1,2,3,a=5,b=6)
```

Le programme affiche :

```
p =  1
liste l : (2, 3, 4)
dictionnaire d : {'a': 5, 'c': 5, 'b': 6}
```

### 3.4.11 Ecriture simplifiée pour des fonctions simples

Lorsque le code d'une fonction tient en une ligne et est le résultat d'une expression, il est possible de condenser son écriture à l'aide du mot-clé `lambda`.

> **syntaxe 3.25 : fonction `lambda`**
>
> `nom_fonction = lambda param_1, ..., param_n : expression`
>
> `nom_fonction` est le nom de la fonction, `param_1` à `param_n` sont les paramètres de cette fonction (ils peuvent également recevoir des valeurs par défaut), `expression` est l'expression retournée par la fonction.

L'exemple suivant utilise cette écriture pour définir la fonction `min` retournant le plus petit entre deux nombres positifs.

```
min = lambda x,y : (abs (x+y) - abs (x-y))/2
```

```
print min (1,2)      # affiche 1
print min (5,4)      # affiche 4
```

Cette écriture correspond à l'écriture non condensée suivante :

```
def min(x,y):
    return (abs (x+y) - abs (x-y))/2

print min (1,2)      # affiche 1
print min (5,4)      # affiche 4
```

### 3.4.12 Fonctions générateur

Le mot-clé `yield` est un peu à part. Utilisé à l'intérieur d'une fonction, il permet d'interrompre le cours de son exécution à un endroit précis de sorte qu'au prochain appel de cette fonction, celle-ci reprendra le cours de son exécution exactement au même endroit avec des variables locales inchangées. Le mot-clé `return` ne doit pas être utilisé. Ces fonctions ou *générateurs* sont utilisées en couple avec le mot-clé `for` pour simuler un ensemble. L'exemple suivant implémente une fonction `fonction_yield` qui simule l'ensemble des entiers compris entre 0 et n exclu.

```
def fonction_yield(n):
    i = 0
    while i < n-1:
        print "yield 1" # affichage : pour voir ce que fait le programme
        yield i         # arrête la fonction qui reprendra
        i = i+1         # à la ligne suivante lors du prochain appel
    print "yield 2"     # affichage : pour voir ce que fait le programme
    yield i             # arrête la fonction qui ne reprendra pas
                        # lors du prochain appel car le code de la fonction
                        # prend fin ici

for a in fonction_yield(2):
    print a                 # affiche tous les éléments que retourne la
                            # fonction fonction_yield, elle simule la liste
                            # [0,1]
print "-----------------------------------------------"
for a in fonction_yield(3):
    print a                 # nouvel appel, l'exécution reprend
                            # au début de la fonction,
                            # affiche tous les éléments que retourne la
                            # fonction fonction_yield, elle simule la liste
                            # [0,1,2]
```

Le programme affiche tous les entiers compris entre 0 et 4 inclus ainsi que le texte "`yield 1`" ou "`yield 2`" selon l'instruction `yield` qui a retourné le résultat. Lorsque la fonction a finalement terminé son exécution, le prochain appel agit comme si c'était la première fois qu'on l'appelait.

```
yield 1
0
yield 2
1
-----------------------------------------------
```

```
yield 1
0
yield 1
1
yield 2
2
```

### 3.4.13 Identificateur appelable

La fonction `callable` retourne un booléen permettant de savoir si un identificateur est une fonction (voir chapitre 4), de savoir par conséquent si tel identificateur est appelable comme une fonction.

```
x = 5
def y () :
    return None
print callable (x)  # affiche False car x est une variable
print callable (y)  # affiche True car y est une fonction
```

### 3.4.14 Fonctions ajoutées lors de l'exécution du programme

#### 3.4.14.1 Fonction `eval`

Cette fonction a déjà été abordée lors des paragraphes 2.4.1 (paragraphe 2.4.2, page 54). Un exemple a déjà été présenté page 54. Elle évalue toute chaîne de caractères contenant une expression écrite avec la syntaxe du langage *Python*. Cette expression peut utiliser toute variable ou toute fonction accessible au moment où est appelée la fonction `eval`.

```
x = 3
y = 4
print eval ("x*x+y*y+2*x*y")  # affiche 49
print (x+y)**2                # affiche 49
```

Si l'expression envoyée à la fonction `eval` inclut une variable non définie, l'interpréteur *Python* génère une erreur comme le montre l'exemple suivant.

```
x = 3
y = 4
print eval ("x*x+y*y+2*x*y+z")
```

La variable `z` n'est pas définie et l'expression n'est pas évaluable.

```
Traceback (most recent call last):
  File "c:\temp\cours.py", line 3, in -toplevel-
    print eval ("x*x+y*y+2*x*y+z")
  File "<string>", line 0, in -toplevel-
NameError: name 'z' is not defined
```

L'erreur se produit dans une chaîne de caractères traduite en programme informatique, c'est pourquoi l'interpréteur ne peut pas situer l'erreur dans un fichier. L'erreur ne se produit dans aucun fichier, cette chaîne de caractères pourrait être définie dans un autre.

#### 3.4.14.2 Fonctions `compile`, `exec`

Plus complète que la fonction `eval`, la fonction `compile` permet d'ajouter une ou plusieurs fonctions au programme, celle-ci étant définie par une chaîne de caractères. Le code est d'abord compilé (fonction `compile`) puis incorporé au programme (fonction `exec`) comme le montre l'exemple suivant.

```
import math
str = """def coordonnees_polaires (x,y):
    rho     = math.sqrt(x*x+y*y)
    theta   = math.atan2 (y,x)
    return rho, theta"""       # fonction définie par une chaîne de caractères

obj = compile(str,"","exec")   # fonction compilée
exec obj                       # fonction incorporée au programme
print coordonnees_polaires(1,1)# affiche (1.4142135623730951, 0.78539816339744828)
```

La fonction `compile` prend en fait trois arguments. Le premier est la chaîne de caractères contenant le code à compiler. Le second paramètre ("" dans l'exemple) contient un nom de fichier dans lequel seront placées les erreurs de compilation. Le troisième paramètre est une chaîne de caractères à choisir parmi "exec" ou "eval". Selon ce choix, ce sera la fonction `exec` ou `eval` qui devra être utilisée pour agréger le résultat de la fonction `compile` au programme. L'exemple suivant donne un exemple d'utilisation de la fonction `compile` avec la fonction `eval`.

```
import math
str = """math.sqrt(x*x+y*y)"""  # expression définie par une chaîne de caractères

obj = compile(str,"","eval")    # expression compilée
x = 1
y = 2
print eval (obj)                # résultat de l'expression
```

## 3.5 Indentation

L'indentation est synonyme de décalage. Pour toute boucle, test, fonction, et plus tard, toute définition de classe, le fait d'indenter ou décaler les lignes permet de définir une dépendance d'un bloc de lignes par rapport à un autre. Les lignes indentées par rapport à une boucle `for` dépendent de celle-ci puisqu'elle seront exécutées à chaque passage dans la boucle. Les lignes indentées par rapport au mot-clé `def` sont considérées comme faisant partie du corps de la fonction. La remarque 3.4 page 59 précise l'erreur que l'interpréteur *Python* retourne en cas de mauvaise indentation.

Contrairement à d'autres langages comme le *C* ou *PERL*, *Python* n'utilise pas de délimiteurs pour regrouper les lignes. L'indentation, souvent présentée comme un moyen de rendre les programmes plus lisibles, est ici intégrée à la syntaxe du langage. Il n'y a pas non plus de délimiteurs entre deux instructions autre qu'un passage à la ligne. Le caractère \ placé à la fin d'une ligne permet de continuer l'écriture d'une instruction à la ligne suivante.

## 3.6 Fonctions usuelles

Certaines fonctions sont communes aux dictionnaires et aux listes, elles sont également définis pour de nombreux objets présents dans les extensions du langages. Quelque soit le contexte, le résultat attendu à la même signification. Les plus courantes sont présentées par la table 3.4.

La fonction `map` permet d'écrire des boucles de façon simplifiée. Elle est utile dans le cas où on souhaite appliquer la même fonction à tous les éléments d'un ensemble. Par exemple les deux dernières lignes du programme suivant sont équivalentes.

```
l = [0,3,4,4,5,6]
print [ est_pair (i) for i in l ]  # affiche [0, 1, 0, 0, 1, 0]
print map (est_pair, l)            # affiche [0, 1, 0, 0, 1, 0]
```

Elle peut aider à simplifier l'écriture lorsque plusieurs listes sont impliquées. Ici encore, les deux dernières lignes sont équivalentes.

```
def addition (x,y) : return x + y
l = [0,3,4,4,5,6]
m = [1,3,4,5,6,8]
print [ addition (l [i], m [i]) for i in range (0, len (l)) ]
print map (addition, l, m)   # affiche [1, 6, 8, 9, 11, 14]
```

Il est possible de substituer `None` à la fonction `f` pour obtenir l'équivalent de la fonction `zip`.

```
print map (None, l,m)  # affiche [(0, 1), (3, 3), (4, 4), (4, 5), (5, 6), (6, 8)]
print zip (l,m)        # affiche [(0, 1), (3, 3), (4, 4), (4, 5), (5, 6), (6, 8)]
```

Comme pour les dictionnaires, la fonction `sorted` permet de parcourir les éléments d'une liste de façon ordonnée. Les deux exemples qui suivent sont presque équivalents. Dans le second, la liste `l` demeure inchangée alors qu'elle est triée dans le premier programme.

```
l   = [ 4, 5, 3, -6, 7, 9]
l.sort ()
for n in l :
    print n
```

```
l   = [ 4, 5, 3, -6, 7, 9]

for n in sorted (l) :
    print n
```

La fonction `enumerate` permet d'éviter l'emploi de la fonction `range` ou `xrange` lorsqu'on souhaite parcourir une liste alors que l'indice et l'élément sont nécessaires.

```
l   = [ 4, 5, 3, -6, 7, 9]
for i in xrange (0, len (l)) :
    print i, l [i]
```

```
l   = [ 4, 5, 3, -6, 7, 9]
for i,v in enumerate (l) :
    print i, v
```

| | |
|---|---|
| `abs(x)` | Retourne la valeur absolue de `x`. |
| `callable(x)` | Dit si la variable `x` peut être appelée. |
| `chr(i)` | Retourne le caractère associé au code numérique `i`. |
| `cmp(x,y)` | Compare `x` et `y`, retourne -1 si `x` < `y`, 0 en cas d'égalité, 1 sinon. |
| `dir(x)` | Retourne l'ensemble des méthodes associées à `x` qui peut être un objet, un module, un variable,... |
| `enumerate(x)` | parcourt un ensemble itérable (voir paragraphe 3.6) |
| `help(x)` | Retourne l'aide associée à `x`. |
| `id(x)` | Retourne un identifiant unique associé à l'objet `x`. Le mot-clé `is` est relié à cet identifiant. |
| `isinstance(x,classe` | Dit si l'objet `x` est de type `classe`, voir le chapitre 4 sur les classes. |
| `issubclass(cl1,cl2)` | Dit si la classe `cl1` hérite de la classe `cl2`, voir le chapitre 4 sur les classes. |
| `len(l)` | Retourne la longueur de `l`. |
| `map(f,l1,l2,...)` | Applique la fonction `f` sur les listes `l1`, `l2`... Voir l'exemple page 86. |
| `max(l)` | Retourne le plus grand élément de `l`. |
| `min(l)` | Retourne le plus petit élément de `l`. |
| `ord(s)` | Fonction réciproque de `chr`. |
| `range(i,j[,k]` | Construit la liste des entiers de `i` à `j`. Si `k` est précisé, va de `k` en `k` à partir de `i`. |
| `reload(module)` | Recharge un module (voir paragraphe 6). |
| `repr(o)` | Retourne une chaîne de caractères qui représente l'objet `o`. |
| `round(x[,n])` | Arrondi `x` à `n` décimales près ou aucune si `n` n'est pas précisé. |
| `sorted(x [,cmp[,key[, reverse]]])` | tri un ensemble itérable (voir paragraphe 3.6) |
| `str(o)` | Retourne une chaîne de caractères qui représente l'objet `o`. |
| `sum(l)` | Retourne la somme de l'ensemble `l`. |
| `type(o)` | Retourne le type de la variable `o`. |
| `xrange(i,j[,k]` | Plus rapide que la fonction `range` mais utilisable que dans une boucle `for`. |
| `zip(l1,l2,...)` | Construit une liste de tuples au lieu d'un tuple de listes. |

**Table 3.4 :** *Liste non exhaustive de fonctions définis par le langage Python sans qu'aucune extension ne soit nécessaire. Elles ne sont pas toujours applicables même si la syntaxe d'appel est correcte, elles produisent une erreur dans ce cas.*

## 3.7 Constructions classiques

Les paragraphes qui suivent décrivent des schémas qu'on retrouve dans les programmes dans de nombreuses situations. Ce sont des combinaisons simples d'une ou deux boucles, d'un test, d'une liste, d'un dictionnaire.

### 3.7.1 Recherche d'un élément

Rechercher un élément consiste à parcourir un ensemble jusqu'à ce qu'on le trouve : le résultat souhaité est sa position ou un résultat indiquant qu'il n'a pas été trouvé. Cette recherche est souvent insérée dans une fonction dont voici le schéma :

```
def recherche (li, c) :
    for i,v in enumerate (li) :
        if v == c : return i
    return -1
li = [ 45, 32, 43, 56 ]
print recherche (li, 43)    # affiche 2
```

La méthode `index` permet de réduire ce programme.

```
print li.index (43)
```

Néanmoins, la fonction `recherche` peut être adaptée pour rechercher la première valeur trouvée parmi deux possibles, ce qu'il n'est pas possible de faire avec la méthode `index`.

```
def recherche (li, c,d) :
    for i,v in enumerate (li) :
        if v in [c,d] : return i
    return -1
li = [ 45, 32, 43, 56 ]
print recherche (li, 43, 32)    # affiche 1
```

Ce court exemple montre qu'il est utile de connaître quelques algorithmes simples. Ils ne sont pas forcément les plus efficaces mais ils aident à construire des programmes plus rapidement et à obtenir un résultat. S'il est positif alors il sera toujours temps de se pencher sur l'optimisation du programme pour qu'il soit plus efficace.

### 3.7.2 Maximum, minimum

La recherche d'un extremum comme le calcul d'une somme fait toujours intervenir une boucle. La différence est qu'on souhaite parfois obtenir la position du maximum.

```
li = [ 0, 434, 43, 6436, 5 ]
m  = li [0]             # initialisation
for l in li :           # boucle
    if m < l : m = l    # m est le maximum
```

Ce code est équivalent à l'instruction `max(li)`. Pour récupérer la position du maximum, il faut itérer sur les positions et non sur les éléments.

```
li = [ 0, 434, 43, 6436, 5 ]
m  = 0
for i in xrange (0, len (li)) :
    if li [m] < li [i] : m = i
```

Pour éviter la boucle, on peut utiliser l'astuce décrite au paragraphe 3.7.7 qui consiste à former une liste de couples avec la position initiale :

```
k = [ (v,i) for i,v in enumerate (li) ]
m = max (k) [1]
```

Il arrive fréquemment qu'on ne doive pas chercher le maximum d'une liste mais le maximum au sein d'un sous-ensemble de cette liste. Dans l'exemple qui suit, on cherche le tuple dont la première valeur est maximale et dont la seconde valeur est égale à 1.

```
li = [ (0,0), (434,0), (43,1), (6436,1), (5,0) ]
m  = -1 # -1 car le premier élément peut ne pas faire partie du sous-ensemble
for i in range (0, len (li)) :
    if li [i][1] == 0 and (m == -1 or li [m][0] < li [i][0]) : m = i
```

### 3.7.3 Recherche dichotomique

La recherche dichotomique est plus rapide qu'une recherche classique mais elle suppose que celle-ci s'effectue dans un ensemble trié. L'idée est de couper en deux l'intervalle de recherche à chaque itération. Comme l'ensemble est trié, en comparant l'élément cherché à l'élément central, on peut éliminer une partie de l'ensemble : la moitié inférieure ou supérieure.

```
def recherche_dichotomique (li, c) :
    a,b = 0, len (li)-1
    while a <= b :
        m = (a+b)/2
        if   c == li [m] : return m
        elif c <  li [m] : b = m-1   # partie supérieure éliminée
        else             : a = m+1   # partie inférieure éliminée
    return -1  # élément non trouvé

li = range (0,100,2)
print recherche_dichotomique (li, 48)  # affiche 24
print recherche_dichotomique (li, 49)  # affiche -1
```

### 3.7.4 Décomposition en matrice

Les quelques lignes qui suivent permettent de décomposer une chaîne de caractères en matrice. Chaque ligne et chaque colonne sont séparées par des séparateurs différents. Ce procédé intervient souvent lorsqu'on récupère des informations depuis un fichier texte lui-même provenant d'un tableur.

```
s       = "case11;case12;case13|case21;case22;case23"
# décomposition en matrice
ligne   = s.split ("|")                     # lignes
mat     = [ l.split (";") for l in ligne ]  # colonnes
```

L'opération inverse :

```
ligne   = [ ";".join (l) for l in mat ]     # colonnes
s       = "|".join (ligne)                  # lignes
```

### 3.7.5 Somme

Le calcul d'une somme fait toujours intervenir une boucle car le langage *Python* ne peut faire des additions qu'avec deux nombres. Le schéma est toujours le même : initialisation et boucle.

```
li = [ 0, 434, 43, 6456 ]
s  = 0                      # initialisation
for l in li :               # boucle
    s += l                  # addition
```

Ce code est équivalent à l'instruction `sum(li)`. Dans ce cas où la somme intègre le résultat d'une fonction et non les éléments d'une liste, il faudrait écrire :

```
def fonction (x) : return x*x
s  = 0
for l in li : s += fonction (l)
```

Et ces deux lignes pourraient être résumées en une seule grâce à l'une de ces deux instructions.

```
s = sum ( [fonction (l) for l in li] )

s = sum ( map (fonction, li) )
```

L'avantage de la solution avec la boucle est qu'elle évite la création d'une liste intermédiaire, c'est un point à prendre en compte si la liste sur laquelle opère la somme est volumineuse. Il peut être intéressant de regarder l'exercice 12.1.9 page 287.

### 3.7.6 Tri

Le tri est une opération fréquente. On n'a pas toujours le temps de programmer le tri le plus efficace comme un tri *quicksort* et un tri plus simple suffit la plupart du temps. Le tri suivant consiste à recherche le plus petit élément puis à échanger sa place avec le premier élément du tableau du tableau. On recommence la même procédure à partir de la seconde position, puis la troisième et ainsi de suite jusqu'à la fin du tableau.

```
for i in xrange (0, len (li)) :
    # recherche du minimum entre i et len (li) exclu
    pos = i
    for j in xrange (i+1, len (li)) :
        if li [j] < li [pos] : pos = j
    # échange
    ech      = li [pos]
    li [pos] = li [i]
    li [i]   = ech
```

La méthode `sort` tri également une liste mais selon un algorithme plus efficace que celui-ci dont la logique est présentée par l'exercice 4, page 261. Ceci explique pourquoi on hésite toujours à programmer un tri quitte à avoir recours à une astuce telle que celle présentées au paragraphe suivant.

### 3.7.7 Tri et position initiale

Il arrive parfois qu'on souhaite trier un tableau puis appliquer la même transformation à un second tableau. Pour cela, il faut récupérer la position initiale des éléments dans le tableau trié. Une possibilité consiste à trier non pas le tableau mais une liste de couples (valeur, position) comme le montre l'exemple suivant :

```
tab = ["zéro", "un", "deux"]                        # tableau à trier
pos = [ (tab [i],i) for i in range (0, len (tab)) ] # tableau de couples
pos.sort ()                                         # tri
print pos               # affiche [('deux', 2), ('un', 1), ('zéro', 0)]
```

La liste [p[1] `for p in pos`] correspond à l'ensemble des positions initiales des éléments de la liste `tab`. Les deux lignes qui suivent permettent d'en déduire la position de chaque élément dans l'ensemble trié. `ordre`[`i`] est le rang de l'élément d'indice `i` avant le tri.

```
ordre = range (0, len (pos))
for i in xrange (0, len (pos)) : ordre [pos [i][1]] = i
```

### 3.7.8 Comptage

On souhaite ici compter le nombre d'occurrences de chaque élément d'un tableau. Par exemple, on pourrait connaître par ce moyen la popularité d'un mot dans un discours politique ou l'étendue du vocabulaire utilisé. L'exemple suivant compte les mots d'une liste de mots.

```
li = ["un", "deux", "un", "trois"]
d  = { }
for l in li :
    if l not in d : d [l] = 1
    else : d [l] += 1
print d   # affiche {'un': 2, 'trois': 1, 'deux': 1}
```

La structure la plus appropriée ici est un dictionnaire puisqu'on cherche à associer une valeur à un élément d'une liste qui peut être de tout type. Si la liste contient des éléments de type modifiable comme une liste, il faudrait convertir ceux-ci en un type immuable comme une chaîne de caractères. L'exemple suivant illustre ce cas en comptant les occurrences des lignes d'une matrice.

```
mat = [ [1,1,1], [2,2,2], [1,1,1]]
d  = { }
for l in mat :
    k = str (l)    # k = tuple (l) lorsque cela est possible
    if k not in d : d [k] = 1
    else : d [k] += 1
print d   # affiche {'[1, 1, 1]': 2, '[2, 2, 2]': 1}
```

On peut également vouloir non pas compter le nombre d'occurrence mais mémoriser les positions des éléments tous identiques. On doit utiliser un dictionnaire de listes :

```
li = ["un", "deux", "un", "trois"]
d  = { }
for i,v in enumerate (li) :
    if v not in d : d [v] = [ i ]
    else : d [v].append (i)
print d   # affiche {'un': [0, 2], 'trois': [3], 'deux': [1]}
```

### 3.7.9 Matrice en un vecteur

Dans un langage comme le *C++*, il arrive fréquemment qu'une matrice ne soit pas représentée par une liste de listes mais par une seule liste car cette représentation est plus efficace. Il faut donc convertir un indice en deux indices ligne et colonne. Il faut bien sûr que le nombre de colonnes sur chaque ligne soit constant. Le premier programme convertit une liste de listes en une seule liste.

```
mat = [[0,1,2],[3,4,5]]
lin = [ i * len (mat [i]) + j \
            for i in range (0, len (mat)) \
            for j in range (0, len (mat [i])) ]
```

Le programme suivant fait l'inverse. Il faut faire attention à la position des crochets et l'ordre des boucles `for`.

```
nc = len (mat [0])
mat = [ [ lin [i * nc + j] for j in range (0, len (mat [i])) ] \
                           for i in range (0, len (mat)) ]
```

### 3.7.10 Fonctions et variables

Une fonction peut aussi recevoir en paramètre une autre fonction. L'exemple suivant inclut la fonction `calcul_n_valeur` qui prend comme paramètres `l` et `f`. Cette fonction calcule pour toutes les valeurs `x` de la liste `l` la valeur `f(x)`. `fonction_carre` ou `fonction_cube` sont passées en paramètres à la fonction `calcul_n_valeur` qui les exécute. Le paragraphe 4.10.1 page 134 présente le même programme mais rédigé avec des classes.

```
def fonction_carre(x) :  return x*x
def fonction_cube (x) :  return x*x*x

def calcul_n_valeur (l,f):
    res = [ f(i) for i in l ]
    return res

l = [0,1,2,3]
print l   # affiche [0, 1, 2, 3]

l1 = calcul_n_valeur (l, fonction_carre)
print l1  # affiche [0, 1, 4, 9]

l2 = calcul_n_valeur (l, fonction_cube)
print l2  # affiche [0, 1, 8, 27]
```

# Chapitre 4

# Classes

---

Imaginons qu'une banque détienne un fichier contenant des informations sur ses clients et qu'il soit impossible pour un client d'avoir accès directement à ces informations. Toutefois, il lui est en théorie possible de demander à son banquier quelles sont les informations le concernant détenues par sa banque. Il est en théorie également possible de les rectifier s'il estime qu'elles sont incorrectes. .

On peut comparer cette banque à un objet qui possède des informations et des moyens permettant de lire ces informations et de les modifier. Vu de l'extérieur, cette banque cache son fonctionnement interne et les informations dont elle dispose, mais propose des services à ses utilisateurs.

On peut considérer la banque comme un *objet* au sens informatique. Ce terme désigne une entité possédant des données et des méthodes permettant de les manipuler. Plus concrètement, une classe est un assemblage de variables appelées *attributs* et de fonctions appelées *méthodes*. L'ensemble des propriétés associées aux classes est regroupé sous la désignation de *programmation objet*.

## 4.1 Présentation des classes : méthodes et attributs

### 4.1.1 Définition, déclaration

**Définition 4.1 : classe**
Une classe est un ensemble incluant des variables ou *attributs* et des fonctions ou *méthodes*. Les attributs sont des variables accessibles depuis toute méthode de la classe où elles sont définies. En *Python*, les classes sont des types modifiables.

**syntaxe 4.2 : classe, définition**

```
class nom_classe :
    # corps de la classe
    # ...
```

Le corps d'une classe peut être vide, inclure des variables ou attributs, des fonctions ou méthodes. Il est en tout cas indenté de façon à indiquer à l'interpréteur *Python* les lignes qui forment le corps de la classe.

Les classes sont l'unique moyen en langage *Python* de définir de nouveaux types propres à celui qui programme. Il n'existe pas de type "matrice" ou de type "graphe" en langage *Python* qui soit prédéfini [1]. Il est néanmoins possible de les définir au

1. Les matrices sont définies dans une extension **externe** `numpy` (voir le chapitre 6).

moyen des classes. Une matrice est par exemple un objet qui inclut les attributs suivant : le nombre de lignes, le nombre de colonnes, les coefficients de la matrice. Cette matrice inclut aussi des méthodes comme des opérations entre deux matrices telles que l'addition, la soustraction, la multiplication ou des opérations sur elle-même comme l'inversion, la transposition, la diagonalisation.

Cette liste n'est pas exhaustive, elle illustre ce que peut être une classe "matrice" - représentation informatique d'un objet "matrice" -, un type complexe incluant des informations de types variés (entier pour les dimensions, réels pour les coefficients), et des méthodes propres à cet objet, capables de manipuler ces informations.

Il est tout-à-fait possible de se passer des classes pour rédiger un programme informatique. Leur utilisation améliore néanmoins sa présentation et la compréhension qu'on peut en avoir. Bien souvent, ceux qui passent d'un langage uniquement fonctionnel à un langage objet ne font pas marche arrière.

**syntaxe 4.3 : classe, création d'une variable de type objet**

```
cl = nom_classe ()
```

La création d'une variable de type objet est identique à celle des types standards du langage *Python* : elle passe par une simple affectation. On appelle aussi `cl` une *instance* de la classe `nom_classe`.

Cette syntaxe est identique à la syntaxe d'appel d'une fonction. La création d'une instance peut également faire intervenir des paramètres (voir paragraphe 4.4). Le terme *instance* va de paire avec le terme *classe* :

**Définition 4.4 : instance**
Une instance d'une classe `C` désigne une variable de type `C`. Le terme instance ne s'applique qu'aux variables dont le type est une classe.

L'exemple suivant permet de définir une classe vide. Le mot-clé `pass` permet de préciser que le corps de la classe ne contient rien.

```
class classe_vide:
    pass
```

Il est tout de même possible de définir une instance de la classe `classe_vide` simplement par l'instruction suivante :

```
class classe_vide:
    pass
cl = classe_vide ()
```

**Remarque 4.5 : type d'une instance**
Dans l'exemple précédent, la variable `nb` n'est pas de type `exemple_classe` mais de type `instance` comme le montre la ligne suivante :

```
print type (cl)   # affiche <type 'instance'>
```

Pour savoir si une variable est une instance d'une classe donnée, il faut utiliser la fonction `isinstance` :

```
isinstance (cl,classe_vide)   # affiche True
```

### 4.1.2 Méthodes

**Définition 4.6 : méthode**
Les méthodes sont des fonctions qui sont associées de manière explicite à une classe. Elles ont comme particularité un accès privilégié aux données de la classe elle-même.

Ces données ou *attributs* sont définis plus loin. Les méthodes sont en fait des fonctions pour lesquelles la liste des paramètres contient obligatoirement un paramètre explicite qui est l'instance de la classe à laquelle cette méthode est associée. Ce paramètre est le moyen d'accéder aux données de la classe.

**syntaxe 4.7 : classe, méthode, définition**

```
class nom_classe :
    def nom_methode (self, param_1, ..., param_n) :
        # corps de la méthode...
```

A part le premier paramètre qui doit de préférence s'appeler `self`, la syntaxe de définition d'une méthode ressemble en tout point à celle d'une fonction. Le corps de la méthode est indenté par rapport à la déclaration de la méthode, elle-même indentée par rapport à la déclaration de la classe.

L'appel à cette méthode obéit à la syntaxe qui suit :

**syntaxe 4.8 : classe, méthode, appel**

```
cl = nom_classe ()    # variable de type nom_classe
t  = cl.nom_methode (valeur_1, ..., valeur_n)
```

L'appel d'une méthode nécessite tout d'abord la création d'une variable. Une fois cette variable créée, il suffit d'ajouter le symbole "." pour exécuter la méthode. Le paramètre `self` est ici implicitement remplacé par `cl` lors de l'appel.

L'exemple suivant simule le tirage de nombres aléatoires à partir d'une suite définie par récurrence $u_{n+1} = (u_n * A) \ mod \ B$ où $A$ et $B$ sont des entiers très grands. Cette suite n'est pas aléatoire mais son comportement imite celui d'une suite aléatoire. Le terme $u_n$ est dans cet exemple contenu dans la variable globale `rnd`.

```
rnd = 42

class exemple_classe:
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        global rnd
        rnd = 397204094 * rnd % 2147483647
        return int (rnd % n)

nb  = exemple_classe ()
l   = [ nb.methode1(100) for i in range(0,10) ]
print l   # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

```
nb2 = exemple_classe ()
l2  = [ nb2.methode1(100) for i in range(0,10) ]
print l2   # affiche [46, 42, 89, 66, 48, 12, 61, 84, 71, 41]
```

Deux instances `nb` et `nb2` de la classe `exemple_classe` sont créées, chacune d'elles est utilisée pour générer aléatoirement dix nombres entiers compris entre 0 et 99 inclus. Les deux listes sont différentes puisque l'instance `nb2` utilise la variable globale `rnd` précédemment modifiée par l'appel `nb.methode1(100)`.

**Remarque 4.9 : méthodes et fonctions**
Les méthodes sont des fonctions insérées à l'intérieur d'une classe. La syntaxe de la déclaration d'une méthode est identique à celle d'une fonction en tenant compte du premier paramètre qui doit impérativement être `self`. Les paramètres par défaut, l'ordre des paramètres, les nombres variables de paramètres présentés au paragraphe 3.4 sont des extensions tout autant applicables aux méthodes qu'aux fonctions.

### 4.1.3 Attributs

**Définition 4.10 : attribut**
Les attributs sont des variables qui sont associées de manière explicite à une classe. Les attributs de la classe se comportent comme des variables globales pour toutes les méthodes de cette classe.

Une classe permet en quelque sorte de regrouper ensemble des informations liées. Elles n'ont de sens qu'ensemble et les méthodes manipulent ces données liées. C'est le cas pour un segment qui est toujours défini par ces deux extrémités qui ne vont pas l'une sans l'autre.

**syntaxe 4.11 : classe, attribut, définition**

```
class nom_classe :
    def nom_methode (self, param_1, ..., param_n) :
        self.nom_attribut = valeur
```

Le paramètre `self` n'est pas un mot-clé même si le premier paramètre est le plus souvent appelé `self`. Il désigne l'instance de la classe sur laquelle va s'appliquer la méthode. La déclaration d'une méthode inclut toujours un paramètre `self` de sorte que `self.nom_attribut` désigne un attribut de la classe. `nom_attribut` seul désignerait une variable locale sans aucun rapport avec un attribut portant le même nom. Les attributs peuvent être déclarés à l'intérieur de n'importe quelle méthode, voire à l'extérieur de la classe elle-même.

L'endroit où est déclaré un attribut a peu d'importance pourvu qu'il le soit avant sa première utilisation. Dans l'exemple qui suit, la méthode `methode1` utilise l'attribut `rnd` sans qu'il ait été créé.

```
class exemple_classe:
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
```

```
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb = exemple_classe ()
l  = [ nb.methode1(100) for i in range(0,10) ]
print l
```

Cet exemple déclenche donc une erreur (ou exception) signifiant que l'attribut `rnd` n'a pas été créé.

```
Traceback (most recent call last):
  File "cours.py", line 8, in -toplevel-
    l = [ nb.methode1(100) for i in range(0,10) ]
  File "cours.py", line 4, in methode1
    self.rnd = 397204094 * self.rnd % 2147483647
AttributeError: exemple_classe instance has no attribute 'rnd'
```

Pour remédier à ce problème, il existe plusieurs endroits où il est possible de créer l'attribut `rnd`. Il est possible de créer l'attribut à l'intérieur de la méthode `methode1`. Mais le programme n'a plus le même sens puisqu'à chaque appel de la méthode `methode1`, l'attribut `rnd` reçoit la valeur 42. La liste de nombres aléatoires contient dix fois la même valeur.

```
class exemple_classe:
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        self.rnd = 42  # déclaration à l'intérieur de la méthode,
                       # doit être précédé du mot-clé self
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb = exemple_classe ()
l  = [ nb.methode1(100) for i in range(0,10) ]
print l  # affiche [19, 19, 19, 19, 19, 19, 19, 19, 19, 19]
```

Il est possible de créer l'attribut `rnd` à l'extérieur de la classe. Cette écriture devrait toutefois être évitée puisque la méthode `methode1` ne peut pas être appelée sans que l'attribut `rnd` ait été ajouté.

```
class exemple_classe:
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire
           compris entre 0 et n-1 inclus"""
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb     = exemple_classe ()
nb.rnd = 42              # déclaration à l'extérieur de la classe,
                         # indispensable pour utiliser la méthode methode1
l = [ nb.methode1(100) for i in range(0,10) ]
print l  # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

## 4.2 Constructeur

L'endroit le plus approprié pour déclarer un attribut est à l'intérieur d'une méthode appelée le *constructeur*. S'il est défini, il est implicitement exécuté lors de la création de chaque instance. Le constructeur d'une classe se présente comme une méthode et suit la même syntaxe à ceci près que son nom est imposé : `__init__`. Hormis le premier paramètre, invariablement `self`, il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

**syntaxe 4.12 : classe, constructeur**

```
class nom_classe :
    def __init__(self, param_1, ..., param_n):
        # code du constructeur
```

`nom_classe` est une classe, `__init__` est son constructeur, sa syntaxe est la même que celle d'une méthode sauf que le constructeur ne peut employer l'instruction `return`.

La modification des paramètres du constructeur implique également la modification de la syntaxe de création d'une instance de cette classe.

**syntaxe 4.13 : classe, instance**

```
x = nom_classe (valeur_1,...,valeur_n)
```

`nom_classe` est une classe, `valeur_1` à `valeur_n` sont les valeurs associées aux paramètres `param_1` à `param_n` du constructeur.

L'exemple suivant montre deux classes pour lesquelles un constructeur a été défini. La première n'ajoute aucun paramètre, la création d'une instance ne nécessite pas de paramètre supplémentaire. La seconde classe ajoute deux paramètres `a` et `b`. Lors de la création d'une instance de la classe `classe2`, il faut ajouter deux valeurs.

```
class classe1:
    def __init__(self):
        # pas de paramètre supplémentaire
        print "constructeur de la classe classe1"
        self.n = 1 # ajout de l'attribut n

x = classe1 ()     # affiche constructeur de la classe classe1
print x.n          # affiche 1

class classe2:
    def __init__(self,a,b):
        # deux paramètres supplémentaires
        print "constructeur de la classe classe2"
        self.n = (a+b)/2  # ajout de l'attribut n

x = classe2 (5,9)  # affiche constructeur de la classe classe2
print x.n          # affiche 7
```

**Remarque 4.14 : constructeur et fonction**
Le constructeur autorise autant de paramètres qu'on souhaite lors de la création

d'une instance et celle-ci suit la même syntaxe qu'une fonction. La création d'une instance pourrait être considérée comme l'appel à une fonction à ceci près que le type du résultat est une instance de classe.

En utilisant un constructeur, l'exemple du paragraphe précédent simulant une suite de variable aléatoire permet d'obtenir une classe autonome qui ne fait pas appel à une variable globale ni à une déclaration d'attribut extérieur à la classe.

```
class exemple_classe:
    def __init__ (self) : # constructeur
        self.rnd = 42     # on crée l'attribut rnd, identique pour chaque instance
                          # --> les suites générées auront toutes le même début
    def methode1(self,n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb  = exemple_classe ()
l   = [ nb.methode1(100) for i in range(0,10) ]
print l   # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]

nb2 = exemple_classe ()
l2  = [ nb2.methode1(100) for i in range(0,10) ]
print l2   # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

De la même manière qu'il existe un constructeur exécuté à chaque création d'instance, il existe un destructeur exécuté à chaque destruction d'instance. Il suffit pour cela de redéfinir la méthode `__del__`. A l'inverse d'autres langages comme le *C++*, cet opérateur est peu utilisé car le *Python* nettoie automatiquement les objets qui ne sont plus utilisés ou plus référencés par une variable. Un exemple est donné page 115.

## 4.3 Apport du langage *Python*

### 4.3.1 Liste des attributs

Chaque attribut d'une instance de classe est inséré dans un dictionnaire appelé `__dict__`, attribut implicitement présent dès la création d'une instance.

```
class exemple_classe:
    def __init__ (self) :
        self.rnd = 42
    def methode1(self,n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb = exemple_classe ()
print nb.__dict__        # affiche {'rnd': 42}
```

Ce dictionnaire offre aussi la possibilité de tester si un attribut existe ou non. Dans un des exemples du paragraphe précédent, l'attribut `rnd` était créé dans la méthode `methode1`, sa valeur était alors initialisée à chaque appel et la fonction retournait sans cesse la même valeur. En testant l'existence de l'attribut `rnd`, il est possible de le créer dans la méthode `methode1` au premier appel sans que les appels suivants ne réinitialisent sa valeur à 42.

```
class exemple_classe:
    def methode1(self,n):
        if "rnd" not in self.__dict__ :  # l'attribut existe-t-il ?
            self.rnd = 42                # création de l'attribut
            self.__dict__ ["rnd"] = 42   # autre écriture possible
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb = exemple_classe ()
l  = [ nb.methode1(100) for i in range(0,10) ]
print l  # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

### 4.3.2 Attributs implicites

Certains attributs sont créés de manière implicite lors de la création d'une instance. Ils contiennent des informations sur l'instance.

| | |
|---|---|
| `__module__` | Contient le nom du module dans lequel est incluse la classe (voir chapitre 6). |
| `__class__` | Contient le nom de la classe de l'instance. Ce nom est précédé du nom du module suivi d'un point. |
| `__dict__` | Contient la liste des attributs de l'instance (voir paragraphe 4.3.1). |
| `__doc__` | Contient un commentaire associé à la classe (voir paragraphe 4.3.3). |

L'attribut `__class__` contient lui même d'autres d'attributs :

| | |
|---|---|
| `__doc__` | Contient un commentaire associé à la classe (voir paragraphe 4.3.3). |
| `__dict__` | Contient la liste des attributs statiques (définis hors d'une méthode) et des méthodes (voir paragraphe 4.5.2). |
| `__name__` | Contient le nom de l'instance. |
| `__bases__` | Contient les classes dont la classe de l'instance hérite (voir paragraphe 4.8). |

```
class classe_vide:
    pass
cl = classe_vide ()
print cl.__module__             # affiche __main__
print cl.__class__              # affiche __main__.classe_vide ()
print cl.__dict__               # affiche {}
print cl.__doc__                # affiche None  (voir paragraphe suivant)
print cl.__class__.__doc__      # affiche None
print cl.__class__.__dict__     # affiche {'__module__': '__main__',
                                #          '__doc__': None}
print cl.__class__.__name__     # affiche classe_vide
print cl.__class__.__bases__    # affiche ()
```

### 4.3.3 Commentaires, aide

Comme les fonctions et les méthodes, des commentaires peuvent être associés à une classe, ils sont affichés grâce à la fonction `help`. Cette dernière présente le commentaire associé à la classe, la liste des méthodes ainsi que chacun des commentaires qui leur sont associés. Ce commentaire est affecté à l'attribut implicite `__doc__`. L'appel à la fonction `help` rassemble le commentaire de toutes les méthodes, le résultat suit le programme ci-dessous.

```
class exemple_classe:
    """simule une suite de nombres aléatoires"""
    def __init__ (self) :
        """constructeur : initialisation de la première valeur"""
        self.rnd = 42
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire
        compris entre 0 et n-1 inclus"""
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)
nb = exemple_classe ()
help (exemple_classe)     # appelle l'aide associée à la classe
```

```
class exemple_classe
 |  simule une suite de nombres aléatoires
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      constructeur : initialisation de la première valeur
 |
 |  methode1(self, n)
 |      simule la génération d'un nombre aléatoire
 |      compris entre 0 et n-1 inclus
```

Pour obtenir seulement le commentaire associé à la classe, il suffit d'écrire l'une des trois lignes suivantes :

```
print exemple_classe.__doc__  # affiche simule une suite de nombres aléatoires
print nb.__doc__              # affiche simule une suite de nombres aléatoires
print nb.__class__.__doc__    # affiche simule une suite de nombres aléatoires
```

La fonction `help` permet d'accéder à l'aide associée à une fonction, une classe. Il existe des outils qui permettent de collecter tous ces commentaires pour construire une documentation au format *HTML* à l'aide d'outils comme *pydoc*[2] ou *Doxygen*[3]. Ces outils sont souvent assez simples d'utilisation.

La fonction `dir` permet aussi d'obtenir des informations sur la classe. Cette fonction appliquée à la classe ou à une instance retourne l'ensemble de la liste des attributs et des méthodes. L'exemple suivant utilise la fonction `dir` avant et après l'appel de la méthode `meth`. Etant donné que cette méthode ajoute un attribut, la fonction `dir` retourne une liste plus longue après l'appel.

2. voir http://pydoc.org/ ou http://docs.python.org/library/pydoc.html
3. http://www.stack.nl/~dimitri/doxygen/ associé à *doxypy* (http://code.foosel.org/doxypy).

```
class essai_class:
    def meth(self):
        x      = 6
        self.y = 7

a = essai_class()
print dir (a)             # affiche ['__doc__', '__module__', 'meth']
a.meth ()
print dir (a)             # affiche ['__doc__', '__module__', 'meth', 'y']
print dir (essai_class)   # affiche ['__doc__', '__module__', 'meth']
```

La fonction `dir` appliquée à la classe elle-même retourne une liste qui inclut les méthodes et les attributs déjà déclarés. Elle n'inclut pas ceux qui sont déclarés dans une méthode jamais exécutée jusqu'à présent.

### 4.3.4 Classe incluse

Parfois, il arrive qu'une classe soit exclusivement utilisée en couple avec une autre, c'est par exemple le cas des itérateurs (voir paragraphe 4.4.2). Il est alors possible d'inclure dans la déclaration d'une classe celle d'une sous-classe.

L'exemple qui suit contient la classe `ensemble_element`. C'est un ensemble de points en trois dimensions (classe `element`) qui n'est utilisé que par cette classe. Déclarer la classe `element` à l'intérieur de la classe `ensemble_element` est un moyen de signifier ce lien.

```
class ensemble_element :

    class element :
        def __init__ (self) :
            self.x, self.y, self.z = 0,0,0

    def __init__ (self) :
        self.all = [ ensemble_element.element () for i in xrange (0,3) ]

    def barycentre (self) :
        b = ensemble_element.element ()
        for el in self.all :
            b.x += el.x
            b.y += el.y
            b.z += el.z
        b.x /= len (self.all)
        b.y /= len (self.all)
        b.z /= len (self.all)
        return b

f = ensemble_element ()
f.all [0].x, f.all [0].y, f.all [0].z = 4.5,1.5,1.5
b = f.barycentre ()
print b.x,b.y,b.z # affiche 1.5 0.5 0.5
```

Pour créer une instance de la classe `element`, il faut faire précéder son nom de la classe où elle est déclarée : `b = ensemble_element.element()` comme c'est le cas dans la méthode `barycentre` par exemple.

## 4.4 Opérateurs, itérateurs

Les opérateurs sont des symboles du langages comme `+`, `-`, `+=`, ... Au travers des opérateurs, il est possible de donner un sens à une syntaxe comme celle de l'exemple suivant :

```
class nouvelle_classe:
    pass
x = nouvelle_classe () + nouvelle_classe ()
```

L'addition n'est pas le seul symbole concerné, le langage *Python* permet de donner un sens à tous les opérateurs numériques et d'autres reliés à des fonctions du langage comme `len` ou `max`.

### 4.4.1 Opérateurs

Le programme suivant contient une classe définissant un nombre complexe. La méthode `ajoute` définit ce qu'est une addition entre nombres complexes.

```
class nombre_complexe:
    def __init__ (self, a = 0, b= 0) : self.a, self.b = a,b
    def get_module (self) :  return math.sqrt (self.a * self.a + self.b * self.b)

    def ajoute (self,c):
        return nombre_complexe (self.a + c.a, self.b + c.b)

c1 = nombre_complexe (0,1)
c2 = nombre_complexe (1,0)
c  = c1.ajoute (c2)         # c = c1 + c2
print c.a, c.b
```

Toutefois, on aimerait bien écrire simplement `c = c1 + c2` au lieu de `c = c1.ajoute(c2)` car cette syntaxe est plus facile à lire et surtout plus intuitive. Le langage *Python* offre cette possibilité. Il existe en effet des méthodes *clés* dont l'implémentation définit ce qui doit être fait dans le cas d'une addition, d'une comparaison, d'un affichage, ... A l'instar du constructeur, toutes ces méthodes clés, qu'on appelle des *opérateurs*, sont encadrées par deux blancs soulignés, leur déclaration suit invariablement le même schéma. Voici celui de l'opérateur `__add__` qui décrit ce qu'il faut faire pour une addition.

**syntaxe 4.15 : classe, opérateur `__add__`**

```
class nom_class :
    def __add__ (self, autre) :
        # corps de l'opérateur
        return ...   # nom_classe
```

`nom_classe` est une classe. L'opérateur `__add__` définit l'addition entre l'instance `self` et l'instance `autre` et retourne une instance de la classe `nom_classe`.

Le programme suivant reprend le précédent de manière à ce que l'addition de deux nombres complexes soit dorénavant une syntaxe correcte.

```
class nombre_complexe:
    def __init__ (self, a = 0, b= 0) : self.a, self.b = a,b
    def get_module (self) : return math.sqrt (self.a * self.a + self.b * self.b)

    def __add__(self, c):
        return nombre_complexe (self.a + c.a, self.b + c.b)

c1 = nombre_complexe (0,1)
c2 = nombre_complexe (1,0)
c  = c1 + c2          # cette expression est maintenant syntaxiquement correcte
c  = c1.__add__ (c2)  # même ligne que la précédente mais écrite explicitement
print c.a, c.b
```

L'avant dernière ligne appelant la méthode `__add__` transcrit de façon explicite ce que le langage *Python* fait lorsqu'il rencontre un opérateur + qui s'applique à des classes. Plus précisément, `c1` et `c2` pourraient être de classes différentes, l'expression serait encore valide du moment que la classe dont dépend `c1` a redéfini la méthode `__add__`. Chaque opérateur possède sa méthode-clé associée. L'opérateur `+=`, différent de + est associé à la méthode-clé `__iadd__`.

**syntaxe 4.16 : classe, opérateur `__iadd__`**

```
class nom_class :
    def __iadd__ (self, autre) :
        # corps de l'opérateur
        return self
```

`nom_classe` est une classe. L'opérateur `__iadd__` définit l'addition entre l'instance `self` et l'instance `autre`. L'instance `self` est modifiée pour recevoir le résultat. L'opérateur retourne invariablement l'instance modifiée `self`.

On étoffe la classe `nombre_complexe` à l'aide de l'opérateur `__iadd__`.

```
class nombre_complexe:
    def __init__ (self, a = 0, b= 0) : self.a, self.b = a,b
    def get_module (self) : return math.sqrt (self.a * self.a + self.b * self.b)
    def __add__(self, c)  : return nombre_complexe (self.a + c.a, self.b + c.b)

    def __iadd__(self, c) :
        self.a += c.a
        self.b += c.b
        return self

c1  = nombre_complexe (0,1)
c2  = nombre_complexe (1,0)
c1 += c2           # utilisation de l'opérateur +=
c1.__iadd__ (c2)   # c'est la transcription explicite de la ligne précédente
print c1.a, c1.b
```

Un autre opérateur souvent utilisé est `__str__` qui permet de redéfinir l'affichage d'un objet lors d'un appel à l'instruction `print`.

**syntaxe 4.17 : classe, opérateur `__str__`**

```
class nom_class :
    def __str__ (self) :
        # corps de l'opérateur
        return...
```

`nom_classe` est une classe. L'opérateur `__str__` construit une chaîne de caractères qu'il retourne comme résultat de façon à être affiché.

L'exemple suivant reprend la classe `nombre_complexe` pour que l'instruction `print` affiche un nombre complexe sous la forme $a + ib$.

```
class nombre_complexe:
    def __init__ (self, a = 0, b= 0) : self.a, self.b = a,b
    def __add__(self, c) : return nombre_complexe (self.a + c.a, self.b + c.b)

    def __str__ (self) :
        if   self.b == 0 : return "%f" % (self.a)
        elif self.b >  0 : return "%f + %f i" % (self.a, self.b)
        else             : return "%f - %f i" % (self.a, -self.b)

c1 = nombre_complexe (0,1)
c2 = nombre_complexe (1,0)
c3 = c1 + c2
print c3       # affiche 1.000000 + 1.000000 i
```

Il existe de nombreux opérateurs qu'il est possible de définir. La table 4.1 (page 137) présente les plus utilisés. Parmi ceux-là, on peut s'attarder sur les opérateurs `__getitem__` et `__setitem__`, ils redéfinissent l'opérateur `[]` permettant d'accéder à un élément d'une liste ou d'un dictionnaire. Le premier, `__getitem__` est utilisé lors d'un calcul, un affichage. Le second, `__setitem__`, est utilisé pour affecter une valeur.

L'exemple suivant définit un point de l'espace avec trois coordonnées. Il redéfinit ou *surcharge* les opérateurs `__getitem__` et `__setitem__` de manière à pouvoir accéder aux coordonnées de la classe `point_espace` qui définit un point dans l'espace. En règle générale, lorsque les indices ne sont pas corrects, ces deux opérateurs lèvent l'exception `IndexError` (voir le chapitre 5).

```
class point_espace:
    def __init__ (self, x,y,z): self._x, self._y, self._z = x,y,z

    def __getitem__(self,i):
        if i == 0 : return self._x
        if i == 1 : return self._y
        if i == 2 : return self._z
        # pour tous les autres cas --> erreur
        raise IndexError ("indice impossible, 0,1,2 autorisés")

    def __setitem__(self,i,x):
        if   i == 0 : self._x = x
        elif i == 1 : self._y = y
        elif i == 2 : self._z = z
        # pour tous les autres cas --> erreur
        raise IndexError ("indice impossible, 0,1,2 autorisés")
```

```
    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)

a = point_espace (1,-2,3)

print a                      # affiche (1.000000,-2.000000,3.000000)
a [1] = -3                   # (__setitem__) affecte -3 à a.y
print "abscisse : ", a [0]   # (__getitem__) affiche abscisse :  1
print "ordonnée : ", a [1]   # (__getitem__) affiche ordonnée :  -3
print "altitude : ", a [2]   # (__getitem__) affiche altitude :  3
```

Par le biais de l'exception `IndexError`, les expressions `a[i]` avec $\texttt{i} \neq 0, 1, 2$ sont impossibles et arrêtent le programme par un message comme celui qui suit obtenu après l'interprétation de `print a [4]` :

```
Traceback (most recent call last):
  File "point_espace.py", line 31, in ?
    print a [4]
  File "point_espace.py", line 13, in __getitem__
    raise IndexError, "indice impossible, 0,1,2 autorisés"
IndexError: indice impossible, 0,1,2 autorisés
```

### 4.4.2 Itérateurs

L'opérateur `__iter__` permet de définir ce qu'on appelle un itérateur. C'est un objet qui permet d'en explorer un autre, comme une liste ou un dictionnaire. Un itérateur est un objet qui désigne un élément d'un ensemble à parcourir et qui connaît l'élément suivant à visiter. Il doit pour cela contenir une référence à l'objet qu'il doit explorer et inclure une méthode `next` qui retourne l'élément suivant ou lève une exception si l'élément actuel est le dernier.

Par exemple, on cherche à explorer tous les éléments d'un objet de type `point_espace` défini au paragraphe précédent. Cette exploration doit s'effectuer au moyen d'une boucle `for`.

```
a = point_espace (1,-2,3)
for x in a:
    print x      # affiche successivement 1,-2,3
```

Cette boucle cache en fait l'utilisation d'un itérateur qui apparaît explicitement dans l'exemple suivant équivalent au précédent (voir paragraphe 3.3.2.3, page 66).

```
a = point_espace (1,-2,3)
it = iter (a)
while True:
    try : print it.next ()
    except StopIteration : break
```

Afin que cet extrait de programme fonctionne, il faut définir un itérateur pour la classe `point_espace`. Cet itérateur doit inclure la méthode `next`. La classe `point_espace` doit quant à elle définir l'opérateur `__iter__` pour retourner l'itérateur qui permettra de l'explorer.

```
class point_espace:
    def __init__ (self, x,y,z):
        self._x, self._y, self._z = x,y,z
    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)
    def __getitem__(self,i):
        if i == 0 : return self._x
        if i == 1 : return self._y
        if i == 2 : return self._z
        # pour tous les autres cas --> erreur
        raise IndexError ("indice impossible, 0,1,2 autorisés")

    class class_iter:
        """cette classe définit un itérateur pour point_espace"""
        def __init__ (self,ins):
            """initialisation, self._ins permet de savoir quelle
               instance de point_espace on explore,
               self._n mémorise l'indice de l'élément exploré"""
            self._n   = 0
            self._ins = ins

        def __iter__ (self) :   # le langage impose cette méthode
           return self          # dans certaines configurations

        def next (self):
            """retourne l'élément d'indice self._n et passe à l'élément suivant"""
            if self._n <= 2:
                v = self._ins [self._n]
                self._n += 1
                return v
            else :
                # si cet élément n'existe pas, lêve une exception
                raise StopIteration

    def __iter__(self):
        """opérateur de la classe point_espace, retourne un itérateur
           permettant de l'explorer"""
        return point_espace.class_iter (self)

a = point_espace (1,-2,3)
for x in a:
    print x      # affiche successivement 1,-2,3
```

Cette syntaxe peut paraître fastidieuse mais elle montre de manière explicite le fonctionnement des itérateurs. Cette construction est plus proche de ce que d'autres langages objets proposent. *Python* offre néanmoins une syntaxe plus courte avec le mot-clé `yield` qui permet d'éviter la création de la classe `class_iter`. Le code de la méthode `__iter__` change mais les dernières lignes du programme précédent qui affichent successivement les éléments de `point_espace` sont toujours valides.

```
class point_espace:
    def __init__ (self, x,y,z):
        self._x, self._y, self._z = x,y,z
    def __str__(self):
        return "(%f,%f,%f)" % (self._x, self._y, self._z)
    def __getitem__(self,i):
        if i == 0 : return self._x
```

```
        if i == 1 : return self._y
        if i == 2 : return self._z
        # pour tous les autres cas --> erreur
        raise IndexError ("indice impossible, 0,1,2 autorisés")

    def __iter__(self):
        """itérateur avec yield (ou générateur)"""
        _n = 0
        while _n <= 2 :
            yield self.__getitem__ (_n)
            _n += 1

a = point_espace (1,-2,3)
for x in a:
    print x      # affiche successivement 1,-2,3
```

## 4.5 Méthodes, attributs statiques et ajout de méthodes

### 4.5.1 Méthode statique

**Définition 4.18 : méthode statique**
Les méthodes statiques sont des méthodes qui peuvent être appelées même si aucune instance de la classe où elles sont définies n'a été créée.

L'exemple suivant définit une classe avec une seule méthode. Comme toutes les méthodes présentées jusqu'à présent, elle inclut le paramètre `self` qui correspond à l'instance pour laquelle elle est appelée.

```
class essai_class:
    def methode (self):
        print "méthode non statique"

x = essai_class ()
x.methode ()
```

Une méthode statique ne nécessite pas qu'une instance soit créée pour être appelée. C'est donc une méthode n'ayant pas besoin du paramètre `self`.

**syntaxe 4.19 : classe, méthode statique**

```
class nom_class :
    def nom_methode(params, ...) :
        # corps de la méthode
        ...
    nom_methode = staticmethod (nom_methode)
```

`nom_classe` est une classe, `nom_methode` est une méthode statique. Il faut pourtant ajouter la ligne suivante pour indiquer à la classe que cette méthode est bien statique à l'aide du mot-clé `staticmethod`.

Le programme précédent est modifié pour inclure une méthode statique. La méthode `methode` ne nécessite aucune création d'instance pour être appelée.

```
class essai_class:
    def methode ():
        print "méthode statique"
    methode = staticmethod(methode)

essai_class.methode ()
```

**Remarque 4.20 : déclaration d'une méthode statique**
Il est également possible de déclarer une fonction statique à l'extérieur d'une classe puis de l'ajouter en tant que méthode statique à cette classe. Le programme suivant déclare une fonction `methode` puis indique à la classe `essai_class` que la fonction est aussi une méthode statique de sa classe (avant-dernière ligne de l'exemple).

```
def methode ():
    print "méthode statique"

class essai_class:
    pass

essai_class.methode = staticmethod(methode)
essai_class.methode ()
```

Toutefois, il est conseillé de placer l'instruction qui contient `staticmethod` à l'intérieur de la classe. Elle n'y sera exécutée qu'une seule fois comme le montre l'exemple suivant :

```
class essai_class:
    print "création d'une instance de la classe essai_class"
    methode = staticmethod(methode)
cl = classe_vide () # affiche création d'une instance de la classe essai_class
ck = classe_vide () # n'affiche rien
```

Les méthodes statiques sont souvent employées pour créer des instances spécifiques d'une classe.

```
class Couleur :
    def __init__ (self, r, v, b) : self.r, self.v, self.b =  r, v, b
    def __str__ (self) : return str ( (self.r,self.v,self.b))
    def blanc () : return Couleur (255,255,255)
    def noir  () : return Couleur (0,0,0)
    blanc = staticmethod (blanc)
    noir  = staticmethod (noir)

c = Couleur.blanc ()
print c               # affiche (255, 255, 255)
c = Couleur.noir ()
print c               # affiche (0, 0, 0)
```

### 4.5.2 Attributs statiques

**Définition 4.21 : attribut statique**
Les attributs statiques sont des attributs qui peuvent être utilisés même si aucune instance de la classe où ils sont définis n'a été créée. Ces attributs sont partagés par toutes les instances.

**syntaxe 4.22 : classe, attribut statique**

```
class nom_class :
    attribut_statique = valeur
    def nom_methode (self,params, ...):
        nom_class.attribut_statique2 = valeur2
    def nom_methode_st (params, ...) :
        nom_class.attribut_statique3 = valeur3
    nom_methode_st = staticmethod (nom_methode_st)
```

`nom_classe` est une classe, `nom_methode` est une méthode non statique, `nom_methode_st` est une méthode statique. Les trois paramètres `attribut_statique`, `attribut_statique2`, `attribut_statique3` sont statiques, soit parce qu'ils sont déclarés en dehors d'une méthode, soit parce que leur déclaration fait intervenir le nom de la classe.

Pour le programme suivant, la méthode `meth` n'utilise pas `self.x` mais `essai_class.x`. L'attribut `x` est alors un attribut statique, partagé par toutes les instances. C'est pourquoi dans l'exemple qui suit l'instruction `z.meth()` affiche la valeur 6 puisque l'appel `y.meth()` a incrémenté la variable statique `x`.

```
class essai_class:
    x = 5
    def meth(self):
        print essai_class.x
        essai_class.x += 1

y = essai_class ()
z = essai_class ()
y.meth()    # affiche 5
z.meth()    # affiche 6
```

**Remarque 4.23 : ambiguïté**
Même si un attribut est statique, il peut être utilisé avec la syntaxe `self.attribut_statique` dans une méthode non statique à condition qu'un attribut non statique ne porte pas le même nom. Si tel est pourtant le cas, certaines confusions peuvent apparaître :

```
class exemple_classe:
    rnd = 42
    def incremente_rnd (self):
        self.rnd += 1
        return self.rnd

cl = exemple_classe()
```

```
print cl.__dict__                      # affiche {}
print cl.__class__.__dict__ ["rnd"]  # affiche 42
cl.incremente_rnd ()
print cl.__dict__                      # affiche {'rnd': 43}
print cl.__class__.__dict__ ["rnd"]  # affiche 42
```

Dans ce cas, ce sont en fait deux attributs qui sont créés. Le premier est un attribut statique créé avec la seconde ligne de l'exemple `rnd = 42`. Le second attribut n'est pas statique et apparaît dès la première exécution de l'instruction `self.rnd+=1` comme le montre son apparition dans l'attribut `__dict__` qui ne recense pas les attributs statiques.

### 4.5.3 Ajout de méthodes

Ce point décrit une fonctionnalité du langage *Python* rarement utilisée. Il offre la possibilité d'ajouter une méthode à une classe alors même que cette fonction est définie à l'extérieur de la déclaration de la classe. Cette fonction doit obligatoirement accepter un premier paramètre qui recevra l'instance de la classe. La syntaxe utilise le mot-clé `classmethod`.

**syntaxe 4.24 : classe,** `classmethod`

```
def nom_methode (cls) :
    # code de la fonction}

class nom_classe :
    # code de la classe
    nom_methode = classmethod (nom_methode)        # syntaxe 1

nom_classe.nom_methode = classmethod (nom_methode) # syntaxe 2
```

`nom_classe` est une classe, `nom_methode` est une méthode, `nom_methode` est une fonction qui est par la suite considérée comme une méthode de la classe `nom_methode` grâce à l'une ou l'autre des deux instructions incluant le mot-clé `classmethod`.

Dans l'exemple qui suit, cette syntaxe est utilisée pour inclure trois méthodes à la classe `essai_class` selon que la méthode est déclarée et affectée à cette classe à l'intérieur ou à l'extérieur du corps de `essai_class`.

```
def meth3 (cls): print "ok meth3", cls.x
def meth4 (cls): print "ok meth4", cls.x

class essai_classe:
    x = 5
    def meth(self): print "ok meth", self.x
    def meth2(cls): print "ok meth2", cls.x

    meth3 = classmethod (meth3)

x = essai_classe ()
x.meth ()                                  # affiche ok meth 5
x.meth2 ()                                 # affiche ok meth2 5
x.meth3 ()                                 # affiche ok meth3 5
```

```
essai_classe.meth4 = classmethod (meth4)
x.meth4 ()                              # affiche ok meth4 5
```

### 4.5.4 Propriétés

Cette fonctionnalité est également peu utilisée, elle permet des raccourcis d'écriture. Les propriétés permettent de faire croire à l'utilisateur d'une instance de classe qu'il utilise une variable alors qu'il utilise en réalité une ou plusieurs méthodes. A chaque fois que le programmeur utilise ce faux attribut, il appelle une méthode qui calcule sa valeur. A chaque fois que le programmeur cherche à modifier la valeur de ce faux attribut, il appelle une autre méthode qui modifie l'instance.

**syntaxe 4.25 : classe, propriété**

```
class nom_classe :
    # code de la classe
    nom_propriete = property (fget, fset, fdel, doc)
```

Au sein de ces trois lignes, `nom_classe` est une classe, `nom_propriete` est le nom de la propriété, `fget` est la méthode qui doit retourner la valeur du pseudo-attribut `nom_propriete`, `fset` est la méthode qui doit modifier la valeur du pseudo-attribut `nom_propriete`, `fdel` est la méthode qui doit détruire le pseudo-attribut `nom_propriete`, `doc` est un commentaire qui apparaîtra lors de l'appel de la fonction `help(nom_class)` ou `help(nom_class.nom_propriete)`.

Pour illustrer l'utilisation des propriétés, on part d'une classe `nombre_complexe` qui ne contient que les parties réelle et imaginaire. Lorsqu'on cherche à obtenir le module [4], on fait appel à une méthode qui calcule ce module. Lorsqu'on cherche à modifier ce module, on fait appel à une autre méthode qui multiplie les parties réelle et imaginaire par un nombre réel positif de manière à ce que le nombre complexe ait le module demandé. On procède de même pour la propriété `arg`.

La propriété `conj` retourne quant à elle le conjugué du nombre complexe mais la réciproque n'est pas prévue. On ne peut affecter une valeur à `conj`.

```
import math

class nombre_complexe(object):          # voir remarque après l'exemple
    def __init__ (self, a = 0, b= 0):
        self.a = a
        self.b = b

    def __str__ (self) :
        if   self.b == 0 : return "%f" % (self.a)
        elif self.b >  0 : return "%f + %f i" % (self.a, self.b)
        else             : return "%f - %f i" % (self.a, -self.b)

    def get_module (self):
        return math.sqrt (self.a * self.a + self.b * self.b)
```

4. Le module désigne ici le module d'un nombre complexe qui est égal à sa norme. On le note $|a+ib| = \sqrt{a^2+b^2}$.

```
    def set_module (self,m):
        r = self.get_module ()
        if r == 0:
            self.a = m
            self.b = 0
        else :
            d       = m / r
            self.a *= d
            self.b *= d

    def get_argument (self) :
        r = self.get_module ()
        if r == 0 : return 0
        else      : return math.atan2 (self.b / r, self.a / r)

    def set_argument (self,arg) :
        m       = self.get_module ()
        self.a  = m * math.cos (arg)
        self.b  = m * math.sin (arg)

    def get_conjugue (self):
        return nombre_complexe (self.a,-self.b)

    module = property (fget = get_module,   fset = set_module,   doc = "module")
    arg    = property (fget = get_argument, fset = set_argument, doc = "argument")
    conj   = property (fget = get_conjugue,                      doc = "conjugué")

c = nombre_complexe (0.5,math.sqrt (3)/2)
print "c = ",         c          # affiche c =  0.500000 + 0.866025 i
print "module = ",    c.module   # affiche module =  1.0
print "argument = ",  c.arg      # affiche argument =  1.0471975512

c           = nombre_complexe ()
c.module    = 1
c.arg       = math.pi * 2 / 3
print "c = ",         c          # affiche c =  -0.500000 + 0.866025 i
print "module = ",    c.module   # affiche module =  1.0
print "argument = ",  c.arg      # affiche argument =  2.09439510239
print "conjugué = ",  c.conj     # affiche conjugué =  -0.500000 - 0.866025 i
```

La propriété `conj` ne possède pas de fonction qui permet de la modifier. Par conséquent, l'instruction `c.conj = nombre_complexe(0,0)` produit l'erreur suivante :

```
Traceback (most recent call last):
  File "cour2.py", line 53, in ?
    c.conj = nombre_complexe (0,0)
AttributeError: can't set attribute
```

Etant donné qu'une propriété porte déjà le nom de `conj`, aucun attribut du même nom ne peut être ajouté à la classe `nombre_complexe`.

**Remarque 4.26 : propriété et héritage**
Afin que la propriété fonctionne correctement, il est nécessaire que la classe hérite de la classe `object` ou une de ses descendantes (voir également paragraphe 4.8).

## 4.6 Copie d'instances

### 4.6.1 Copie d'instance de classe simple

Aussi étrange que cela puisse paraître, le signe = ne permet pas de recopier une instance de classe. Il permet d'obtenir deux noms différents pour désigner le même objet. Dans l'exemple qui suit, la ligne `nb2 = nb` ne fait pas de copie de l'instance `nb`, elle permet d'obtenir un second nom `nb2` pour l'instance `nb`. Vu de l'extérieur, la ligne `nb2.rnd = 0` paraît modifier à la fois les objets `nb` et `nb2` puisque les lignes `print nb.rnd` et `print nb2.rnd` affichent la même chose. En réalité, `nb` et `nb2` désignent le même objet.

```
class exemple_classe:
    def __init__ (self) : self.rnd = 42
    def methode1(self,n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb  = exemple_classe ()
nb2 = nb
print nb.rnd        # affiche 42
print nb2.rnd       # affiche 42

nb2.rnd = 0

print nb2.rnd       # affiche 0, comme prévu
print nb.rnd        # affiche 0, si nb et nb2 étaient des objets différents,
                    # cette ligne devrait afficher 42
```

Pour créer une copie de l'instance `nb`, il faut le dire explicitement en utilisant la fonction `copy` du module `copy` (voir le chapitre 6).

**syntaxe 4.27 : fonction `copy`**

```
import copy
nom_copy = copy.copy (nom_instance)
```

`nom_instance` est une instance à copier, `nom_copy` est le nom désignant la copie.

L'exemple suivant applique cette copie sur la classe `exemple_classe` générant des nombres aléatoires.

```
class exemple_classe:
    def __init__ (self) : self.rnd = 42
    def methode1(self,n):
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb = exemple_classe ()

import copy           # pour utiliser le module copy
nb2 = copy.copy (nb)  # copie explicite

print nb.rnd    # affiche 42
print nb2.rnd   # affiche 42
```

```
nb2.rnd = 0

print nb2.rnd   # affiche 0
print nb.rnd    # affiche 42
```

**Remarque 4.28 : suppression de variables associées à la même instance**
Le symbole égalité ne fait donc pas de copie, ceci signifie qu'une même instance de classe peut porter plusieurs noms.

```
m  = [ 0, 1 ]
m2 = m
del m2   # supprime l'identificateur mais pas la liste
print m  # affiche [0, 1]
```

La suppression d'un objet n'est effective que s'il ne reste aucune variable le référençant. L'exemple suivant le montre.

```
class CreationDestruction (object) :

    def __init__ (self) :
        print "constructeur"

    def __new__ (self) :
        print "__new__"
        return object.__new__ (self)

    def __del__ (self) :
        print "__del__"

print "a"
m = CreationDestruction ()
print "b"
m2 = m
print "c"
del m
print "d"
del m2
print "e"
```

La sortie de ce programme est la suivante :

```
a
__new__
constructeur
b
c
d
__del__
e
```

Le destructeur est appelé autant de fois que le constructeur. Il est appelé lorsque plus aucun identificateur n'est relié à l'objet. Cette configuration survient lors de l'exemple précédent car le mot-clé `del` a détruit tous les identificateurs `m` et `m2` qui étaient reliés au même objet.

### 4.6.2 Copie d'instance de classes incluant d'autres classes

La fonction `copy` n'est pas suffisante lorsqu'une classe inclut des attributs qui sont eux-mêmes des classes incluant des attributs. Dans l'exemple qui suit, la classe `exemple_classe` inclut un attribut de type `classe_incluse` qui contient un attribut `attr`. Lors de la copie à l'aide de l'instruction `nb2 = copy.copy(nb)`, l'attribut `inclus` n'est pas copié, c'est l'instruction `nb2.inclus = nb.inclus` qui est exécutée. On se retrouve donc avec deux noms qui désignent encore le même objet.

```
class classe_incluse:
    def __init__ (self) : self.attr = 3

class exemple_classe:
    def __init__ (self) :
        self.inclus = classe_incluse ()
        self.rnd    = 42

nb = exemple_classe ()

import copy            # pour utiliser le module copy
nb2 = copy.copy (nb)   # copie explicite

print nb.inclus.attr   # affiche 3
print nb2.inclus.attr  # affiche 3

nb2.inclus.attr = 0

print nb.inclus.attr   # affiche 0 (on voudrait 3 ici)
print nb2.inclus.attr  # affiche 0
```

Pour effectivement copier les attributs dont le type est une classe, la première option - la plus simple - est de remplacer la fonction `copy` par la fonction `deepcopy`. Le comportement de cette fonction dans le cas des classes est le même que dans le cas des listes comme l'explique la remarque 2.18 page 48. La seconde solution, rarement utilisée, est d'utiliser l'opérateur `__copy__` et ainsi écrire le code associé à la copie des attributs de la classe.

**syntaxe 4.29 : classe, opérateur `__copy__`**

```
class nom_classe :
    def __copy__ () :
        copie = nom_classe (...)
        # ...
        return copie
```

`nom_classe` est le nom d'une classe. La méthode `__copy__` doit retourner une instance de la classe `nom_classe`, dans cet exemple, cette instance a pour nom `copie`.

L'exemple suivant montre un exemple d'implémentation de la classe `__copy__`. Cette méthode crée d'abord une autre instance `copie` de la classe `exemple_classe` puis initialise un par un ses membres. L'attribut `rnd` est recopié grâce à une affectation car c'est un nombre. L'attribut `inclus` est recopié grâce à la fonction `copy` du module `copy` car c'est une instance de classe. Après la copie, on vérifie bien que

modifier l'attribut `inclus.attr` de l'instance `nb` ne modifie pas l'attribut `inclus.attr` de l'instance `nb2`.

```
import copy

class classe_incluse:
    def __init__ (self) : self.attr = 3

class exemple_classe:
    def __init__ (self) :
        self.inclus  = classe_incluse ()
        self.rnd     = 42
    def __copy__ (self):
        copie        = exemple_classe ()
        copie.rnd    = self.rnd
        copie.inclus = copy.copy (self.inclus)
        return copie

nb  = exemple_classe ()

nb2 = copy.copy (nb)   # copie explicite,
                       # utilise l'opérateur __copy__,
                       # cette ligne est équivalente à
                       # nb2 = nb.__copy__()

print nb.rnd           # affiche 42
print nb2.rnd          # affiche 42
print nb.inclus.attr   # affiche 3
print nb2.inclus.attr  # affiche 3

nb.inclus.attr = 0
nb.rnd         = 1

print nb.rnd           # affiche 1
print nb2.rnd          # affiche 42
print nb.inclus.attr   # affiche 0
print nb2.inclus.attr  # affiche 3 (c'est le résultat souhaité)
```

**Remarque 4.30 : affectation et copie**
On peut se demander pourquoi l'affectation n'est pas équivalente à une copie. Cela tient au fait que l'affectation en langage *Python* est sans cesse utilisée pour affecter le résultat d'une fonction à une variable. Lorsque ce résultat est de taille conséquente, une copie peut prendre du temps. Il est préférable que le résultat de la fonction reçoive le nom prévu pour le résultat.

```
def fonction_liste ():
    return range (4,7) # retourne la liste [4,5,6]
l = fonction_liste ()  # la liste [4,5,6] n'est pas recopiée,
                       # l'identificateur l lui est affecté
```

Lorsqu'une fonction retourne un résultat mais que celui-ci n'est pas attribué à un nom de variable. Le langage *Python* détecte automatiquement que ce résultat n'est plus lié à aucune variable. Il est détruit automatiquement.

```
def fonction_liste ():
    return range (4,7)
```

```
fonction_liste () # la liste [4,5,6] n'est pas recopiée,
                  # elle n'est pas non plus attribuée à une variable,
                  # elle est alors détruite automatiquement par le langage Python
```

### 4.6.3 Listes et dictionnaires

Les listes et les dictionnaires sont des types modifiables et aussi des classes. Par conséquent, l'affectation et la copie ont un comportement identique à celui des classes.

```
l  = [4,5,6]
l2 = l
print l       # affiche [4, 5, 6]
print l2      # affiche [4, 5, 6]
l2 [1] = 10
print l       # affiche [4, 10, 6]
print l2      # affiche [4, 10, 6]
```

Pour effectuer une copie, il faut écrire le code suivant :

```
l  = [4,5,6]
import copy
l2 = copy.copy (l)
print l       # affiche [4, 5, 6]
print l2      # affiche [4, 5, 6]
l2 [1] = 10
print l       # affiche [4, 5, 6]
print l2      # affiche [4, 10, 6]
```

La fonction `copy` ne suffit pourtant pas lorsque l'objet à copier est par exemple une liste incluant d'autres objets. Elle copiera la liste et ne fera pas de copie des objets eux-mêmes.

```
import copy
l  = [ [i] for i in range(0,3)]
ll = copy.copy (l)
print l, "  -  ", ll    # affiche [[0], [1], [2]]   -   [[0], [1], [2]]
ll [0][0] = 6
print l, "  -  ", ll    # affiche [[6], [1], [2]]   -   [[6], [1], [2]]
```

Il n'est pas possible de modifier la méthode `__copy__` d'un objet de type liste. Il existe néanmoins la fonction `deepcopy` qui permet de faire une copie à la fois de la liste et des objets qu'elle contient.

```
import copy
l  = [ [i] for i in range(0,3)]
ll = copy.deepcopy (l)
print l, "  -  ", ll    # affiche [[0], [1], [2]]   -   [[0], [1], [2]]
ll [0][0] = 6
print l, "  -  ", ll    # affiche [[0], [1], [2]]   -   [[6], [1], [2]]
```

### 4.6.4 `copy` et `deepcopy`

La fonction `copy` effectue une copie d'un objet, la fonction `deepcopy` effectue une copie d'un objet et de ceux qu'il contient. La fonction `copy` est associée à la méthode `__copy__` tandis que la fonction `deepcopy` est associée à la méthode `__deepcopy__`. Il est rare que l'une de ces deux méthodes doivent être redéfinies. L'intérêt de ce paragraphe est plus de montrer le mécanisme que cache la fonction `deepcopy` qui est la raison pour laquelle il existe deux fonctions de copie et non une seule.

**syntaxe 4.31 : fonction `deepcopy`**

```
import copy
memo = { }
nom_copy = copy.deepcopy (nom_instance [,memo])
```

`nom_instance` est une instance à copier, `nom_copy` est le nom désignant la copie. `memo` est un paramètre facultatif : s'il est envoyé à la fonction `deepcopy`, il contiendra alors la liste de toutes les copies d'objet effectuées lors de cet appel.

**syntaxe 4.32 : classe, opérateur `__deepcopy__`**

```
class nom_classe :
    def __deepcopy__ (self,memo) :
        copie = copy.copy (self)
        # ...
        return copie
```

`nom_classe` est le nom d'une classe. La méthode `__deepcopy__` doit retourner une instance de la classe `nom_classe`, dans cet exemple, cette instance a pour nom `copie`. Le paramètre `memo` permet de conserver la liste des copies effectuées à condition d'appeler `deepcopy` avec un dictionnaire en paramètre.

Le programme suivant reprend le second programme du paragraphe 4.6.2 et modifie la classe `classe_incluse` pour distinguer copie et copie profonde. Il peut être utile de lire le paragraphe 2.3.3.4 (page 52) pour comprendre pourquoi un dictionnaire utilisant comme clé une instance de classe est possible.

```
import copy

class classe_incluse:
    def __init__ (self) : self.attr = 3

class exemple_classe:
    def __init__ (self) :
        self.inclus = classe_incluse ()
        self.rnd    = 42
    def __copy__ (self):
        copie       = exemple_classe ()
        copie.rnd   = self.rnd
        return copie
    def __deepcopy__ (self,memo):
        if self in memo : return memo [self]
        copie        = copy.copy (self)
        memo [self]  = copie    # mémorise la copie de self qui est copie
```

```
        copie.inclus = copy.deepcopy (self.inclus,memo)
        return copie

nb = exemple_classe ()

nb2 = copy.deepcopy (nb)   # copie explicite à tous niveaux,
                           # utilise l'opérateur __copy__,
                           # cette ligne est équivalente à
                           # nb2 = nb.__deepcopy__()

print nb.rnd           # affiche 42
print nb2.rnd          # affiche 42
print nb.inclus.attr   # affiche 3
print nb2.inclus.attr  # affiche 3

nb.inclus.attr = 0
nb.rnd = 1

print nb.rnd           # affiche 1
print nb2.rnd          # affiche 42
print nb.inclus.attr   # affiche 0
print nb2.inclus.attr  # affiche 3  # résultat souhaité
```

On peut se demander quel est l'intérêt de la méthode `__deepcopy__` et surtout du paramètre `memo` modifié par la ligne `memo[self] = copie`. Ce détail est important lorsqu'un objet inclut un attribut égal à lui-même ou inclut un objet qui fait référence à l'objet de départ comme dans l'exemple qui suit.

```
import copy

class Objet1 :
    def __init__ (self, i) : self.i = i
    def __str__ (self) :
        return "o1 " + str (self.i) + " : " + str (self.o2.i)

class Objet2 :
    def __init__ (self, i, o) :
        self.i  = i
        self.o1 = o
        o.o2    = self
    def __str__ (self) :
        return "o2 " + str (self.i) + " : " + str (self.o1.i)

    def __deepcopy__ (self,memo) : return Objet2 (self.i, self.o1)

o1 = Objet1 (1)
o2 = Objet2 (2, o1)
print o1  # affiche o1 1 : 2
print o2  # affiche o2 2 : 1

o3   = copy.deepcopy (o2)
o3.i = 4
print o1  # affiche o1 1 : 4    --> on voudrait 2
print o2  # affiche o2 2 : 1
print o3  # affiche o2 4 : 1
```

On modifie le programme comme suit pour obtenir une recopie d'instances de classes qui pointent les unes sur vers les autres. Le paramètre `memo` sert à savoir si la copie

de l'objet a déjà été effectuée ou non. Si non, on fait une copie, si oui, on retourne la copie précédemment effectuée et conservée dans `memo`.

```
import copy

class Objet1 :
    def __init__ (self, i) : self.i = i
    def __str__ (self) :
        return "o1 " + str (self.i) + " : " + str (self.o2.i)
    def __deepcopy__ (self,memo={}) :
        if self in memo : return memo [self]
        r           = Objet1 (self.i)
        memo [self] = r
        r.o2        = copy.deepcopy (self.o2, memo)
        return r

class Objet2 :
    def __init__ (self, i, o) :
        self.i  = i
        self.o1 = o
        o.o2    = self
    def __str__ (self) :
        return "o2 " + str (self.i) + " : " + str (self.o1.i)

    def __deepcopy__ (self,memo = {}) :
        if self in memo : return memo [self]
        r           = Objet2 (self.i, self.o1)
        memo [self] = r
        r.o1        = copy.deepcopy (self.o1, memo)
        return r

o1 = Objet1 (1)
o2 = Objet2 (2, o1)

print o1  # affiche o1 1 : 2
print o2  # affiche o2 2 : 1

o3   = copy.deepcopy (o2)
o3.i = 4
print o1  # affiche o1 1 : 2    --> on a 2 cette fois-ci
print o2  # affiche o2 2 : 1
print o3  # affiche o2 4 : 1
```

Ces problématiques se rencontrent souvent lorsqu'on aborde le problème de la sérialisation d'un objet qui consiste à enregistrer tout objet dans un fichier, même si cet objet inclut des références à des objets qui font référence à lui-même. C'est ce qu'on appelle des références circulaires. L'enregistrement d'un tel objet avec des références circulaires et sa relecture depuis un fichier se résolvent avec les mêmes artifices que ceux proposés ici pour la copie. L'utilisation des opérateurs `__copy__` et `__deepcopy__` est peu fréquente. Les fonctions `copy` et `deepcopy` du module `copy` suffisent dans la plupart des cas.

## 4.7 Attributs figés

Il arrive parfois qu'une classe contienne peu d'informations et soit utilisée pour créer un très grand nombre d'instances. Les paragraphes précédents ont montré que l'utilisation des attributs était assez souple. Il est toujours possible d'ajouter un attribut à n'importe quelle instance. En contrepartie, chaque instance conserve en mémoire un dictionnaire `__dict__` qui recense tous les attributs qui lui sont associés. Pour une classe susceptible d'être fréquemment instanciée comme un point dans l'espace (voir paragraphe 4.5.4), chaque instance n'a pas besoin d'avoir une liste variable d'attributs. Le langage *Python* offre la possibilité de figer cette liste.

**syntaxe 4.33 : classe, `__slots__`**

```
class nom_classe (object) :
    __slots__ = "attribut_1", ..., "attribut_n"
```

`nom_classe` est le nom de la classe, elle doit hériter de `object` ou d'une classe qui en hérite elle-même (voir paragraphe 4.8). Il faut ensuite ajouter au début du corps de la classe la ligne `__slots__` = "`attribut_1`",...,"`attribut_n`" où `attribut_1` à `attribut_n` sont les noms des attributs de la classe. Aucun autre ne sera accepté.

L'exemple suivant utilise cette syntaxe pour définir un point avec seulement trois attributs `_x`, `_y`, `_z`.

```
class point_espace(object):
    __slots__ = "_x", "_y", "_z"

    def __init__ (self, x,y,z): self._x, self._y, self._z = x,y,z
    def __str__(self): return "(%f,%f,%f)" % (self._x, self._y, self._z)

a = point_espace (1,-2,3)
print a
```

Etant donné que la liste des attributs est figée, l'instruction `a.j = 6` qui ajoute un attribut `j` à l'instance `a` déclenche une exception (voir paragraphe 5). La même erreur se déclenche si on cherche à ajouter cet attribut depuis une méthode (`self.j = 6`).

```
Traceback (most recent call last):
  File "cours4.py", line 15, in ?
    a.j = 6
AttributeError: 'point_espace' object has no attribute 'j'
```

L'attribut `__dict__` n'existe pas non plus, par conséquent, l'expression `a.__dict__` génère la même exception. La présence de l'instruction `__slots__` = ... n'a aucun incidence sur les attributs statiques.

## 4.8 Héritage

L'héritage est un des grands avantages de la programmation objet. Il permet de créer une classe à partir d'une autre en ajoutant des attributs, en modifiant ou

en ajoutant des méthodes. En quelque sorte, on peut modifier des méthodes d'une classe tout en conservant la possibilité d'utiliser les anciennes versions.

### 4.8.1 Exemple autour de pièces de monnaie

On désire réaliser une expérience à l'aide d'une pièce de monnaie. On effectue cent tirages successifs et on compte le nombre de fois où la face pile tombe. Le programme suivant implémente cette expérience sans utiliser la programmation objet.

```
import random  # extension interne incluant des fonctions
               # simulant des nombres aléatoires,
               # random.randint (a,b) --> retourne un nombre entier entre a et b
               # cette ligne doit être ajoutée à tous les exemples suivant
               # même si elle n'y figure plus

def cent_tirages () :
    s = 0
    for i in range (0,100) : s += random.randint (0,1)
    return s

print cent_tirages ()
```

On désire maintenant réaliser cette même expérience pour une pièce truquée pour laquelle la face pile sort avec une probabilité de $0,7$. Une solution consiste à réécrire la fonction `cent_tirages` pour la pièce truquée.

```
def cent_tirages () :
    s = 0
    for i in range (0,100) :
        t = random.randint (0,10)
        if t >= 3 : s += 1
    return s

print cent_tirages ()
```

Toutefois cette solution n'est pas satisfaisante car il faudrait réécrire cette fonction pour chaque pièce différente pour laquelle on voudrait réaliser cette expérience. Une autre solution consiste donc à passer en paramètre de la fonction `cent_tirages` une fonction qui reproduit le comportement d'une pièce, qu'elle soit normale ou truquée.

```
def piece_normale () :
    return random.randint (0,1)

def piece_truquee () :
    t = random.randint (0,10)
    if t >= 3 : return 1
    else : return 0

def cent_tirages (piece) :
    s = 0
    for i in range (0,100) : s += piece ()
    return s

print cent_tirages (piece_normale)
print cent_tirages (piece_truquee)
```

Mais cette solution possède toujours un inconvénient car les fonctions associées à chaque pièce n'acceptent aucun paramètre. Il n'est pas possible de définir une pièce qui est normale si la face *pile* vient de sortir et qui devient truquée si la face *face* vient de sortir [5]. On choisit alors de représenter une pièce normale par une classe.

```
class piece_normale :
    def tirage (self) :
        return random.randint (0,1)

    def cent_tirages (self) :
        s = 0
        for i in range (0,100) : s += self.tirage ()
        return s

p = piece_normale ()
print p.cent_tirages ()
```

On peut aisément recopier et adapter ce code pour la pièce truquée.

```
class piece_normale :
    def tirage (self) :
        return random.randint (0,1)

    def cent_tirages (self) :
        s = 0
        for i in range (0,100) : s += self.tirage ()
        return s

class piece_truquee :
    def tirage (self) :
        t = random.randint (0,10)
        if t >= 3 : return 1
        else : return 0

    def cent_tirages (self) :
        s = 0
        for i in range (0,100) : s += self.tirage ()
        return s

p  = piece_normale ()
print p.cent_tirages ()
p2 = piece_truquee ()
print p2.cent_tirages ()
```

Toutefois, pour les deux classes `piece_normale` et `piece_truquee`, la méthode `cent_tirage` est exactement la même. Il serait préférable de ne pas répéter ce code puisque si nous devions modifier la première - un nombre de tirages différent par

5. Il faudrait pour cela créer une variable globale.

```
avant = 0
def piece_tres_truquee() :
    global avant
    if avant == 0 : avant = piece_truquee()
    else : avant = piece_normale()
    return avant
```

C'est comme si la fonction `piece_tres_truquee` avait un paramètre caché. Cette solution n'est pas conseillée car c'est le genre de détail qu'on oublie par la suite.

exemple -, il faudrait également modifier la seconde. La solution passe par l'héritage. On va définir la classe `piece_truquee` à partir de la classe `piece_normale` en remplaçant seulement la méthode `tirage` puisqu'elle est la seule à changer.

On indique à la classe `piece_truquee` qu'elle hérite - ou dérive - de la classe `piece_normale` en mettant `piece_normale` entre parenthèses sur la ligne de la déclaration de la classe `piece_truquee`. Comme la méthode `cent_tirages` ne change pas, elle n'a pas besoin d'apparaître dans la définition de la nouvelle classe même si cette méthode est aussi applicable à une instance de la classe `piece_truquee`.

```
class piece_normale :
    def tirage (self) :
        return random.randint (0,1)

    def cent_tirages (self) :
        s = 0
        for i in range (0,100) : s += self.tirage ()
        return s

class piece_truquee (piece_normale) :
    def tirage (self) :
        t = random.randint (0,10)
        if t >= 3 : return 1
        else : return 0

p  = piece_normale ()
print p.cent_tirages ()
p2 = piece_truquee ()
print p2.cent_tirages ()
```

Enfin, on peut définir une pièce très truquée qui devient truquée si *face* vient de sortir et qui redevient normale si *pile* vient de sortir. Cette pièce très truquée sera implémentée par la classe `piece_tres_truquee`. Elle doit contenir un attribut `avant` qui conserve la valeur du précédent tirage. Elle doit redéfinir la méthode `tirage` pour être une pièce normale ou truquée selon la valeur de l'attribut `avant`. Pour éviter de réécrire des méthodes déjà écrites, la méthode `tirage` de la classe `piece_tres_truquee` doit appeler la méthode `tirage` de la classe `piece_truquee` ou celle de la classe `piece_normale` selon la valeur de l'attribut `avant`.

```
class piece_normale :
    def tirage (self) :
        return random.randint (0,1)

    def cent_tirages (self) :
        s = 0
        for i in range (0,100) : s += self.tirage ()
        return s

class piece_truquee (piece_normale) :
    def tirage (self) :
        t = random.randint (0,10)
        if t >= 3 : return 1
        else : return 0

class piece_tres_truquee (piece_truquee) :
    def __init__(self) :
```

```
        # création de l'attribut avant
        self.avant = 0

    def tirage (self) :
        if self.avant == 0 :
            # appel de la méthode tirage de la classe piece_truquee
            self.avant = piece_truquee.tirage (self)
        else :
            # appel de la méthode tirage de la classe piece_normale
            self.avant = piece_normale.tirage (self)
        return self.avant

p = piece_normale ()
print "normale ", p.cent_tirages ()
p2 = piece_truquee ()
print "truquee ", p2.cent_tirages ()
p3 = piece_tres_truquee ()
print "tres truquee ", p3.cent_tirages ()
```

L'héritage propose donc une manière élégante d'organiser un programme. Il rend possible la modification des classes d'un programme sans pour autant les altérer.

**Définition 4.34 : héritage**
On dit qu'une classe $B$ hérite d'une autre classe $A$ si la déclaration de $B$ inclut les attributs et les méthodes de la classe $A$.

La surcharge est un autre concept qui va de pair avec l'héritage. Elle consiste à redéfinir des méthodes déjà définies chez l'ancêtre. Cela permet de modifier le comportement de méthodes bien que celles-ci soient utilisées par d'autres méthodes dont la définition reste inchangée.

**Définition 4.35 : surcharge**
Lorsqu'une classe $B$ hérite de la classe $A$ et redéfinit une méthode de la classe $A$ portant le même nom, on dit qu'elle surcharge cette méthode. S'il n'est pas explicitement précisé qu'on fait appel à une méthode d'une classe donnée, c'est toujours la méthode surchargée qui est exécutée.

### 4.8.2 Syntaxe

**syntaxe 4.36 : classe, héritage**

```
class nom_classe (nom_ancetre) :
    # corps de la classe
    # ...
```

`nom_classe` désigne le nom d'une classe qui hérite ou dérive d'une autre classe `nom_ancetre`. Celle-ci `nom_ancetre` doit être une classe déjà définie.

L'utilisation de la fonction `help` permet de connaître tous les ancêtres d'une classe. On applique cette fonction à la classe `piece_tres_truquee` définie au paragraphe précédent.

```
help (piece_tres_truquee)
```

On obtient le résultat suivant :

```
Help on class piece_tres_truquee in module __main__:

class piece_tres_truquee(piece_truquee)
 |  Method resolution order:
 |      piece_tres_truquee
 |      piece_truquee
 |      piece_normale
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  tirage(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from piece_normale:
 |
 |  cent_tirages(self)
```

**Remarque 4.37 : Method resolution order**
La rubrique `Method resolution order` indique la liste des héritages successifs qui ont mené à la classe `piece_tres_truquee`. Cette rubrique indique aussi que, lorsqu'on appelle une méthode de la classe `piece_tres_truquee`, si elle n'est pas redéfinie dans cette classe, le langage *Python* la cherchera chez l'ancêtre direct, ici, la classe `piece_truquee`. Si elle ne s'y trouve toujours pas, *Python* ira la chercher aux niveaux précédents jusqu'à ce qu'il la trouve.

L'attribut `__bases__` d'une classe (voir paragraphe 4.3.2) contient le (ou les ancêtres, voir paragraphe 4.8.4). Il suffit d'interroger cet attribut pour savoir si une classe hérite d'une autre comme le montre l'exemple suivant.

```
for l in piece_tres_truquee.__bases__ :
    print l   # affiche __main__.piece_truquee
print piece_normale in piece_tres_truquee.__bases__  # affiche False
print piece_truquee in piece_tres_truquee.__bases__  # affiche True
```

La fonction `issubclass` permet d'obtenir un résultat équivalent. `issubclass(A,B)` indique si la classe `A` hérite directement ou indirectement de la classe `B`. Le paragraphe 4.8.5 revient sur cette fonction.

```
print issubclass (piece_tres_truquee, piece_normale)  # affiche True
print issubclass (piece_truquee, piece_normale)       # affiche True
```

Dans les exemples précédents, `piece_normale` ne dérive d'aucune autre classe. Toutefois, le langage *Python* propose une classe d'objets dont héritent toutes les autres classes définies par le langage : c'est la classe `object`. Les paragraphes précédents ont montré qu'elle offrait certains avantages (voir paragraphe 4.5.4 sur les propriétés ou encore paragraphe 4.7 sur les attributs non liés).

Le paragraphe précédent a montré qu'il était parfois utile d'appeler dans une méthode une autre méthode appartenant explicitement à l'ancêtre direct de cette classe ou à un de ses ancêtres. La syntaxe est la suivante.

**syntaxe 4.38 : classe, appel d'une méthode de l'ancêtre**

```
class nom_classe (nom_ancetre) :
    def nom_autre_methode (self, ...) :
        # ...
    def nom_methode (self, ...) :
        nom_ancetre.nom_methode (self, ...)
            # appel de la méthode définie chez l'ancêtre
        nom_ancetre.nom_autre_methode (self, ...)
            # appel d'une autre méthode définie chez l'ancêtre
        self.nom_autre_methode (...)
            # appel d'une méthode surchargée
```

`nom_classe` désigne le nom d'une classe, `nom_ancetre` est le nom de la classe dont `nom_classe` hérite ou dérive. `nom_methode` est une méthode surchargée qui appelle la méthode portant le même nom mais définie dans la classe `nom_ancetre` ou un de ses ancêtres. `nom_autre_methode` est une autre méthode. La méthode `nom_methode` de la classe `nom_classe` peut faire explicitement appel à une méthode définie chez l'ancêtre `nom_ancetre` même si elle est également surchargée ou faire appel à la méthode surchargée.

Ces appels sont très fréquents en ce qui concerne les constructeurs qui appellent le constructeur de l'ancêtre. Il est même conseillé de le faire à chaque fois.

```
class A :
    def __init__ (self) :
        self.x = 0
class B (A) :
    def __init__ (self) :
        A.__init__ (self)
        self.y = 0
```

**Remarque 4.39 : surcharge d'attributs**

Contrairement aux méthodes, la surcharge d'attributs n'est pas possible. Si un ancêtre possède un attribut d'identificateur `a`, les classes dérivées le possèdent aussi et ne peuvent en déclarer un autre du même nom. Cela tient au fait que quelque soit la méthode utilisée, celle de l'ancêtre ou celle d'une classe dérivée, c'est le même dictionnaire d'attributs `__dict__` qui est utilisé. En revanche, si la classe ancêtre déclare un attribut dans son constructeur, il ne faut pas oublier de l'appeler dans le constructeur de la classe fille afin que cette attribut existe pour la classe fille.

```
class ancetre :
    def __init__(self) :
        self.a = 5
    def __str__ (self) :
        return "a = " + str (self.a)

class fille (ancetre) :
    def __init__(self) :
        ancetre.__init__(self)     # cette ligne est importante
```

```
                                    # car sans elle, l'attribut a n'existe pas
        self.a += 1
    def __str__ (self) :
        s = "a = " + str (self.a)
        return s

x = ancetre ()
print x         # affiche a = 5
y = fille ()
print y         # affiche a = 6
```

### 4.8.3 Sens de l'héritage

Il n'est pas toujours évident de concevoir le sens d'un héritage. En mathématique, le carré est un rectangle dont les côtés sont égaux. A priori, une classe `carre` doit dériver d'une classe `rectangle`.

```
class rectangle :
    def __init__(self,a,b) :
        self.a,self.b = a,b
    def __str__ (self) :
        return "rectangle " + str (self.a) +  " x " +  str (self.b)

class carre (rectangle) :
    def __init__( self, a) :
        rectangle.__init__ (self, a,a)

r = rectangle (3,4)
print r  # affiche rectangle 3 x 4
c = carre (5)
print c  # affiche rectangle 5 x 5
```

Toutefois, on peut aussi considérer que la classe `carre` contient une information redondante puisqu'elle possède deux attributs qui seront toujours égaux. On peut se demander s'il n'est pas préférable que la classe `rectangle` hérite de la classe `carre`.

```
class carre :
    def __init__( self, a) :
        self.a = a
    def __str__ (self) :
        return "carre " + str (self.a)

class rectangle (carre):
    def __init__(self,a,b) :
        carre.__init__(self, a)
        self.b = b
    def __str__ (self) :
        return "rectangle " + str (self.a) +  " x " +  str (self.b)

r = rectangle (3,4)
print r  # affiche rectangle 3 x 4
c = carre (5)
print c  # affiche carre 5
```

Cette seconde version minimise l'information à mémoriser puisque la classe `carre` ne contient qu'un seul attribut et non deux comme dans l'exemple précédent. Néan-

moins, il a fallu surcharger l'opérateur `__str__` afin d'afficher la nouvelle information.

Il n'y a pas de meilleur choix parmi ces deux solutions proposées. La première solution va dans le sens des propriétés croissantes, les méthodes implémentées pour les classes de bases restent vraies pour les suivantes. La seconde solution va dans le sens des attributs croissants, des méthodes implémentées pour les classes de bases doivent souvent être adaptées pour les héritiers. En contrepartie, il n'y a pas d'information redondante.

Ce problème d'héritage ne se pose pas à chaque fois. Dans l'exemple du paragraphe 4.8.1 autour des pièces truquées, il y a moins d'ambiguïté sur le sens de l'héritage. Celui-ci est guidé par le problème à résoudre qui s'avère plus simple à concevoir dans le sens d'un héritage d'une pièce normale vers une pièce truquée.

Dans le cas des classes `carre` et `rectangle`, il n'est pas possible de déterminer la meilleure solution tant que leur usage ultérieur n'est pas connu. Ce problème revient également lorsqu'on définit des matrices et des vecteurs. Un vecteur est une matrice d'une seul colonne, il ne possède qu'une seule dimension au lieu de deux pour une matrice. L'exercice 12.1.10 page 288 revient sur ce dilemme.

### 4.8.4 Héritage multiple

Jusqu'à présent, tous les exemples d'héritages entre classes n'ont fait intervenir que deux classes, la classe ancêtre dont hérite la classe descendante. L'héritage multiple part du principe qu'il peut y avoir plusieurs ancêtres pour une même classe. La classe descendante hérite dans ce cas de tous les attributs et méthodes de tous ses ancêtres.

Dans l'exemple qui suit, la classe `C` hérite des classes `A` et `B`. Elle hérite donc des méthodes de `carre` et `cube`. Chacune des classes `A` et `B` contient un constructeur qui initialise l'attribut `a`. Le constructeur de la classe `C` appelle le constructeur de la classe `A` pour initialiser cet attribut.

```
class A :
    def __init__ (self) : self.a = 5
    def carre (self) : return self.a ** 2

class B :
    def __init__ (self) : self.a = 6
    def cube (self) : return self.a ** 3

class C (A,B) :
    def __init__ (self):
        A.__init__ (self)

x = C ()
print x.carre ()    # affiche 25
print x.cube ()     # affiche 125
```

Mais ces héritages multiples peuvent parfois apporter quelques ambiguïtés comme le cas où au moins deux ancêtres possèdent une méthode du même nom. Dans l'exemple qui suit, la classe `C` hérite toujours des classes `A` et `B`. Ces deux classes

possèdent une méthode `calcul`. La classe `C`, qui hérite des deux, possède aussi une méthode `calcul` qui, par défaut, sera celle de la classe `A`.

```
class A :
    def __init__ (self) : self.a = 5
    def calcul (self) : return self.a ** 2

class B :
    def __init__ (self) : self.a = 6
    def calcul (self) : return self.a ** 3

class C (A,B) :
    def __init__ (self):
        A.__init__ (self)

x = C ()
print x.calcul ()  # affiche 25
```

Cette information est disponible via la fonction `help` appliquée à la classe `C`. C'est dans ce genre de situations que l'information apportée par la section `Method resolution order` est importante (voir remarque 4.37 page 127).

```
class C(A, B)
 |  Method resolution order:
 |      C
 |      A
 |      B
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  calcul(self)
```

Pour préciser que la méthode `calcul` de la classe `C` doit appeler la méthode `calcul` de la classe `B` et non `A`, il faut l'écrire explicitement en surchargeant cette méthode.

```
class A :
    def __init__ (self) : self.a = 5
    def calcul (self) : return self.a ** 2

class B :
    def __init__ (self) : self.a = 6
    def calcul (self) : return self.a ** 3

class C (A,B) :
    def __init__ (self):
        A.__init__ (self)

    def calcul (self) :
        return B.calcul (self)

x = C ()
print x.calcul ()  # affiche 125
```

**Remarque 4.40 : héritage multiple et constructeur**
L'exemple précédent est un cas particulier où il n'est pas utile d'appeler les construc-

teurs des deux classes dont la classe `C` hérite mais c'est un cas particulier. Le constructeur de la classe `C` devrait être ainsi :

```
class C (A,B) :
    def __init__ (self):
        A.__init__ (self)
        B.__init__ (self)
```

### 4.8.5 Fonctions `issubclass` et `isinstance`

La fonction `issubclass` permet de savoir si une classe hérite d'une autre.

> **syntaxe 4.41 : fonction `isubclass`**
>
> `issubclass (B,A)`
>
> Le résultat de cette fonction est vrai si la classe `B` hérite de la classe `A`, le résultat est faux dans tous les autres cas. La fonction prend comme argument des classes et non des instances de classes.

L'exemple qui suit utilise cette fonction dont le résultat est vrai même pour des classes qui n'héritent pas directement l'une de l'autre.

```
class A (object) : pass
class B (A)      : pass
class C (B)      : pass

print issubclass (A, B)     # affiche False
print issubclass (B, A)     # affiche True
print issubclass (A, C)     # affiche False
print issubclass (C, A)     # affiche True
```

Lorsqu'on souhaite appliquer la fonction à une instance de classe, il faut faire appel à l'attribut `__class__`. En reprenant les classes définies par l'exemple précédant cela donne :

```
a = A ()
b = B ()
print issubclass (a.__class__, B)     # affiche False
print issubclass (b.__class__, A)     # affiche True
print issubclass (a.__class__, A)     # affiche True
```

La fonction `isinstance` permet de savoir si une instance de classe est d'une type donné. Elle est équivalente à la fonction `issubclass` à ceci près qu'elle prend comme argument une instance et une classe. L'exemple précédent devient avec la fonction `isinstance` :

```
a = A ()
b = B ()
print isinstance (a, B)     # affiche False
print isinstance (b, A)     # affiche True
print isinstance (a, A)     # affiche True
```

L'utilisation des fonctions `issubclass` et `isinstance` n'est pas très fréquente mais elle permet par exemple d'écrire une fonction qui peut prendre en entrée des types variables.

```
def fonction_somme_list (ens) :
    r = "list "
    for e in ens : r += e
    return r

def fonction_somme_dict (ens) :
    r = "dict "
    for k,v in ens.items () : r += v
    return r

def fonction_somme (ens) :
    if   isinstance (ens, dict) : return fonction_somme_dict (ens)
    elif isinstance (ens, list) : return fonction_somme_list (ens)
    else                        : return "erreur"

li = ["un", "deux", "trois"]
di = {1:"un", 2:"deux", 3:"trois"}
tu = ("un", "deux", "trois")
print fonction_somme (li)  # affiche list undeuxtrois
print fonction_somme (di)  # affiche dict undeuxtrois
print fonction_somme (tu)  # affiche erreur
```

L'avantage est d'avoir une seule fonction capable de s'adapter à différents type de variables, y compris des types créés par un programmeur en utilisant les classes.

## 4.9 Compilation de classes

La compilation de classe fonctionne de manière similaire à celle de la compilation de fonctions (voir paragraphe 3.4.14.2, page 85). Il s'agit de définir une classe sous forme de chaîne de caractères puis d'appeler la fonction `compile` pour ajouter cette classe au programme et s'en servir. L'exemple suivant reprend deux classes décrites au paragraphe 4.8.3. Elles sont incluses dans une chaîne de caractères, compilées puis incluses au programme (fonction `exec`).

```
s = """class carre :
    def __init__( self, a) : self.a = a
    def __str__ (self)     : return "carre " + str (self.a)

class rectangle (carre):
    def __init__(self,a,b) :
        carre.__init__(self, a)
        self.b = b
    def __str__ (self) :
        return "rectangle " + str (self.a) +  " x " +  str (self.b)"""

obj = compile(s,"","exec")       # code à compiler
exec (obj)                       # classes incorporées au programme

r = rectangle (3,4)
print r  # affiche rectangle 3 x 4
```

```
c = carre (5)
print c  # affiche carre 5
```

Comme toute fonction, la fonction `compile` génère une exception lorsque la chaîne de caractères contient une erreur. Le programme qui suit essaye de compiler une chaîne de caractères confondant `self` et `seilf`.

```
# coding: latin-1
"""erreur de compilation incluses dans le code inséré dans la
chaîne de caractère s"""
s = """class carre :
    def __init__( self, a) :
        seilf.a = a   # erreur de compilation
    def __str__ (self) :
        return "carre " + str (self.a) """

obj = compile(s,"variable s","exec")   # code à compiler
exec (obj)                             # classes incorporées au programme

c = carre (5)
print c  # affiche carre 5
```

L'exécution de ce programme affiche le message suivant :

```
Traceback (most recent call last):
  File "C:\temp\cours.py", line 14, in -toplevel-
    c = carre (5)
  File "variable s", line 3, in __init__
NameError: global name 'seilf' is not defined
```

Le message d'erreur est le même que pour un programme ne faisant pas appel à la fonction `compile` à ceci près que le fichier où a lieu l'erreur est `variable s` qui est le second paramètre envoyé à la fonction `compile`. Le numéro de ligne fait référence à la troisième ligne de la chaîne de caractères `s` et non à la troisième ligne du programme.

## 4.10 Constructions classiques

### 4.10.1 Héritage

Le premier exemple est classique puisqu'il reprend le programme du paragraphe 3.7.10 de la page 92 pour le réécrire avec des classes et éviter de passer des fonctions comme paramètre d'une autre fonction. La première classe définit le module des suivantes. La méthode `calcul` n'accepte qu'un seul paramètre `x` mais pourrait également prendre en compte des constantes si celles-ci sont renseignées via le constructeur de la classe. C'est l'avantage de cette solution déjà illustrée par les pièces normales et truquées.

```
class Fonction :
    def calcul (self, x) : pass
    def calcul_n_valeur (self, l) :
        res = [ self.calcul (i) for i in l ]
        return res
```

```
class Carre (Fonction) :
    def calcul (self, x) : return x*x

class Cube (Fonction) :
    def calcul (self, x) : return x*x*x

l = [0,1,2,3]
print l   # affiche [0, 1, 2, 3]

l1 = Carre ().calcul_n_valeur (l)  # l1 vaut [0, 1, 4, 9]
l2 = Cube () .calcul_n_valeur (l)  # l2 vaut [0, 1, 8, 27]
```

La version suivante mélange héritage et méthodes envoyées comme paramètre à une fonction. Il est préférable d'éviter cette construction même si elle est très utilisée par les interfaces graphiques. Elle n'est pas toujours transposable dans tous les langages de programmation tandis que le programme précédent aura un équivalent dans tous les langages objet.

```
class Fonction :
    def calcul (self, x) : pass
class Carre (Fonction) :
    def calcul (self, x) : return x*x
class Cube (Fonction) :
    def calcul (self, x) : return x*x*x

def calcul_n_valeur (l,f):
    res = [ f(i) for i in l ]
    return res

l = [0,1,2,3]
l1 = calcul_n_valeur (l, Carre ().calcul) # l1 vaut [0, 1, 4, 9]
l2 = calcul_n_valeur (l, Cube ().calcul)  # l2 vaut [0, 1, 8, 27]
```

### 4.10.2 Deux lignées d'héritages

C'est une configuration qui arrive fréquemment lorsqu'on a d'un côté différentes structures pour les mêmes données et de l'autre différents algorithmes pour le même objectif. Par exemple, une matrice peut être définie comme une liste de listes ou un dictionnaire de 2-uples [6]. La multiplication de deux matrices peut être une multiplication classique ou celle de *Strassen* [7].

La question est comment modéliser ces deux structures et ces deux multiplications sachant que les quatre paires *structure* - *algorithme* doivent fonctionner. On pourrait simplement créer deux classes faisant référence aux deux structures différentes et à l'intérieur de chacune d'entre elles, avoir deux méthodes de multiplication. Néanmoins, si une nouvelle structure ou un nouvel algorithme apparaît, la mise à jour peut être fastidieuse.

Il est conseillé dans ce cas d'avoir quatre classes et de définir une interface d'échanges communes. L'algorithme de multiplication ne doit pas savoir quelle structure il ma-

6. voir exercice 10.2, page 242

7. L'algorithme de multiplication de deux matrices de *Strassen* est plus rapide qu'une multiplication classique pour de grandes matrices. Son coût est en $O(n^{\frac{\ln 7}{\ln 2}}) \sim O(n^{2,807})$ au lieu de $O(n^3)$.

nipule : il doit y accéder par des méthodes. De cette manière, c'est la classe qui indique l'algorithme choisi et non une méthode. Ajouter un troisième algorithme ou une troisième structure revient à ajouter une classe : l'interface d'échange ne change pas. Le programme pourrait suivre le schéma qui suit.

```
class Matrice :
    def __init__ (self,lin,col,coef):
        self.lin, self.col = lin, col

    # interface d'échange
    def get_lin () : return self.lin
    def get_col () : return self.col
    def __getitem__(self,i,j): pass
    def __setitem__(self,i,j,v): pass
    def get_submat(self, i1,j1,i2,j2): pass
    def set_submat(self, i1,j1,mat): pass
    # fin de l'interface d'échange

    def trace (self) :
        t = 0
        for i in xrange (0, self.lin):
            t += self (i,i)
        return t

class MatriceList (Matrice) :
    def __init__ (self,lin,col,coef):
        Matrice.__init__ (self, \
                      lin, col, coef)
        #...

    def __getitem__ (self, i,j) : #...
    def __setitem__ (self, i,j, v) : #...
    def get_submat(self, i1,j1,i2,j2): #...
    def set_submat(self, i1,j1,mat): #...

class MatriceDict (Matrice) :
    def __init__ (self,lin,col,coef):
        Matrice.__init__ (self, \
                       lin, col, coef)
        #...

    def __getitem__ (self, i,j) : #...
    def __setitem__ (self, i,j, v) : #...
    def get_submat(self, i1,j1,i2,j2): #...
    def set_submat(self, i1,j1,mat): #...
```

```
class Produit :
    def calcul (self, mat1, mat2):
        pass

class ProduitClassique (Produit) :
    def calcul (self, mat1, mat2):
        #...
        return

class ProduitStrassen (Produit) :
    def calcul (self, mat1,mat2):
        #...
        return
```

Cette construction autorise même la multiplication de matrices de structures différentes. Très répandue, cette architecture est souvent plus coûteuse au moment de la conception car il faut bien penser l'interface d'échange mais elle l'est beaucoup moins par la suite. Il existe d'autres assemblages de classes assez fréquents, regroupés sous le terme de *Design Patterns*. Pour peu que ceux-ci soient connus de celui qui conçoit un programme, sa relecture et sa compréhension en sont facilitées si les commentaires font mention du *pattern* utilisé.

| | |
|---|---|
| `__cmp__(self,x)` | Retourne un entier égale à -1, 0, 1, chacune de ces valeurs étant associés respectivement à : `self < x`, `self == x`, `self > x`. Cet opérateur est appelé par la fonction `cmp`. |
| `__str__(self)` | Convertit un objet en une chaîne de caractère qui sera affichée par la fonction `print` ou obtenu avec la fonction `str`. |
| `__contains__(self,x)` | Retourne `True` ou `False` selon que `x` appartient à `self`. Le mot-clé `in` renvoie à cet opérateur. En d'autres termes, `if x in obj` : appelle `obj.__contains__(x)`. |
| `__len__(self)` | Retourne le nombre d'élément de `self`. Cet opérateur est appelé par la fonction `len`. |
| `__abs__(self)` | Cet opérateur est appelé par la fonction `abs`. |
| `__getitem__(self,i)` | Cet opérateur est appelé lorsqu'on cherche à accéder à un élément de l'objet `self` d'indice `i` comme si c'était une liste. Si l'indice `i` est incorrect, l'exception `IndexError` doit être levée. |
| `__setitem__(self,i,v)` | Cet opérateur est appelé lorsqu'on cherche à affecter une valeur `v` à un élément de l'objet `self` d'indice `i` comme si c'était une liste ou un dictionnaire. Si l'indice `i` est incorrect, l'exception `IndexError` doit être levée. |
| `__delitem__(self,i)` | Cet opérateur est appelé lorsqu'on cherche à supprimer l'élément de l'objet `self` d'indice `i` comme si c'était une liste ou un dictionnaire. Si l'indice `i` est incorrect, l'exception `IndexError` doit être levée. |
| `__int__(self)`<br>`__float__(self)`<br>`__complex__(self)` | Ces opérateurs implémente la conversion de l'instance `self` en entier, réel ou complexe. |
| `__add__(self,x)`<br>`__div__(self,x)`<br>`__mul__(self,x)`<br>`__sub__(self,x)`<br>`__pow__(self,x)`<br>`__lshift__(self,x)`<br>`__rshift__(self,x)` | Opérateurs appelés pour les opérations `+`, `/`, `*`, `-`, `**`, `<<`, `>>` |
| `__iadd__(self,x)`<br>`__idiv__(self,x)`<br>`__imul__(self,x)`<br>`__isub__(self,x)`<br>`__ipow__(self,x)`<br>`__ilshift__(self,x)`<br>`__irshift__(self,x)` | Opérateurs appelés pour les opérations `+=`, `/=`, `*=`, `-=`, `**=`, `«=`, `»=` |

**Table 4.1 :** *Opérateurs ou fonctions spéciales les plus utilisées. La page http://docs.python.org/reference/datamodel.html#special-method-names (voir également http://docs.python.org/library/operator.html#mapping-operators-to-functions) recense toutes les fonctions spéciales qu'il est possible de redéfinir pour une classe.*

# Chapitre 5

# Exceptions

---

Le petit programme suivant déclenche une erreur parce qu'il effectue une division par zéro.

```
x = 0
y = 1.0 / x
```

Il déclenche une erreur ou ce qu'on appelle une *exception*.

```
Traceback (most recent call last):
  File "cours.py", line 2, in ?
    y = 1.0 / x
ZeroDivisionError: float division
```

Le mécanisme des exceptions permet au programme de "rattraper" les erreurs, de détecter qu'une erreur s'est produite et d'agir en conséquence afin que le programme ne s'arrête pas.

## 5.1 Principe des exceptions

### 5.1.1 Attraper toutes les erreurs

> **Définition 5.1 : exception**
> Une exception est un objet qui indique que le programme ne peut continuer son exécution.

On décide par exemple qu'on veut rattraper toutes les erreurs du programme et afficher un message d'erreur. Le programme suivant appelle la fonction `inverse` qui retourne l'inverse d'un nombre.

```
def inverse (x):
    y = 1.0 / x
    return y

a = inverse (2)
print a
b = inverse (0)
print b
```

Lorsque le paramètre `x = 0`, le programme effectue une division par zéro et déclenche une erreur qui paraît différente de la première d'après la longueur du message. C'est

pourtant la même erreur : cette liste correspond en fait à ce qu'on appelle la pile d'appels [1]. Si l'erreur se produit dans une fonction elle-même appelée par une autre... la pile d'appel permet d'obtenir la liste de toutes les fonctions pour remonter jusqu'à celle où l'erreur s'est produite.

```
Traceback (most recent call last):
  File "cours.py", line 8, in ?
    b = inverse (0)
  File "cours.py", line 3, in inverse
    y = 1.0 / x
ZeroDivisionError: float division
```

Afin de rattraper l'erreur, on insère le code susceptible de produire une erreur entre les mots clés `try` et `except`.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    a = inverse (2)
    print a
    b = inverse (0)  # déclenche une exception
    print b
except :
    print "le programme a déclenché une erreur"
```

Le programme essaye d'exécuter les quatre instructions incluses entre les instructions `try` et `except`. Si une erreur se produit, le programme exécute alors les lignes qui suivent l'instruction `except`. L'erreur se produit en fait à l'intérieur de la fonction `inverse` mais celle-ci est appelée à l'intérieur d'un code "protégé" contre les erreurs. Le programme précédent affiche les deux lignes suivantes.

```
0.5
le programme a déclenché une erreur
```

Il est aussi possible d'ajouter une clause `else` qui sert de préfixe à une liste d'instructions qui ne sera exécutée que si aucune exception n'est déclenchée.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    print inverse (2)  # pas d'erreur
    print inverse (1)  # pas d'erreur non plus
except :
    print "le programme a déclenché une erreur"
else :
    print "tout s'est bien passé"
```

Ce dernier programme ne déclenche aucune exception et affiche les lignes suivantes.

1. *call stack* en anglais

```
0.5
1.0
tout s'est bien passé
```

Pour résumer, la syntaxe suivante permet d'attraper toutes les erreurs qui se produisent pendant l'exécution d'une partie du programme. Cette syntaxe permet en quelque sorte de protéger cette partie du programme contre les erreurs.

**syntaxe 5.2 : exception**

```
try :
    # ... instructions à protéger
except :
    # ... que faire en cas d'erreur
else :
    # ... que faire lorsque aucune erreur n'est apparue
```

Toute erreur déclenchée alors que le programme exécute les instructions qui suivent le mot-clé `try` déclenche immédiatement l'exécution des lignes qui suivent le mot-clé `except`. Dans le cas contraire, le programme se poursuit avec l'exécution des lignes qui suivent le mot-clé `else`. Cette dernière partie est facultative, la clause `else` peut ou non être présente. Dans tous les cas, l'exécution du programme ne s'arrête pas.

**Remarque 5.3 : plusieurs erreurs**

Lorsqu'une section de code est protégée contre les exceptions, si elle contient plusieurs erreurs, son exécution s'arrête à la première des erreurs découvertes. Par exemple, dès la première erreur qui correspond au calcul d'une puissance non entière d'un nombre négatif, l'exécution du programme suivant est dirigée vers l'instruction qui suit le mot-clé `except`.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    print (-2.1) ** 3.1  # première erreur
    print inverse (2)
    print inverse (0)    # seconde erreur
except :
    print "le programme a déclenché une erreur"
```

### 5.1.2 Obtenir le type d'erreur, attraper un type d'exception

Parfois, plusieurs types d'erreurs peuvent être déclenchés à l'intérieur d'une portion de code protégée. Pour avoir une information sur ce type, il est possible de récupérer une variable de type `Exception`.

```
def inverse (x):
    y = 1.0 / x
    return y
```

```
try :
    print inverse (2)
    print inverse (0)
except Exception, exc:
    print "exception de type ", exc.__class__
         # affiche exception de type  exceptions.ZeroDivisionError
    print "message ", exc
         # affiche le message associé à l'exception
```

Le programme précédent récupère une exception sous la forme d'une variable appelée `exc`. Cette variable est en fait une instance d'une classe d'erreur, `exc.__class__` correspond au nom de cette classe. A l'aide de la fonction `isinstance`, il est possible d'exécuter des traitements différents selon le type d'erreur.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    print (-2.1) ** 3.1  # première erreur
    print inverse (2)
    print inverse (0)    # seconde erreur
except Exception, exc:
    if isinstance (exc, ZeroDivisionError) :
        print "division par zéro"
    else :
        print "erreur insoupçonnée : ", exc.__class__
        print "message ", exc
```

L'exemple précédent affiche le message qui suit parce que la première erreur intervient lors du calcul de $(-2.1) * *3.1$.

```
erreur insoupçonnée :  exceptions.ValueError
```

Une autre syntaxe plus simple permet d'attraper un type d'exception donné en accolant au mot-clé `except` le type de l'exception qu'on désire attraper. L'exemple précédent est équivalent au suivant mais syntaxiquement différent.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    print (-2.1) ** 3.1
    print inverse (2)
    print inverse (0)
except ZeroDivisionError:
    print "division par zéro"
except Exception, exc:
    print "erreur insoupçonnée : ", exc.__class__
    print "message ", exc
```

Cette syntaxe obéit au schéma qui suit.

**syntaxe 5.4 : exception d'un type donné**

```
try :
    # ... instructions à protéger
except type_exception_1 :
    # ... que faire en cas d'erreur de type type_exception_1
except type_exception_i :
    # ... que faire en cas d'erreur de type type_exception_i
except type_exception_n :
    # ... que faire en cas d'erreur de type type_exception_n
except :
    # ... que faire en cas d'erreur d'un type différent de tous
    #     les précédents types
else :
    # ... que faire lorsque une erreur aucune erreur n'est apparue
```

Toute erreur déclenchée alors que le programme exécute les instructions qui suivent le mot-clé `try` déclenche immédiatement l'exécution des lignes qui suivent un mot-clé `except`. Le programme compare le type d'exception aux types `type_exception_1` à `type_exception_n`. S'il existe une correspondance alors ce sont les instructions de la clause `except` associée qui seront exécutées et uniquement ces instructions. La dernière clause `except` est facultative, elle est utile lorsque aucun type de ceux prévus ne correspond à l'exception générée. La clause `else` est aussi facultative. Si la dernière clause `except` n'est pas spécifiée et que l'exception déclenchée ne correspond à aucune de celle listée plus haut, le programme s'arrête sur cette erreur à moins que celle-ci ne soit attrapée plus tard.

Le langage *Python* propose une liste d'exceptions standards (voir table 5.1, page 149). Lorsqu'une erreur ne correspond pas à l'une de ces exceptions, il est possible de créer une exception propre à un certain type d'erreur (voir paragraphe 5.2). Lorsqu'une fonction ou une méthode déclenche une exception non standard, généralement, le commentaire qui lui est associé l'indique.

### 5.1.3 Lancer une exception

Lorsqu'une fonction détecte une erreur, il lui est possible de déclencher une exception par l'intermédiaire du mot-clé `raise`. La fonction `inverse` compare `x` à `0` et déclenche l'exception `ValueError` si `x` est nul. Cette exception est attrapée plus bas.

```
def inverse (x):
    if x == 0 :
        raise ValueError
    y = 1.0 / x
    return y

try :
    print inverse (0)  # erreur
except ValueError:
    print "erreur"
```

Il est parfois utile d'associer un message à une exception afin que l'utilisateur ne soit pas perdu. Le programme qui suit est identique au précédent à ceci près qu'il associe à l'exception `ValueError` qui précise l'erreur et mentionne la fonction où elle s'est produite. Le message est ensuite intercepté plus bas.

```
def inverse (x):
    if x == 0 :
        raise ValueError ("valeur nulle interdite, fonction inverse")
    y = 1.0 / x
    return y

try :
    print inverse (0)  # erreur
except ValueError, exc:
    print "erreur, message : ", exc
```

Le déclenchement d'une exception suit la syntaxe suivante.

> **syntaxe 5.5 : exception,** `raise`
>
> `raise exception_type (message) :`
>
> Cette instruction lance l'exception `exception_type` associée au message `message`. Le message est facultatif, lorsqu'il n'y en a pas, la syntaxe se résume à `raise exception_type`.

Et pour attraper cette exception et le message qui lui est associé, il faut utiliser la syntaxe 5.4 décrite au paragraphe précédent.

### 5.1.4 Héritage et exception

L'instruction `help(ZeroDivisionError)` retourne l'aide associée à l'exception `ZeroDivisionError`. Celle-ci indique que l'exception `ZeroDivisionError` est en fait un cas particulier de l'exception `ArithmeticError`, elle-même un cas particulier de `StandardError`.

```
class ZeroDivisionError(ArithmeticError)
 |  Second argument to a division or modulo operation was zero.
 |
 |  Method resolution order:
 |      ZeroDivisionError
 |      ArithmeticError
 |      StandardError
 |      Exception
```

Toutes les exceptions sont des cas particuliers de l'exception de type `Exception`. C'est pourquoi l'instruction `except Exception, e` : attrape toutes les exceptions. L'instruction `except ArithmeticError` : attrape toutes les erreurs de type `ArithmeticError`, ce qui inclut les erreurs de type `ZeroDivisionError`. Autrement dit, toute exception de type `ZeroDivisionError` est attrapée par les instructions suivantes :

- `except ZeroDivisionError` :
- `except ArithmeticError` :
- `except StandardError` :

– except Exception :

Plus précisément, chaque exception est une classe qui dérive directement ou indirectement de la classe Exception. L'instruction except ArithmeticError : par exemple attrape toutes les exceptions de type ArithmeticError et toutes celles qui en dérivent comme la classe ZeroDivisionError (voir également le paragraphe 5.2).

### 5.1.5 Instructions try, except imbriquées

Comme pour les boucles, il est possible d'imbriquer les portions protégées de code les unes dans les autres. Dans l'exemple qui suit, la première erreur est l'appel à une fonction non définie, ce qui déclenche l'exception NameError.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    try :
        print inverses (0)  # fonction inexistante --> exception NameError
        print inverse (0)   # division par zéro --> ZeroDivisionError
    except NameError:
        print "appel à une fonction non définie"
except ZeroDivisionError, exc:
    print "erreur ", exc
```

En revanche, dans le second exemple, les deux lignes print inverse(0) et print inverses(0) ont été permutées. La première exception déclenchée est la division par zéro. La première clause except n'interceptera pas cette erreur puisqu'elle n'est pas du type recherché.

```
def inverse (x):
    y = 1.0 / x
    return y

try :
    try :
        print inverse (0)   # division par zéro --> ZeroDivisionError
        print inverses (0)  # fonction inexistante --> exception NameError
    except NameError:
        print "appel à une fonction non définie"
except ZeroDivisionError, exc:
    print "erreur ", exc
```

Une autre imbrication possible est l'appel à une fonction qui inclut déjà une partie de code protégée. L'exemple suivant appelle la fonction inverse qui intercepte les exceptions de type ZeroDivisionError pour retourner une grande valeur lorsque x = 0. La seconde exception générée survient lors de l'appel à la fonction inverses qui déclenche l'exception NameError, elle aussi interceptée.

```
def inverse (x):
    try :
        y = 1.0 / x
    except ZeroDivisionError, exc:
```

```
        print "erreur ", exc
        if x > 0 : return 1000000000
        else : return -1000000000
    return y

try :
    print inverse (0)   # division par zéro    --> la fonction inverse sait gérer
    print inverses (0)  # fonction inexistante --> exception NameError
except NameError:
    print "appel à une fonction non définie"
```

## 5.2 Définir ses propres exceptions

### 5.2.1 Dériver une classe d'exception

Pour définir sa propre exception, il faut créer une classe qui dérive d'une classe d'exception existante (voir paragraphe 5.1.4), par exemple, la classe `Exception`. L'exemple suivant crée une exception `AucunChiffre` qui est lancée par la fonction `conversion` lorsque la chaîne de caractères qu'elle doit convertir ne contient pas que des chiffres.

```
class AucunChiffre (Exception) :
    """chaîne de caractères contenant aussi autre chose que des chiffres"""

def conversion (s) :
    """conversion d'une chaîne de caractères en entier"""
    if not s.isdigit () :
        raise AucunChiffre (s)
    return int (s)

try :
    s = "123a"
    print s, " = ", conversion (s)
except AucunChiffre, exc :
    # on affiche ici le commentaire associé à la classe d'exception
    # et le message associé
    print AucunChiffre.__doc__, " : ", exc
```

**Remarque 5.6 : exception et opérateur `__str__`**
En redéfinissant l'opérateur `__str__` d'une exception, il est possible d'afficher des messages plus explicites avec la seule instruction `print`.

```
class AucunChiffre (Exception) :
    """chaîne de caractères contenant aussi autre chose que des chiffres"""
    def __str__ (self) :
        return self.__doc__ + " " + Exception.__str__ (self)
```

### 5.2.2 Personnalisation d'une classe d'exception

Il est parfois utile qu'une exception contienne davantage d'informations qu'un simple message. L'exemple suivant reprend l'exemple du paragraphe précédent. L'exception

`AucunChiffre` inclut cette fois-ci un paramètre supplémentaire contenant le nom de la fonction où l'erreur a été déclenchée.

La classe `AucunChiffre` possède dorénavant un constructeur qui doit recevoir deux paramètres : une valeur et un nom de fonction. L'exception est levée à l'aide de l'instruction `raise AucunChiffre (s,"conversion")` qui regroupe dans un T-uple les paramètres à envoyer à l'exception.

L'opérateur `__str__` a été modifié de façon à ajouter ces deux informations dans le message associé à l'exception. Ainsi, l'instruction `print exc` présente à l'avant dernière ligne de cet exemple affiche un message plus complet.

```
class AucunChiffre (Exception) :
    """chaîne de caractères contenant aussi autre chose que des chiffres"""
    def __init__(self, s, f = "") :
        Exception.__init__(self, s)
        self.s = s
        self.f = f
    def __str__(self) :
        return """exception AucunChiffre, depuis la fonction """ + self.f + \
                  " avec le paramètre " + self.s

def conversion (s) :
    """conversion d'une chaîne de caractères en entier"""
    if not s.isdigit () :
        raise AucunChiffre (s, "conversion")
    return int (s)

try :
    s = "123a"
    i = conversion (s)
    print s, " = ", i
except AucunChiffre, exc :
    print exc
    print "fonction : ", exc.f
```

Etant donné que le programme déclenche une exception dans la section de code protégée, les deux derniers affichages sont les seuls exécutés correctement. Ils produisent les deux lignes qui suivent.

```
exception AucunChiffre, depuis la fonction conversion avec le paramètre 123a
fonction :  conversion
```

## 5.3 Exemples d'utilisation des exceptions

### 5.3.1 Les itérateurs

Les itérateurs sont des outils qui permettent de parcourir des objets qui sont des ensembles, comme une liste, un dictionnaire. Ils fonctionnent toujours de la même manière. L'exemple déjà présenté au paragraphe 4.4.2, page 106) et repris en partie ici définit une classe contenant trois coordonnées, ainsi qu'un itérateur permettant de parcourir ces trois coordonnées. Arrivée à la troisième itération, l'exception `StopIteration` est déclenchée. Cette exception indique à une boucle `for` de s'arrêter.

```
class point_espace:

    #...

    class class_iter:
        def __init__ (self,ins):
            self._n   = 0
            self._ins = ins
        def __iter__ (self) :
            return self
        def next (self):
            if self._n <= 2:
                v = self._ins [self._n]
                self._n += 1
                return v
            else :
                raise StopIteration

    def __iter__(self):
        return point_espace.class_iter (self)

# ...
```

Cet exemple montre seulement que les exceptions n'interviennent pas seulement lors d'erreurs mais font parfois partie intégrante d'un algorithme.

### 5.3.2 Exception ou valeur aberrante

Sans exception, une solution pour indiquer un cas de mauvaise utilisation d'une fonction est de retourner une valeur aberrante. Retourner -1 pour une fonction dont le résultat est nécessairement positif est une valeur aberrante. Cette convention permet de signifier à celui qui appelle la fonction que son appel n'a pu être traité correctement. Dans l'exemple qui suit, la fonction `racine_carree` retourne un couple de résultats, `True` ou `False` pour savoir si le calcul est possible, suivi du résultat qui n'a un sens que si `True` est retournée en première valeur.

```
def racine_carree(x) :
    if x < 0 : return False, 0
    else : return True, x ** 0.5
print racine_carree (-1)  # (False, 0)
print racine_carree (1)   # (True, 1.0)
```

Plutôt que de compliquer le programme avec deux résultats ou une valeur aberrante, on préfère souvent déclencher une exception, ici, `ValueError`. La plupart du temps, cette exception n'est pas déclenchée. Il est donc superflu de retourner un couple plutôt qu'une seule valeur.

```
def racine_carree(x) :
    if x < 0 : raise ValueError ("valeur négative")
    return x ** 0.5
print racine_carree (-1)  # déclenche une exception
print racine_carree (1)
```

### 5.3.3 Le piège des exceptions

Ce paragraphe évoque certains problèmes lorsqu'une exception est levée. L'exemple utilise les fichiers décrits au chapitre 7. Lorsqu'une exception est levée à l'intérieur d'une fonction, l'exécution de celle-ci s'interrompt. Si l'exception est attrapée, le programme continue sans problème ; les objets momentanément créés seront détruits par le garbage collector. Il faut pourtant faire attention dans le cas par exemple où l'exception est levée alors qu'un fichier est ouvert : il ne sera pas fermé.

```
# coding: latin-1
def ajoute_resultat_division (nom, x, y) :
    """ajoute le résultat de la division x/y au fichier nom"""
    f = open (nom, "a")
    f.write (str (x) + "/" + str (y) + "= ")
    f.write ( str ((float (x)/y)) + "\n" )     # exception si y == 0
    f.close ()

for i in range (0, 5) :
    try :
        print str (i-1) + "/" + str (i-2)
        ajoute_resultat_division ("essai.txt", i-1,i-2)
    except Exception, e : print "erreur avec i = ", i, ",", e
```

Les écritures dans le fichier se font en mode ajout ("`a`"), le fichier "`essai.txt`" contiendra tout ce qui aura été écrit.

```
-1/-2
0/-1
1/0
erreur avec i =  2 , float division
2/1
3/2
```

*côté affichage*

```
-1/-2= 0.5
0/-1= 0.0
2/1= 2.0
3/2= 1.5
1/0=
```

*côté fichier*

Il n'y pas de surprise du côté des affichages produit par l'instruction `print`. `1/0` précède `2/1`. La différence vient de la dernière ligne du fichier qui intervient de façon surprenante en dernière position. Lorsque `y==0`, la fonction s'interrompt avant de fermer le fichier. C'est le garbage collector qui le fermera bien après l'appel suivant qui ouvre à nouveau le fichier pour écrire `2/1= 2.0`.

| | |
|---|---|
| `Exception` | Elle n'est en principe jamais explicitement générée par les fonctions *Python* mais elle permet d'attraper toutes les exceptions car toutes dérivent de la classe `Exception`. |
| `AttributeError` | Une référence à un attribut inexistant ou une affectation a échoué. |
| `ArithmeticError` | Une opération arithmétique a échoué. |
| `FloatingPointError` | Une opération avec des nombres réels a échoué. |
| `IOError` | Une opération concernant les entrées/sorties (Input/Output) a échoué. Cette erreur survient par exemple lorsqu'on cherche à lire un fichier qui n'existe pas. |
| `ImportError` | Cette erreur survient lorsqu'on cherche à importer un module qui n'existe pas (voir chapitre 6). |
| `IndentationError` | L'interpréteur ne peut interpréter une partie du programme à cause d'un problème d'indentation. Il n'est pas possible d'exécuter un programme mal indenté mais cette erreur peut se produire lors de l'utilisation de la fonction `compile` (voir le paragraphe 3.4.14.2, page 85). |
| `IndexError` | On utilise un index erroné pour accéder à un élément d'une liste, d'un dictionnaire ou de tout autre tableau. |
| `KeyError` | Une clé est utilisée pour accéder à un élément d'un dictionnaire dont elle ne fait pas partie. |
| `NameError` | On utilise une variable, une fonction, une classe qui n'existe pas. |
| `OverflowError` | Un calcul sur des entiers ou des réels dépasse les capacités de codage des entiers ou des réels. |
| `StopIteration` | Cette exception est utilisée pour signifier qu'un itérateur atteint la fin d'un ensemble, un tableau, un dictionnaire. |
| `TypeError` | Erreur de type, une fonction est appliquée sur un objet qu'elle n'est pas censée manipuler. |
| `UnicodeError` | Il faut parfois convertir des chaînes de caractères de type `unicode` en chaînes de caractères de type `str` et réciproquement. Cette exception intervient lorsque la conversion n'est pas possible (voir également le paragraphe 7.8 page 191). |
| `ValueError` | Cette exception survient lorsqu'une valeur est inappropriée pour une certaine opération, par exemple, l'obtention du logarithme d'un nombre négatif. |
| `ZeroDivisionError` | Cette exception survient pour une division par zéro. |

**Table 5.1 :** *Exceptions standard les plus couramment utilisées. Ces exceptions sont définies par le langage Python. Il n'est pas nécessaire d'inclure un module. La page http://docs.python.org/library/exceptions.html recense toutes les exceptions prédéfinies.*

# Chapitre 6

# Modules

---

Il est souvent préférable de répartir le code d'un grand programme sur plusieurs fichiers. Parmi tous ces fichiers, un seul est considéré comme fichier principal, il contient son point d'entrée, les premières instructions exécutées. Les autres fichiers sont considérés comme des modules, en quelque sorte, des annexes qui contiennent tout ce dont le fichier principal a besoin.

## 6.1 Modules et fichiers

### 6.1.1 Exemple

Cet exemple montre comment répartir un programme sur deux fichiers. Le premier est appelé *module* car il n'inclut pas le point d'entrée du programme.

> **Définition 6.1 : point d'entrée du programme**
> Le point d'entrée d'un programme est la première instruction exécutée par l'ordinateur lors de l'exécution de ce programme.

Cet exemple de module contient une fonction, une classe et une variable. Ces trois éléments peuvent être utilisés par n'importe quel fichier qui importe ce module. Le nom d'un module correspond au nom du fichier sans son extension.

**Fichier :** `module_exemple.py`

```
# coding: latin-1
"""exemple de module, aide associée"""

exemple_variable = 3

def exemple_fonction () :
    """exemple de fonction"""
    return 0

class exemple_classe :
    """exemple de classe"""
    def __str__ (self) :
        return "exemple_classe"
```

**Fichier :** `exemple.py`

```
import module_exemple

c = module_exemple.exemple_classe ()
print c
print module_exemple.exemple_fonction()
help (module_exemple)
```

Pour importer un module, il suffit d'insérer l'instruction `import nom_module` avant d'utiliser une des choses qu'il définit. Ces importations sont souvent regroupées au début du programme, elles sont de cette façon mises en évidence même s'il est possible de les faire n'importe où. L'exemple ci-dessus à droite importe le module défini

à gauche. Les modules commencent le plus souvent par une chaîne de caractères comme dans l'exemple précédent, celle-ci contient l'aide associée à ce module. Elle apparaît avec l'instruction `help(module_exemple)`.

Rien ne différencie les deux fichiers `module_exemple.py` et `exemple.py` excepté le fait que le second utilise des éléments définis par le premier. Dans un programme composé de plusieurs fichiers, un seul contient le point d'entrée et tous les autres sont des modules.

La syntaxe d'appel d'un élément d'un module est identique à celle d'une classe. On peut considérer un module comme une classe avec ses méthodes et ses attributs à la seule différence qu'il ne peut y avoir qu'une seule instance d'un même module. La répétition de l'instruction `import module_exemple` n'aura aucun effet : un module n'est importé que lors de la première instruction `import nom_module` rencontré lors de l'exécution du programme.

**Remarque 6.2 : fichier** `*.pyc`
L'utilisation d'un module qu'on a soi-même conçu provoque l'apparition d'un fichier d'extension `pyc`. Il correspond à la traduction du module en *bytecode* plus rapidement exploitable par l'interpréteur *Python*. Ce fichier est généré à chaque modification du module. Lorsqu'un module est importé, *Python* vérifie la date des deux fichiers d'extension `py` et `pyc`. Si le premier est plus récent, le second est recréé. Cela permet un gain de temps lorsqu'il y a un grand nombre de modules. Il faut faire attention lorsque le fichier source d'extension `py` est supprimé, il est alors encore possible de l'importer tant que sa version d'extension `pyc` est présente.

**Remarque 6.3 : recharger un module**
Le module `module_exemple` contient une variable `exemple_variable` peut être modifiée au cours de l'exécution du programme. Il est possible de revenir à sa valeur initiale en forçant *Python* à recharger le module grâce à la fonction `reload`.

```
import module_exemple
module_exemple.exemple_variable = 10
reload (module_exemple)
print module_exemple.exemple_variable   # affiche 3
```

### 6.1.2 Autres syntaxes

Il existe trois syntaxes différentes pour importer un module. La première est décrite au paragraphe précédent. Il en existe une autre qui permet d'affecter à un module un identificateur différent du nom du fichier dans lequel il est décrit. En ajoutant l'instruction `as` suivi d'un autre nom `alias`, le module sera désigné par la suite par l'identificateur `alias` comme le montre l'exemple suivant.

**Fichier :** `exemple2.py`

```
import module_exemple as alias

c = alias.exemple_classe ()
print c
print alias.exemple_fonction ()
help (alias)
```

Une autre syntaxe permet de se passer d'identificateur pour un module en utilisant le mot-clé `from`. En utilisant la syntaxe `from module import` $*$, tous les identificateurs (fonctions, classes, variables) sont directement accessibles sans les faire précéder d'un identificateur de module ainsi que le montre l'exemple suivant.

**Fichier :** `exemple3.py`

```
from module_exemple import *

c = exemple_classe ()
print c
print exemple_fonction ()
```

La partie `import` $*$ permet d'importer toutes les classes, attributs ou fonctions d'un module mais il est possible d'écrire `from module_exemple import exemple_class` pour n'importer que cette classe. La fonction `exemple_fonction` ne sera pas accessible. Toutefois, cette syntaxe est déconseillée. Elle réduit la longueur de certaines fonctions ou classes puisqu'il n'est plus besoin de faire apparaître le module d'où elle proviennent et cela ne permet plus de distinguer une classe ou une fonction définie dans ce fichier de celles définies dans un autre module importé.

Il existe une dernière syntaxe d'importation d'un module qui est utile quand on ne sait pas encore au moment d'écriture du programme le nom du module à importer. Celui-ci sera précisé à l'aide d'une chaîne de caractères au moyen de la fonction `__import__`.

**Fichier :** `exemple4.py`

```
alias = __import__ ("module_exemple")

c = alias.exemple_classe ()
print c
print alias.exemple_fonction ()
help (alias)
```

### 6.1.3 Nom d'un module

Le nom d'un module est défini par le nom du fichier sous lequel il est enregistré. Dans l'exemple du paragraphe précédent, le module avait pour nom de fichier `module_exemple.py`, le nom de ce module est donc `module_exemple`.

Néanmoins, ce module peut également être exécuté comme un programme normal. Si tel est le cas, son nom devient `__main__`. C'est pourquoi, les quelques lignes qui suivent apparaissent souvent. Elles ne sont exécutées que si ce fichier a pour nom `__main__`. Un seul fichier peut porter ce nom : celui qui contient le point d'entrée.

```
if __name__ == "__main__" :
    print "ce fichier est le programme principal"
```

Cette astuce est régulièrement utilisée pour tester les fonctions et classes définies dans un module. Etant donné que cette partie n'est exécutée que si ce fichier est le programme principal, ajouter du code après le test `if __name__ ==` "`__main__`" : n'a aucune incidence sur tout programme incluant ce fichier comme module.

### 6.1.4 Emplacement d'un module

Lorsque le module est placé dans le même répertoire que le programme qui l'utilise, l'instruction `import nom_module_sans_extension` suffit. Cette instruction suffit également si ce module est placé dans le répertoire `site – packages` présent dans le répertoire d'installation de *Python*. Si ce n'est pas le cas, il faut préciser à l'interpréteur *Python* où il doit chercher ce module :

```
import sys
sys.path.append (sys.path [0] + "/../common")
################ sys.path [0] = répertoire de ce programme
import nom_module
```

La variable `sys.path` contient les répertoires où *Python* va chercher les modules. Le premier d'entre eux est le répertoire du programme. Il suffit d'ajouter à cette liste le répertoire désiré, ici, un répertoire appelé `common` situé au même niveau que le répertoire du programme. A ce sujet, il est conseillé d'utiliser le plus souvent possible des chemins relatifs et non absolus [1]. De cette façon, on peut recopier le programme et ses modules à un autre endroit du disque dur sans altérer leur fonctionnement.

### 6.1.5 Ajouter un module en cours d'exécution

De la même façon que *Python* est capable d'inclure de nouvelles portions de code en cours d'exécution [2], il est également capable d'inclure en cours d'exécution des modules dont on ne connaît pas le nom au début de l'exécution. Cela s'effectue grâce à la fonction `__import__` déjà présentée ci-dessus. Néanmoins, cette fonction ne peut pas importer un module si celui-ci est désigné par un nom de fichier incluant son répertoire. Il faut d'abord déterminer le répertoire où est le module grâce à la fonction `split` du module `os.path`. Le programme suivant illustre cette possibilité en proposant une fonction qui importe un module connaissant le nom du fichier qui le contient.

```
# coding: latin-1
def import_fichier (module) :
    import os.path
    import sys
    if os.path.exists (module) :           # on teste l'existence du fichier
        folder,name = os.path.split (module)  # on obtient le répertoire du module
        if folder not in sys.path :
            sys.path.append (folder)       # on ajoute le répertoire dans la liste
                                           # des répertoires autorisés
        name = name.replace (".py", "")    # on enlève l'extension
        module = __import__ (name)         # on importe le module
        return module
    else :
        # si le fichier n'existe pas --> on lève une exception
        raise ImportError ("impossible d'importer le module " + module)
```

1. Depuis un répertoire courant, les chemins relatifs permettent de faire référence à d'autres répertoires sans avoir à prendre en compte leur emplacement sur le disque dur contrairement aux chemins absolus comme C:/Python26/python.exe.

2. grâce à la fonction `exec`

```
# on importe un module
mod = import_fichier (r"D:\Dupre\informatique\programme\corde.py")
# on affiche l'aide associée
help (mod)
```

### 6.1.6 Liste des modules importés

Le dictionnaire `modules` du module `sys` contient l'ensemble des modules importés. Le programme suivant affiche cette liste.

```
import sys
for m in sys.modules :
    print m, " " * (14 - len(str(m))), sys.modules [m]
```

```
os              <module 'os' from 'c:\python26\lib\os.pyc'>
os.path         <module 'ntpath' from 'c:\python26\lib\ntpath.pyc'>
re              <module 're' from 'c:\python26\lib\re.pyc'>
site            <module 'site' from 'c:\python26\lib\site.pyc'>
sys             <module 'sys' (built-in)>
types           <module 'types' from 'c:\python26\lib\types.pyc'>
...
```

Lorsque le programme stipule l'import d'un module, *Python* vérifie s'il n'est pas déjà présent dans cette liste. Dans le cas contraire, il l'importe. Chaque module n'est importé qu'une seule fois. La première instruction `import module_exemple` rencontrée introduit une nouvelle entrée dans le dictionnaire `sys.modules` :

```
module_exemple  <module 'module_exemple' from 'D:\python_cours\module_exemple.py'>
```

Le dictionnaire `sys.modules` peut être utilisé pour vérifier la présence d'un module ou lui assigner un autre identificateur. Un module est un objet qui n'autorise qu'une seule instance.

```
if "module_exemple" in sys.modules :
    m = sys.modules ["module_exemple"]
    m.exemple_fonction ()
```

### 6.1.7 Attributs communs à tout module

Une fois importés, tous les modules possèdent cinq attributs qui contiennent des informations comme leur nom, le chemin du fichier correspondant, l'aide associée.

| | |
|---|---|
| `__all__` | Contient toutes les variables, fonctions, classes du module. |
| `__builtins__` | Ce dictionnaire contient toutes les fonctions et classes inhérentes au langage *Python* utilisées par le module. |
| `__doc__` | Contient l'aide associée au module. |
| `__file__` | Contient le nom du fichier qui définit le module. Son extension est `pyc`). |
| `__name__` | Cette variable contient a priori le nom du module sauf si le module est le point d'entrée du programme auquel cas cette variable contient "`__main__`". |

Ces attributs sont accessibles si le nom du module est utilisé comme préfixe. Sans préfixe, ce sont ceux du module lui-même.

```
import os
print os.__name__, os.__doc__
if __name__ == "__main__" : print "ce fichier est le point d'entrée"
else : print "ce fichier est importé"
```

### 6.1.8 Arborescence de modules, paquetage

Lorsque le nombre de modules devient conséquent, il est parfois souhaitable de répartir tous ces fichiers dans plusieurs répertoires. Il faudrait alors inclure tous ces répertoires dans la liste `sys.path` du module `sys` ce qui paraît fastidieux. *Python* propose la définition de paquetage, ce dernier englobe tous les fichiers *Python* d'un répertoire à condition que celui-ci contienne un fichier `__init__.py` qui peut être vide. La figure 6.1 présente une telle organisation et l'exemple suivant explicite comment importer chacun de ces fichiers sans avoir à modifier les chemins d'importation.

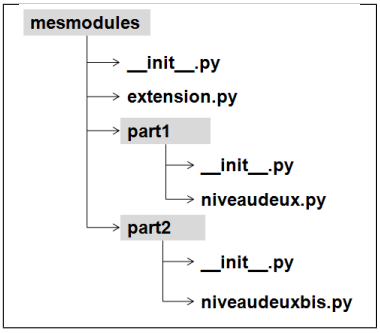



**Figure 6.1 :** *Arborescence de modules, un paquetage est défini par un ensemble de fichiers Python et d'un fichier* `__init__.py`*. Les répertoires sont grisées tandis que les fichiers apparaissent avec leur extension.*

```
import mesmodules.extension
import mesmodules.part1.niveaudeux
import mesmodules.part2.niveaudeuxbis
```

Lors de la première instruction `import mesmodules.extension`, le langage *Python* ne s'intéresse pas qu'au seul fichier `extension.py`, il exécute également le contenu du fichier `__init__.py`. Si cela est nécessaire, c'est ici qu'il faut insérer les instructions à exécuter avant l'import de n'importe quel module du paquetage.

## 6.2 Modules internes

*Python* dispose de nombreux modules préinstallés [3]. Cette liste est trop longue pour figurer dans ce document, elle est aussi susceptible de s'allonger au fur et à mesure du développement du langage *Python*. La table 6.2 (page 167) regroupe les modules les plus utilisés.

3. La page http://docs.python.org/modindex.html recense tous les modules disponibles avec *Python*.

Le programme suivant calcule l'intégrale de Monte Carlo de la fonction $f(x) = \sqrt{x}$ qui consiste à tirer des nombres aléatoires dans l'intervalle `a, b` puis à faire la moyenne des $\sqrt{x}$ obtenu.

```
import random  # import du module random : simulation du hasard
import math    # import du module math : fonctions mathématiques

def integrale_monte_carlo (a,b,f,n) :
    somme = 0.0
    for i in range (0,n) :
        x = random.random () * (b-a) + a
        y = f(x)
        somme += f(x)
    return somme / n

def racine (x) : return math.sqrt (x)

print integrale (0,1,racine,100000)
```

Le programme suivant utilise le module `urllib` pour télécharger le contenu d'une page et l'afficher.

```
def get_page_html (url):
    import urllib
    d = urllib.urlopen(url)
    res = d.read ()
    d.close ()
    return res

url = "http://www.lemonde.fr"
print get_page_html (url)
```

## 6.3 Modules externes

Les modules externes ne sont pas fournis avec *Python*, ils nécessitent une installation supplémentaire. Il serait impossible de couvrir tous les thèmes abordés par ces extensions. La simplicité d'utilisation du langage *Python* et son interfaçage facile avec le langage *C* contribue à sa popularité. Il permet de relier entre eux des projets conçus dans des environnements différents, dans des langages différents. Depuis les versions 2.3, 2.4 du langage *Python*, la plupart des modules externes sont faciles à installer, faciles à utiliser d'après les exemples que fournissent de plus en plus les sites Internet qui les hébergent. De plus, il s'écoule peu de temps entre la mise à disposition d'une nouvelle version du langage *Python* et la mise à jour du module pour cette version [4]. Le paragraphe 1.6 (page 24) donne une liste de modules utiles du point de vue d'un ingénieur généraliste.

Néanmoins, de nombreux modules ont été conçus pour un besoin spécifique et ne sont plus maintenus. Cela convient lors de l'écriture d'un programme qui remplit un

4. Ceci n'est pas tout-à-fait vrai pour la version 3.0 du langage dont les changements sont trop conséquents. Un changement du premier numéro de version indique souvent des changements majeurs.

besoin ponctuel. Pour une application plus ambitieuse, il est nécessaire de vérifier quelques informations comme la date de création, celle de la dernière mise à jour, la présence d'une documentation, une prévision pour la sortie de la future version, si c'est une personne lambda qui l'a conçu ou si c'est une organisation comme celle qui fournit le module `pygoogledesktop` [5]. C'est le cas de tous les modules présentés au pragraphe 1.6.

Concernant leur installation, certains modules externes comme `SciPy` peuvent être installés à l'aide d'un fichier exécutable sous *Windows*. Il suffit d'exécuter ce fichier pour pouvoir se servir du module par la suite dans un programme *Python*. Ce mode d'installation est disponible pour la plupart des modules de taille conséquente.

D'autres modules apparaissent compressés dans un fichier. Une fois décompressés, ils incluent un fichier `setup.py`. Le langage *Python* fournit une procédure d'installation standard : il suffit d'écrire quelques lignes dans une fenêtre de commande ouverte dans le répertoire où a été décompressé le fichier contenant le module à installer.

```
c:\python26\python setup.py install
```

Sous *Linux* et *Mac OS X*, cette ligne devient :

```
python setup.py install
```

## 6.4 *Python* et les autres langages

Il est impossible d'obtenir avec le même langage à la fois une grande vitesse de développement et une grande vitesse d'exécution. La façon dont est gérée la mémoire est un élément important qui explique l'appartenance d'un langage à l'une ou l'autre des deux catégories. *Python* inclut une fonctionnalité qu'on appelle le *garbage collector*. A chaque objet créé correspond un emplacement mémoire. Celui-ci peut être créé et détruit de façon explicite ou être totalement géré par le langage, ce que fait *Python*. Il est inutile de se soucier d'un objet dont on ne sert plus, il sera détruit automatiquement et la place mémoire qu'il utilisait sera de nouveau exploitable. Cette aisance de programmation suppose que le langage sache déterminer avec exactitude quand une variable n'est plus utilisée. Ceci signifie que *Python* mémorise des informations supplémentaires sur tous les objets créés en cours d'exécution pour assurer leur référencement. *Python* est à tout moment en mesure de connaître l'ensemble des noms de variables utilisés pour désigner le même objet. Le *garbage collector* alourdit un programme de façon cachée, il le ralentit tout en facilitant la conception des programmes.

C'est pour accélérer l'exécution que le langage *Python* est parfois associé à d'autres langages. Le programme final écrit en *Python* utilise des fonctionnalités haut niveau codées dans un autre langage plus rapide.

### 6.4.1 Langage *Java*

Il est possible d'utiliser des classes *Java* en *Python*. Cette possibilité ne sera pas plus détaillée ici car le langage *Java* est également un langage interprété même s'il

5. http://code.google.com/p/pythongoogledesktop/

est nettement plus rapide. Pour cela, il faut utiliser une version de l'interpréteur *Python* codée lui-même en *Java* ce qui est le cas de la version *Java* de *Python* : *Jython*[6].

### 6.4.2 Langage *C*

Il existe deux façons de construire un programme mêlant *Python* et langage *C*. La première méthode est la réalisation de modules *Python* écrits en *C* ou *C++*. A moins de devoir construire un module le plus efficace possible, il est peu conseillé de le faire directement car cela implique de gérer soi-même le référencement des variables créées en *C* ou *C++* et exportées vers *Python*[7]. Il est préférable d'utiliser des outils comme la librairie *SWIG*[8], ou encore la librairie *Boost Python*[9] qui simplifie le travail d'intégration. C'est cette dernière option qui est détaillée dans ce livre.

L'utilisation d'un autre langage est en effet indispensable lorsque l'exigence de rapidité est trop grande. Les modules scientifiques tels que `scipy` ou `numpy` ne répondent pas à tous les besoins. Il faut parfois développer soi-même des outils numériques en *C++*[10] même si la partie interface est assurée par *Python*.

L'autre façon de mélanger les deux langages est l'inclusion de petites parties *C* dans un programme *Python*. Ce point a déjà été évoqué au paragraphe 1.6.7 (page 27). Cela permet une optimisation essentiellement locale et numérique mais pas l'utilisation sous *Python* d'un projet existant programmé en *C++* ou *C*. Sous *Microsoft Windows*, il existe également une version du langage *IronPython*[11] développée pour ce système d'exploitation. L'intégration des DLL y est plus facile.

D'une manière générale, il est préférable de scinder nettement les parties d'un même programme qui s'occupent de calculs numériques de celles qui prennent en charge l'interface graphique. Construire un module écrit en *C++* pour gérer les calculs numériques est une façon drastique et efficace d'opérer cette scission.

## 6.5 *Boost Python*

La librairie *Boost*[12] est une librairie écrite en C++ qui a l'avantage d'être portable : elle est utilisable sur les systèmes d'exploitation les plus courants. Régulièrement mise à jour, elle propose une extension de la librairie *Standard Template Library (STL)* incluant notamment les expressions régulières, les graphes, les threads, ... *Boost Python* est une partie de cette librairie qui se présente également sous forme de *template*[13].

6. http://www.jython.org/Project/
7. Il faut tenir compte dans le module en *C* des opérations nécessaires au garbage collector.
8. http://www.swig.org/
9. http://www.boost.org/doc/libs/1_36_0/libs/python/doc/index.html
10. en finance par exemple
11. http://ironpython.net/
12. http://www.boost.org/
13. Les *template* ou *patron de classe* sont un concept de métaprogrammation. Lorsque deux classes sont presque identiques à l'exception d'un type, le langage *C++* autorise la création d'un patron unique pour ces deux classes. Le *C++* est à typage statique : il faut écrire deux classes pour créer un tableau d'entiers et un tableau de réels même si le code est le même. Les *template*

Il est préférable d'avoir programmé en *C++* et en *Python* pour pouvoir mettre en œuvre les exemples proposés ci-dessous. La prise en main nécessite un investissement, une certaine habitude des longs messages d'erreurs issus des *template* mais une fois que cela est fait, la librairie accélère notablement l'écriture d'un module. Il n'est plus utile de se soucier de l'interfaçage avec *Python* et notamment du référencement déréférencement des objets pour le *garbage collector*[14].

**Remarque 6.4 : utilisation d'un logiciel de suivi de source**
Le nombre de fichiers est assez important, il est conseillé de conserver l'historique des modifications grâce à un logiciel de suivi de source tels que *TortoiseSVN*[15]. Cela évite de perdre parfois une demi-journée de travail.

### 6.5.1 Exemple

Vous pourrez trouver à l'adresse suivante [16] l'ensemble des sources nécessaires à la réalisation d'un module avec *Boost Python*[17]. Ce projet est configuré pour *Microsoft Visual Studio C++* et définit un premier module appelé *PythonSample* qui contient quelques fonctions et une classe. Le résultat est une DLL (Dynamic Link Library) qui sera appelée par le langage *Python*. Cet exemple est conçu pour *Windows*.

Un point important est l'utilisation de la librairie `c:\python26\python26.lib` qui explique qu'un module compilé pour une version donnée de *Python* ne puisse être utilisé avec une autre version : un module compilé avec la version 2.4 ne fonctionnera pas avec la version 2.6.

L'exemple proposé contient également différentes fonctions de conversion des structures *Python* vers des containers de la *STL* ou *Standard Template Library*. Il suffit d'ouvrir le fichier `pythonsample.sln` avec *Microsoft Visual C++* pour avoir accès au code source, c'est ce que montre la figure 6.2. Le fichier `exesample.sln` définit un programme exécutable permettant de tester le module sans *Python* (voir figure 6.3). Les paragraphes qui suivent présentent quelques éléments de syntaxe et se concluent par un exercice qui consiste à ajouter une fonction au module.

### 6.5.2 Les grandes lignes

Un module construit avec *Boost Python* se compose de trois éléments, une inamovible DLL `boost_python.dll`, un fichier `<module>.py` portant le nom du module, une DLL `<module>.dll`. Les deux premiers fichiers ne changent pas, le dernier contient les classes et fonctions proposées par le module. L'organisation proposée dans l'exemple du paragraphe 6.5.1 suit le schéma de la figure 6.3. L'utilisation d'un

permettent de factoriser le code de ces deux classes.

14. Le langage *Python* détruit automatique les objets qui ne sont plus utilisés. Cette fonctionnalité s'appuie sur un référencement de tous les objets créés. Alors qu'il est implicite en *Python*, il est explicite dans un module *Python* écrit en langage *C* à moins d'utiliser une librairie comme *Boost Python* qui prend en charge une partie de cette tâche (voir la page http://docs.python.org/c-api/intro.html#objects-types-and-reference-counts).

15. http://tortoisesvn.tigris.org/

16. http://www.xavierdupre.fr/

17. L'adresse suivante fournit une aide plus complète que ce document : http://www.boost.org/libs/python/doc/tutorial/doc/html/index.html.

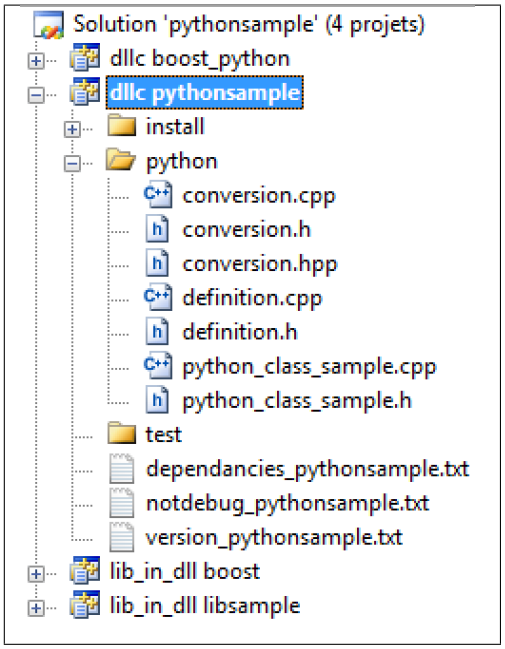



**Figure 6.2 :** *Copie d'écran du projet ouvert avec Microsoft Visual C++. Les deux projets* `dllc boost_python` *et* `lib_in_dll boost` *contiennent les sources Boost Python. Le projet* `lib_in_dll libsample` *contient le code d'une librairie qui est écrite sans tenir compte du fait que celle-ci sera utilisée sous forme de module Python. Le dernier projet* `dllc pythonsample` *ajoute la couche qui fera le pont entre la librairie* `libsample` *et le langage Python. Les fichiers* `conversion.*` *contiennent des fonctions de conversion des types Python vers des types C++ ou plus précisément de la STL (Standard Template Library). Le fichier* `definition.cpp` *contient le point d'entrée du module : la macro* `BOOST_PYTHON_MODULE` *qui déclare à Python les fonctions et classes du module.*

programme exécutable n'est pas indispensable. Toutefois, cela permet de tester le module sans *Python* et ainsi de faciliter sa conception.

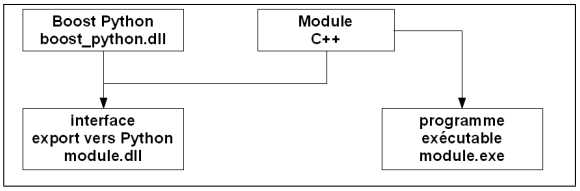



**Figure 6.3 :** *Schéma général de l'exemple introduit au paragraphe 6.5.1. Le code de la librairie est utilisé selon deux façons, à l'intérieur d'un module Python ce qui est l'objectif recherché et à l'intérieur d'un programme exécutable utilisé pour tester cette librairie sans passer par Python.*

Une fois le module prêt, il est utilisable dans tout programme *Python*. Il suffit de l'importer. Lors de cette étape, une fonction va être exécutée pour indiquer au langage *Python* l'ensemble des classes et des fonctions présentes dans le module.

#### 6.5.2.1 Initialiser le module

L'initialisation d'un module correspond à la déclaration au langage *Python* de l'ensemble des classes et des fonctions à exporter. Cette partie peut contenir des tâches qui doivent être faites la première fois que *Python* importe le module (initialisation de variables globales par exemple, chargement de données, ...). L'objectif de cette initialisation est de construire pour chaque fonction ou classe ou méthode un objet *Python* correspondant.

```
BOOST_PYTHON_MODULE(PythonSample)
{
```

```
  // étapes d'initialisation
} ;
```

#### 6.5.2.2 Ajout d'une fonction

La fonction `boost :: python :: def` permet de déclarer une fonction. On peut y préciser les arguments, des valeurs par défaut, un message d'aide retourné par l'instruction *Python* `help`.

```
int fonction (int r, const char * s, double x = 4) ;
BOOST_PYTHON_MODULE(PythonSample)
{
  boost::python::def ("fonction", fonction,
                        (boost::python::arg ("r"),
                         boost::python::arg ("s"),
                         boost::python::arg ("x") = 4),
                         "aide associée à la fonction") ;
} ;
```

Les arguments de `boost :: python :: def` sont les suivants :

1. Le premier argument est le nom de la fonction exportée en *Python*. Ce nom peut tout à fait être différent de celui de la fonction *C++*.
2. Le second argument est la fonction elle-même.
3. Le troisième argument est la liste des arguments suivis ou non d'une valeur par défaut.
4. Le dernier argument est l'aide associée à la fonction.

Côté *Python*, cela donne :

```
import PythonSample
print PythonSample.fonction (4, "second")
print PythonSample.fonction (4, "second", 5)
```

La surcharge de fonction est possible avec *Boost Python* mais elle ne sera pas détaillée ici. Le langage *Python* n'autorise pas la surcharge, la rendre possible revient à créer une fonction qui appellera telle ou telle autre fonction selon le type et le nombre des paramètres qu'elle aura reçus.

#### 6.5.2.3 Ajout d'une classe

La déclaration d'une classe suit le même schéma que pour une fonction mais avec le template `boost :: python :: class_`. Lorsque le constructeur n'a pas d'argument, la déclaration suit le schéma suivant :

```
   boost::python::class_<PythonClassSample> obj (
             "ClassSample",
             "help on PythonClassSample") ) ;
```

Le template `class_` est instancié sur la classe à exporter, ici `PythonClassSample`. On crée un objet `obj` de type `boost :: python :: class_ < PythonClassSample >`. Son constructeur prend comme arguments :

1. Le nom de la classe en *Python*.
2. L'aide qui lui est associée. Ce second paramètre est facultatif mais il est conseillé de le renseigner.

Au cas où la classe aurait un constructeur avec des paramètres, il faudrait ajouter le code suivant :

```
boost::python::class_<PythonClassSample> obj (
        "ClassSample",
        "help on PythonClassSample",
        boost::python::init<int,const char*> (                    // ajout
             (boost::python::arg ("a"),                           // ajout
              boost::python::arg ("s") = "default value for s"),  // ajout
              "help on PythonClassSample constructor" )           // ajout
        ) ;
```

C'est-à-dire un autre template `boost :: python :: init < ... >` dont les arguments sont les types des paramètres du constructeur, c'est-à-dire entier et chaîne de caractères dans l'exemple précédent. La suite est identique à la déclaration d'une fonction. Il est également possible de définir un héritage de la manière qui suit :

```
boost::python::class_<PythonClassSample,
                      boost::python::bases<ClassBase> > obj (
                           "ClassSample",
                           "help on PythonClassSample") ;
```

Le template `class_` peut recevoir un autre argument qui est `boost :: python :: bases < ClassBase >` avec `ClassBase` désignant la classe mère de la classe déclarée. Cette dernière doit avoir été déclarée au préalable avec cette même syntaxe `boost :: python :: class_`.

```
boost::python::class_<PythonClassSample,
           boost::python::bases<ClassBase> > obj (
         // ...
```

**Remarque 6.5 : template *C++***
Lorsqu'un template est un argument d'un autre template, il faut insérer un espace entre deux symboles < ou > consécutifs pour éviter la confusion avec les opérateurs `<<` ou `>>` qui existent en *C++*.

#### 6.5.2.4 Ajout d'une méthode

La déclaration d'une classe passe par la déclaration d'une instance du template `class_`. Dans les exemples utilisés, cette instance porte le nom `obj`. C'est cette instance qui va permettre de déclarer une méthode exactement de la même manière qu'une fonction :

```
obj.def (   "Random",
            &PythonClassSample::Random,
            (boost::python::arg ("pos")),
            "help on the method") ;
```

Cette même méthode `def` permet également de surcharge le constructeur au cas où la classe pourrait être initialisée de différentes manières. Cela donne :

```
boost::python::class_<MyVector> obj ("Vector", "help on vector",
        boost::python::init<int> (
        (PY_ARG ("n")),
        "crée un vecteur de dimension n")) ;
// ajout d'un constructeur sans paramètre
obj.def (boost::python::init<MyVector> ()) ;
```

La classe *Python* `Vector` peut être créée avec un ou aucun paramètre :

```
v  = PythonSample.Vector ()
v2 = PythonSample.Vector (10)
```

#### 6.5.2.5 Ajout d'un attribut

La déclaration d'un attribut s'effectue avec la méthode `def_readwrite` :

```
obj.def_readwrite ("a", &PythonClassSample::_a, "retourne un accès à a") ;
```

Le premier paramètre sera son nom sous *Python*, le second le paramètre lui-même, le troisième l'aide associée. La méthode `def_readwrite` permet d'exporter un attribut de sorte qu'il soit lisible et modifiable.

### 6.5.3 Exemple concret : ajout d'une fonction

L'objectif est d'ajouter une fonction au module. La première étape consiste à décompresser le fichier téléchargé [18], à compiler l'ensemble des fichiers sous *Microsoft Visual C++* en version *release* et *debug*. L'exécution du programme `PythonSample/test/test.py` doit s'effectuer sans problème. Ajouter une fonction revient ensuite à suivre les étapes suivantes :

1. Déclarer la fonction dans le fichier `libSample/mafonction.h` qu'on crée.
2. Implémenter la fonction dans le fichier `libSample/mafonction.cpp` qu'on crée également.

La partie purement *C++* est terminée [19], il faut maintenant définir la transition *C++*→*Python* grâce aux étapes suivantes :

1. On déclare une fonction dans un fichier `PythonSample/mafonction.h` qu'on crée. Elle peut par exemple porter le même nom que la précédente mais préfixé par `python_`.
2. On peut passer à l'implémentation de la fonction *Python* dans un fichier `PythonSample/mafonction.cpp` qu'on crée aussi. Le code est sensiblement le même pour chaque fonction ajoutée : il faut convertir les paramètres depuis des structures *Python* en type *C++*, appeler la fonction *C++* puis effectuer la conversion inverse. On pourra s'aider des fonctions contenues dans le fichier `PythonSample/python/conversion.h`.

18. depuis l'adresse http://www.xavierdupre.fr/

19. C'est facultatif mais il est également conseillé d'écrire d'écrire un code pour tester cette fonction dans le fichier `exeSample/source/main.cpp`. Cela réduit le risque d'erreurs d'exécution dans cette partie.

3. Il suffit de déclarer la fonction au sein de la macro `BOOST_PYTHON_MODULE` dans le fichier `PythonSample/python/definition.cpp`.

Il ne reste plus qu'à tester la fonction nouvellement incluse depuis le programme `PythonSample/test/test.py`. Le contenu de chaque fichier modifié lors de ces étapes est présenté par la figure 6.4 page 168.

### 6.5.4 Garbage collector et pointeur

Le langage *Python* gère lui-même la destruction des objets au travers d'un *garbage collector*. Cela signifie qu'un objet créé en *Python* provenant d'un module défini dans module en *C++* sera lui aussi détruit par *Python*. A partir du moment où le destructeur d'un objet libère les pointeurs qu'il a alloué, la mémoire ne sera pas corrompue. Ce schéma ne permet pas de prévoir quand l'objet sera détruit.

Pour parer aux problèmes éventuels qui pourraient survenir notamment lorsqu'un objet contient des pointeurs sur des données détenues par un autre objet, il est possible de spécifier à *Boost Python* qu'un résultat est une référence : son identificateur *Python* peut être détruit mais pas l'objet qu'il désigne. C'est l'objet du template `boost :: python :: return_internal_reference <> ()`. Il est utilisé pour spécifier qu'une fonction retourne un résultat qui ne doit pas être détruit.

Par exemple, on utilise cette fonctionnalité pour définir un opérateur d'affectation comme l'opérateur `+=`. La fonction suivante a la même signature *C++* que l'opérateur `+=`.

```
class PythonClassSample
{
    ...
    PythonClassSample & __iadd__ (const PythonClassSample &a) ;
    ...
} ;
```

Son résultat est une référence sur un objet existant. Si aucune mention spécifique n'est précisée lors de l'export de la méthode (avec `def`), lors d'un appel à l'opérateur `+=`, *Python* va considérer que le résultat est un objet différent alors qu'il s'agit de deux identificateurs *Python* faisant référence au même objet *C++*. C'est pour cela qu'on ajoute l'argument suivant à la méthode `def` :

```
    x.def ("__iadd__",
           &PythonClassSample::__iadd__,
           boost::python::return_internal_reference<>(),
           "addition") ;
```

De cette façon, il y aura bien deux identificateurs *Python* faisant référence au même objet *C++*. C'est aussi cette syntaxe qui permet de déclarer une fonction retournant un pointeur sur un objet *C++*.

### 6.5.5 Utilisation, installation

Le module terminé se compose de trois fichiers :

1. la DLL compilée avec le code C++ du module,
2. un fichier portant le même nom que la DLL mais d'extension .py : `PythonSample.py`,
3. un fichier `__init__.py`.

Voici ce fichier pour l'exemple présenté au paragraphe 6.5.1. La fonction importante de ce fichier est `load_dynamic` qui charge la DLL en mémoire au cas où elle n'aurait jamais été chargée.

```
import sys
if "PythonSample" not in sys.modules :
    PythonSample = imp.load_dynamic ('PythonSample', PythonSample.dll)
    sys.modules ["PythonSample"]  = PythonSample
```

### 6.5.6 Détails sur l'exemple du paragraphe 6.5.1

Certains fichiers exposés ici ont été simplifiés par rapport à l'exemple téléchargeable. Celui-ci a été configuré de manière à produire deux DLL, une autre pour la version *release* `PythonSample.dll` et une pour la version *debug* `PythonSampled.dll`. Ceci permet d'utiliser l'une ou l'autre des versions dans le même programme. La version *debug* facilite la correction des erreurs qui pourraient survenir lors de l'utilisation du module sous *Python*(voir le paragraphe 6.5.7). Il suffit alors d'écrire :

```
import PythonSampled as PythonSample
# au lieu de import PythonSample
```

De cette façon, passer d'une DLL à l'autre est simple et permet de débugger, tâche qu'introduit le paragraphe suivant. Si le programme *Python* s'étale sur plusieurs fichiers, il est plus pratique d'ajouter au début de chacun fichier qui utilise le module *C++* :

```
import sys
if "PythonSampled" in sys.modules : PythonSample = sys.modules ["PythonSampled"]
else : import PythonSample
```

### 6.5.7 Debuggage

Il ne serait pas pensable de développer des outils sans pouvoir utiliser un débuggeur. Dans l'exemple présenté au paragraphe 6.5.1, le fichier `testd.py` n'utilise pas le module compilé dans sa version *release* mais dans sa version ***debug***. L'exécution est plus lente mais permet de faire appel au débuggeur pour corriger une éventuelle erreur d'exécution.

Une manière de débugger consiste à momentanément arrêter l'exécution du programme *Python* (à l'aide de la fonction `raw_input` par exemple), le temps d'attacher le débuggeur de *Microsoft Visual C++* au processus dont le nom est *python.exe* ou *pythonw.exe*. Il ne reste plus qu'à placer des pointeurs d'arrêt puis à continuer l'exécution du programme *Python* jusqu'à ce que ces pointeurs d'arrêt soient atteints.

Une seconde méthode consiste à volontairement insérer dans le code *C++* du module l'instruction `__debugbreak`(). Lorsque le module est exécuté via un programme

*Python* et qu'il arrive sur cette ligne, il vous demande l'autorisation de continuer l'exécution via un débuggeur. Il faudra penser à enlever la ligne une fois le module corrigé.

Il reste toutefois une dernière façon plus classique de débugger qui est l'ajout dans le code d'instructions écrivant des messages dans un fichier texte. On écrit des *logs* ou des *traces*. De cette façon, lorsque le programme provoque une erreur, on connaît la séquence de messages générés par le programme. S'ils sont suffisamment explicites, cela permet de corriger l'erreur. Sinon, l'ajout de messages supplémentaires puis une nouvelle exécution du programme permettront d'être plus précis quant à la cause de l'erreur.

Outre ces trois méthodes de débuggage, il est possible qu'un module doivent retourner un code d'erreur ou générer une exception qui soit attrapée par l'interpréteur *Python* puis afficher. C'est l'objet des lignes qui suivent :

```
if (<condition>)
    throw std::runtime_error ("message d'erreur") ;
```

Toute exception dérivant de `runtime_error` sera interceptée par l'interpréteur *Python*. La librairie *Boost Python* génère également des exceptions notamment lorsque les paramètres envoyés à une fonction du module ne correspondent pas avec sa signature.

En terme d'architecture, il est conseillé de bien scinder la partie utilisant *Boost Python* des algorithmes. *Boost Python* n'est qu'une passerelle vers *Python*. Il pourrait être nécessaire de construire une autre passerelle vers d'autres langages tel que le *Visual Basic pour Applications (VBA)*. De même, outre la passerelle vers *Python*, il est utile de construire un programme exécutable permettant de tester les algorithmes écrits en *C++* avant de les exporter. Cela réduit beaucoup le temps de débuggage qui peut alors gérer séparément la partie *C++* de la partie interfaçage entre *Python* et *C++*.

### 6.5.8 Pour aller plus loin

La librairie *Boost Python*[20] permet de faire beaucoup plus de choses que ce qui est présenté dans ces paragraphes. Ceux-ci contiennent néanmoins l'essentiel pour exporter vers *Python* les objets les plus simples. Il est également possible de :

- surcharger une méthode *C++* dans un programme *Python*,
- exécuter un programme écrit en *Python* depuis un programme *C++*.

Il faut parfois une certaine dose de volonté et de ténacité pour parvenir à maîtriser un outil tel que *Boost Python*, surtout lorsqu'une erreur se déclare dans le code de librairie après une modification en dehors de cette même librairie.

20. http://www.boost.org/libs/python/doc/

| | |
|---|---|
| `calendar` | Gérer les calendriers, les dates (voir chapitre 7). |
| `cgi` | Utilisé dans les scripts CGI (programmation Internet) |
| `cmath` | Fonctions mathématiques complexes. |
| `codecs` | Jeux de caractères (voir paragraphe 7.8) |
| `copy` | Copies d'instances de classes. |
| `csv` | Gestion des fichiers au format CSV (utilisés par *Microsoft Excel*). |
| `datetime` | Calculs sur les dates et heures (voir chapitre 7). |
| `gc` | Gestion du garbage collector. |
| `getopt` | Lire les options des paramètres passés en arguments d'un programme *Python*. |
| `glob` | Chercher des fichiers (voir chapitre 7). |
| `hashlib` | Fonctions de cryptage. |
| `htmllib` | Lire le format HTML. |
| `math` | Fonctions mathématiques standard telles que `cos`, `sin`, `exp`, `log`... |
| `os` | Fonctions systèmes dont certaines fonctions permettant de gérer les fichiers (voir chapitre 7). |
| `os.path` | Manipulations de noms de fichiers (voir chapitre 7). |
| `pickle` | Sérialisation d'objets, la sérialisation consiste à convertir des données structurées de façon complexe en une structure linéaire facilement enregistrable dans un fichier (voir chapitre 7). |
| `profile` | Etudier le temps passé dans les fonctions d'un programme. |
| `random` | Génération de nombres aléatoires. |
| `re` | Expressions régulières (voir paragraphe 7.6). |
| `shutil` | Copie de fichiers (voir chapitre 7). |
| `sqlite3` | Accès aux fonctionnalités du gestionnaire de base de données SQLite3. |
| `string` | Manipulations des chaînes de caractères. |
| `sys` | Fonctions systèmes, fonctions liées au langage *Python* (voir chapitre 7). |
| `threading` | Utilisation de threads (voir chapitre 9). |
| `time` | Accès à l'heure, l'heure système, l'heure d'une fichier. |
| `Tkinter` | Interface graphique (voir chapitre 8). |
| `unittest` | Tests unitaires (ou comment améliorer la fiabilité d'un programme). |
| `urllib` | Pour lire le contenu de page HTML sans utiliser un navigateur. |
| `urllib2` | Module plus complet que `urllib`. |
| `xml.dom` | Lecture du format XML. |
| `xml.sax` | Lecture du format XML. |
| `zipfile` | Lecture de fichiers ZIP (voir chapitre 7). |

**Table 6.1 :** *Liste de modules souvent utilisés. La liste exhaustive est disponible à l'adresse http://docs.python.org/modindex.html. Les moteurs de recherche sur Internet retournent des résultats assez pertinents sur des requêtes du type python + le nom du module. Les résultats sont principalement en langue anglaise.*

Fichier libSample/mafonction.h

```
#ifndef LIB_MAFONCTION_H
#define LIB_MAFONCTION_H

#include <vector>

void somme_vecteur (
  const std::vector<double> &v1,
  const std::vector<double> &v2,
  std::vector<double> &res) ;

#endif
```

Fichier libSample/mafonction.cpp

```
#include "mafonction.h"
void somme_vecteur (
  const std::vector<double> &v1,
  const std::vector<double> &v2,
  std::vector<double> &res)
{
  if (v1.size () != v2.size ())
    throw std::runtime_error (
          "dimensions différentes") ;
  res.resize (v1.size ()) ;
  std::vector<double>::
          const_iterator it1,it2 ;
  std::vector<double>::iterator it3 ;
  for (it1 = v1.begin (),
       it2 = v2.begin (),
       it3 = res.begin () ;
       it1 != v1.end () ;
       ++it1, ++it2, ++it3)
       *it3 = *it1 + *it2 ;
}
```

Fichier PythonSample/mafonction.h

```
#ifndef PYTHON_MAFONCTION_H
#define PYTHON_MAFONCTION_H

#include "python/definition.h"

boost::python::list
     python_somme_vecteur (
        boost::python::list v1,
        boost::python::list v2) ;

#endif
```

Fichier PythonSample/mafonction.cpp

```
#include "python/definition.h"
#include "mafonction.h"
#include "../libsample/mafonction.h"
#include "python/conversion.h"
#include "python/conversion.hpp"
boost::python::list
     python_somme_vecteur (
        boost::python::list v1,
        boost::python::list v2)
{
  std::vector<double> cv1,cv2,cres ;
  PythonConvert (v1, cv1) ;
  PythonConvert (v2, cv2) ;
  somme_vecteur (cv1, cv2, cres) ;
  boost::python::list res ;
  PythonConvert (cres, res) ;
  return res ;
}
```

Fichier PythonSample/python/main.cpp

```
// ...
#include "../mafonction.h"
BOOST_PYTHON_MODULE(PythonSample)
{
  // ...
  def ("somme_vecteur",
        &python_somme_vecteur,
        (boost::python::arg ("v1"),
         boost::python::arg ("v2")),
 "retourne la somme de deux vecteurs");
  // ...
} ;
```

Fichier PythonSample/test/test.py

```
import PythonSample as PS
v1 = [1.0, 2.0]
v2 = [6.5, 7.8]
v3 = PS.somme_vecteur (v1, v2)
```

**Figure 6.4 :** *Six fichiers modifiés pour ajouter une fonction dans un module Python écrit en C++ à partir de l'exemple accessible depuis l'adresse http://www.xavierdupre.fr/.*

# Chapitre 7

# Fichiers, expressions régulières, dates

---

Lorsqu'un programme termine son exécution, toutes les informations stockées dans des variables sont perdues. Un moyen de les conserver est de les enregistrer dans un fichier sur disque dur. A l'intérieur de celui-ci, ces informations peuvent apparaître sous un format texte qui est lisible par n'importe quel éditeur de texte, dans un format compressé, ou sous un autre format connu par le concepteur du programme. On appelle ce dernier type un format binaire, il est adéquat lorsque les données à conserver sont très nombreuses ou lorsqu'on désire que celles-ci ne puissent pas être lues par une autre application que le programme lui-même. En d'autres termes, le format binaire est illisible excepté pour celui qui l'a conçu.

Ce chapitre abordera pour commencer les formats texte, binaire et compressé (*zip*) directement manipulable depuis *Python*. Les manipulations de fichiers suivront pour terminer sur les expressions régulières qui sont très utilisées pour effectuer des recherches textuelles. A l'issue de ce chapitre, on peut envisager la recherche à l'intérieur de tous les documents textes présents sur l'ordinateur, de dates particulières, de tous les numéros de téléphones commençant par 06... En utilisant des modules tels que `reportlab` [1] ou encore `win32com` [2], il serait possible d'étendre cette fonctionnalité aux fichiers de type *pdf* et aux fichiers *Microsoft Word, Excel*.

## 7.1 Format texte

Les fichiers texte sont les plus simples : ce sont des suites de caractères. Le format *HTML* et *XML* font partie de cette catégorie. Ils servent autant à conserver des informations qu'à en échanger comme par exemple transmettre une matrice à *Microsoft Excel*.

Ce format, même s'il est simple, implique une certaine organisation dans la façon de conserver les données afin de pouvoir les récupérer. Le cas le plus fréquent est l'enregistrement d'une matrice : on choisira d'écrire les nombres les uns à la suite des autres en choisissant un séparateur de colonnes et un séparateur de lignes. Ce point sera abordé à la fin de cette section.

### 7.1.1 Ecriture

La première étape est l'écriture. Les informations sont toujours écrites sous forme de chaînes de caractères et toujours ajoutées à la fin du fichier qui s'allonge jusqu'à

1. http://www.reportlab.com/software/opensource/rl-toolkit/
2. http://python.net/crew/mhammond/win32/Downloads.html

ce que toutes les informations y soient écrites. L'écriture s'effectue toujours selon le même schéma.

1. création ou ouverture du fichier,
2. écriture,
3. fermeture.

Lors de l'ouverture, le fichier dans lequel seront écrites les informations est créé s'il n'existe pas ou nettoyé s'il existe déjà. La fermeture permet à d'autres programmes de lire ce que vous avez placé dans ce fichier. Sans cette dernière étape, il sera impossible d'y accéder à nouveau pour le lire ou y écrire à nouveau. A l'intérieur d'un programme informatique, écrire dans un fichier suit toujours le même schéma :

```
f = open ("nom-fichier", "w")    # ouverture

f.write ( s )    # écriture de la chaîne de caractères  s
f.write ( s2 )   # écriture de la chaîne de caractères  s2
...

f.close ()  # fermeture
```

Les étapes d'ouverture et de fermeture sont toujours présentes en ce qui concerne les fichiers. Il s'agit d'indiquer au système d'exploitation que le programme souhaite accéder à un fichier et interdire à tout autre programme l'accès à ce fichier. Un autre programme qui souhaiterait créer un fichier du même nom ne le pourrait pas tant que l'étape de fermeture n'est pas exécutée. En revanche, il pourrait tout à fait le lire car la lecture ne perturbe pas l'écriture.

**Remarque 7.1 : fichiers ouverts et non fermés**
Lorsque que le programme se termine, même s'il reste des fichiers "ouverts" pour lesquels la méthode `close` n'a pas été appelée, ils seront automatiquement fermés.

Certains caractères sont fort utiles lors de l'écriture de fichiers texte afin d'organiser les données. Le symbole ; est très utilisé comme séparateur de colonnes pour une matrice, on utilise également le passage à la ligne ou la tabulation. Comme ce ne sont pas des caractères "visibles", ils ont des codes :

| | |
|---|---|
| `\n` | passage à la ligne |
| `\t` | insertion d'une tabulation, indique un passage à la colonne suivante dans le logiciel *Microsoft Excel* |

Il existe peu de manières différentes de conserver une matrice dans un fichier, le programme ressemble dans presque tous les cas à celui qui suit :

```
mat =  ... # matrice de type liste de listes
f = open ("mat.txt", "w")
for i in range (0,len (mat)) :                # la fonction join est aussi
    for j in range (0, len (mat [i])) :       # fréquemment utilisée
        f.write ( str (mat [i][j]) + "\t")    # pour assembler les chaînes
    f.write ("\n")                            # un une seule et réduire le
f.close ()                                    # nombre d'appels à f.write
```

**Remarque 7.2 : fonction `open`**
La fonction `open` accepte deux paramètres, le premier est le nom du fichier, le second définit le mode d'ouverture : "`w`" pour écrire (**w**rite), "a" pour écrire et ajouter (**a**ppend), "r" pour lire (**r**ead). Ceci signifie que la fonction `open` sert à ouvrir un fichier quelque soit l'utilisation qu'on en fait.

**Remarque 7.3 : autre syntaxe avec `print`**
Il existe une autre syntaxe qui permet de remplacer la méthode `write`. Elle utilise la même syntaxe que celle de l'instruction `print` et fonctionne de la même manière à ceci près qu'elle ajoute `>> f,` entre le mot-clé `print` et le texte à écrire dans le fichier `f`. Néanmoins, ce n'est pas cette syntaxe qui sera mise en avant dans les versions futures de *Python*.

```
mat = ... # matrice de type liste de listes
f = open ("mat.txt", "w")
for i in range (0,len (mat)) :
    for j in range (0, len (mat [i])) :
        print >> f, str (mat [i][j]), "\t",  # ligne changée
    print >> f                               # ligne changée
f.close ()
```

**Remarque 7.4 : écriture différée**
A la première écriture dans un fichier (premier appel à la fonction `write`), la taille du fichier créée est souvent nulle. L'écriture dans un fichier n'est pas immédiate, le langage *Python* attend d'avoir reçu beaucoup d'informations avant de les écrire physiquement sur le disque dur. Les informations sont placées dans un tampon ou *buffer*. Lorsque le tampon est plein, il est écrit sur disque dur. Pour éviter ce délai, il faut soit fermer puis réouvrir le fichier soit appeler la méthode `flush` qui ne prend aucun paramètre. Ce mécanisme vise à réduire le nombre d'accès au disque dur, il n'est pas beaucoup plus long d'y écrire un caractère plutôt que 1000 en une fois.

### 7.1.2 Ecriture en mode "ajout"

Lorsqu'on écrit des informations dans un fichier, deux cas se présentent. Le premier consiste à ne pas tenir compte du précédent contenu de ce fichier lors de son ouverture pour écriture et à l'écraser. C'est le cas traité par le précédent paragraphe. Le second cas consiste à ajouter toute nouvelle information à celles déjà présentes lors de l'ouverture du fichier. Ce second cas est presque identique au suivant hormis la première ligne qui change :

```
f = open ("nom-fichier", "a")    # ouverture en mode ajout, mode "a"
...
```

Pour comprendre la différence entre ces deux modes d'ouverture, voici deux programmes. Celui de gauche n'utilise pas le mode ajout tandis que celui de droite l'utilise lors de la seconde ouverture.

```
f = open ("essai.txt", "w")
f.write (" premiere fois ")
f.close ()
f = open ("essai.txt", "w")
f.write (" seconde fois ")
f.close ()

# essai.txt : seconde fois
```

```
f = open ("essai.txt", "w")
f.write (" premiere fois ")
f.close ()
f = open ("essai.txt", "a")  ###
f.write (" seconde fois ")
f.close ()

# essai.txt : premiere fois  seconde fois
```

Le premier programme crée un fichier "`essai.txt`" qui ne contient que les informations écrites lors de la seconde phase d'écriture, soit `seconde fois`. Le second utilise le mode ajout lors de la seconde ouverture. Le fichier "`essai.txt`", même s'il existait avant l'exécution de ce programme, est effacé puis rempli avec l'information `premiere fois`. Lors de la seconde ouverture, en mode ajout, une seconde chaîne de caractères est ajoutée. le fichier "`essai.txt`", après l'exécution du programme contient donc le message : `premiere fois seconde fois`.

**Remarque 7.5 : fichier de traces**
Un des moyens pour comprendre ou suivre l'évolution d'un programme est d'écrire des informations dans un fichier ouvert en mode ajout qui est ouvert et fermé sans cesse. Ce sont des fichiers de *traces* ou de *log*. Ils sont souvent utilisés pour vérifier des calculs complexes. Ils permettent par exemple de comparer deux versions différentes d'un programme pour trouver à quel endroit ils commencent à diverger.

### 7.1.3 Lecture

La lecture d'un fichier permet de retrouver les informations stockées grâce à une étape préalable d'écriture. Elle se déroule selon le même principe, à savoir :

1. ouverture du fichier en mode lecture,
2. lecture,
3. fermeture.

Une différence apparaît cependant lors de la lecture d'un fichier : celle-ci s'effectue ligne par ligne alors que l'écriture ne suit pas forcément un découpage en ligne. Les instructions à écrire pour lire un fichier diffèrent rarement du schéma qui suit où seule la ligne indiquée par $(*)$ change en fonction ce qu'il faut faire avec les informations lues.

```
f = open ("essai.txt", "r")  # ouverture du fichier en mode lecture
for ligne in f :             # pour toutes les lignes du fichier
    print ligne              # on affiche la ligne (*)
f.close ()                   # on ferme le fichier
```

Pour des fichiers qui ne sont pas trop gros ($< 100000$ lignes), il est possible d'utiliser la méthode `readlines` qui récupère toutes les lignes d'un fichier texte en une seule fois. Le programme suivant donne le même résultat que le précédent.

```
f = open ("essai.txt", "r") # ouverture du fichier en mode lecture
l = f.readlines ()          # lecture de toutes les lignes, placées dans une liste
f.close ()                  # fermeture du fichier

for s in l : print s        # on affiche les lignes à l'écran (*)
```

**Remarque 7.6 : code de fin de ligne**
Lorsque le programme précédent lit une ligne dans un fichier, le résultat lu inclut le ou les caractères (\n \r) qui marquent la fin d'une ligne. C'est pour cela que la lecture est parfois suivie d'une étape de nettoyage.

```
f = open ("essai.txt", "r") # ouverture du fichier en mode lecture
l = f.readlines ()          # lecture de toutes les lignes, placées dans une liste
f.close ()                  # fermeture du fichier

     # contiendra la liste des lignes nettoyées
l_net = [ s.strip ("\n\r") for s in l ]
```

Les informations peuvent être structurées de façon plus élaborée dans un fichier texte, c'est le cas des formats *HTML* ou *XML*. Pour ce type de format plus complexe, il est déconseillé de concevoir soi-même un programme capable de les lire, il existe presque toujours un module qui permette de le faire. C'est le cas du module `HTMLParser` pour le format *HTML* ou `xml.sax` pour le format *XML*. De plus, les modules sont régulièrement mis à jour et suivent l'évolution des formats qu'ils décryptent.

Un fichier texte est le moyen le plus simple d'échanger des matrices avec un tableur et il n'est pas besoin de modules dans ce cas. Lorsqu'on enregistre une feuille de calcul sous format texte, le fichier obtenu est organisé en colonnes : sur une même ligne, les informations sont disposées en colonne délimitées par un séparateur qui est souvent une tabulation (\t) ou un point virgule comme dans l'exemple suivant :

```
nom  ; prénom ; livre
Hugo  ; Victor  ; Les misérables
Kessel ; Joseph  ; Le lion
Woolf ; Virginia  ; Mrs Dalloway
Calvino ; Italo  ; Le baron perché
```

Pour lire ce fichier, il est nécessaire de scinder chaque ligne en une liste de chaînes de caractères, on utilise pour cela la méthode `split` des chaînes de caractères.

```
mat = []                      # création d'une liste vide,
f = open ("essai.txt", "r")   # ouverture du fichier en mode lecture
for li in f :                 # pour toutes les lignes du fichier
    s = li.strip ("\n\r")     # on enlève les caractères de fin de ligne
    l = s.split (";")         # on découpe en colonnes
    mat.append (l)            # on ajoute la ligne à la matrice
f.close ()                    # fermeture du fichier
```

Ce format de fichier texte est appelé *CSV*[3], il peut être relu depuis un programme *Python* comme le montre l'exemple précédent ou être chargé depuis *Microsoft Excel* en précisant que le format du fichier est le format *CSV*. Pour les valeurs numériques, il ne faut pas oublier de convertir en caractères lors de l'écriture et de convertir en nombres lors de la lecture.

**Remarque 7.7 : nombres français et anglais**
Les nombres réels s'écrivent en anglais avec un point pour séparer la partie entière

3. Comma Separated Value

de la partie décimale. En français, il s'agit d'une virgule. Il est possible que, lors de la conversion d'une matrice, il faille remplacer les points par des virgules et réciproquement pour éviter les problèmes de conversion.

## 7.2 Fichiers zip

Les fichiers *zip* sont très répandus de nos jours et constituent un standard de compression facile d'accès quelque soit l'ordinateur et son système d'exploitation. Le langage *Python* propose quelques fonctions pour compresser et décompresser ces fichiers par l'intermédiaire du module `zipfile`. Le format de compression *zip* est un des plus répandus bien qu'il ne soit pas le plus performant [4]. Ce format n'est pas seulement utilisé pour compresser mais aussi comme un moyen de regrouper plusieurs fichiers en un seul.

### 7.2.1 Lecture

L'exemple suivant permet par exemple d'obtenir la liste des fichiers inclus dans un fichier *zip* :

```
import zipfile
file = zipfile.ZipFile ("exemplezip.zip", "r")
for info in file.infolist () :
    print info.filename, info.date_time, info.file_size
file.close ()
```

Les fichiers compressés ne sont pas forcément des fichiers textes mais de tout format. Le programme suivant extrait un fichier parmi ceux qui ont été compressés puis affiche son contenu (on suppose que le fichier lu est au format texte donc lisible).

```
import zipfile
file = zipfile.ZipFile ("exemplezip.zip", "r")
data = file.read ("informatique/testzip.py")
file.close ()
print data
```

On retrouve dans ce cas les étapes d'ouverture et de fermeture même si la première est implicitement inclus dans le constructeur de la classe `ZipFile`.

### 7.2.2 Ecriture

Pour créer un fichier *zip*, le procédé ressemble à la création de n'importe quel fichier. La seule différence provient du fait qu'il est possible de stocker le fichier à compresser sous un autre nom à l'intérieur du fichier *zip*, ce qui explique les deux premiers arguments de la méthode `write`. Le troisième paramètre indique si le fichier doit être compressé (`zipfile.ZIP_DEFLATED`) ou non (`zipfile.ZIP_STORED`).

4. D'autres formats proposent de meilleurs taux de compressions sur les fichiers textes existent, voir http://www.7-zip.org/.

```
import zipfile
file = zipfile.ZipFile ("test.zip", "w")
file.write ("fichier.txt", "nom_fichier_dans_zip.txt", zipfile.ZIP_DEFLATED)
file.close ()
```

Une utilisation possible de ce procédé serait l'envoi automatique d'un mail contenant un fichier `zip` en pièce jointe. Une requête comme *python* précédant le nom de votre serveur de mail permettra, via un moteur de recherche, de trouver des exemples sur Internet.

## 7.3 Manipulation de fichiers

Il arrive fréquemment de copier, recopier, déplacer, effacer des fichiers. Lorsqu'il s'agit de quelques fichiers, le faire manuellement ne pose pas de problème. Lorsqu'il s'agit de traiter plusieurs centaines de fichiers, il est préférable d'écrire un programme qui s'occupe de le faire automatiquement. Cela peut être la création automatique d'un fichier `zip` incluant tous les fichiers modifiés durant la journée ou la réorganisation de fichiers musicaux au format `mp3` à l'aide de modules complémentaires tel que `mutagen` [5] ou `pymedia` [6].

**Remarque 7.8 : majuscules et minuscules**
Pour ceux qui ne sont pas familiers des systèmes d'exploitation, il faut noter que *Windows* ne fait pas de différences entre les majuscules et les minuscules à l'intérieur d'un nom de fichier. Les systèmes *Linux* et *Mac OS X* font cette différence [7].

### 7.3.1 Gestion des noms de chemins

Le module `os.path` propose plusieurs fonctions très utiles qui permettent entre autres de tester l'existence d'un fichier, d'un répertoire, de récupérer diverses informations comme sa date de création, sa taille... Cette liste présentée par la table 7.1 est loin d'être exhaustive mais elle donne une idée de ce qu'il est possible de faire.

### 7.3.2 Copie, suppression

Le module `shutil` permet de copier des fichiers, le module `os` permet de les supprimer ou de les renommer, il permet également de créer ou de supprimer des répertoires. Ces opérations sont regroupées dans la table 7.2.

### 7.3.3 Liste de fichiers

La fonction `listdir` du module `os` permet de retourner les listes des éléments inclus dans un répertoire (fichiers et sous-répertoires). Toutefois, le module `glob` propose

5. http://code.google.com/p/mutagen/
6. http://pymedia.org/
7. Ceci explique que certains programmes aient des comportements différents selon le système d'exploitation sur lequel ils sont exécutés ou encore que certains liens Internet vers des fichiers ne débouchent sur rien car ils ont été saisis avec des différences au niveau des minuscules majuscules.

| | |
|---|---|
| `abspath(path)` | Retourne le chemin absolu d'un fichier ou d'un répertoire. |
| `commonprefix(list)` | Retourne le plus grand préfixe commun à un ensemble de chemins. |
| `dirname(path)` | Retourne le nom du répertoire. |
| `exists(path)` | Dit si un chemin est valide ou non. |
| `getatime(path)`<br>`getmtime(path)`<br>`getctime(path)` | Retourne diverses dates concernant un chemin, date du dernier accès (`getatime`), date de la dernière modification (`getmtime`), date de création (`getctime`). |
| `getsize(file)` | Retourne la taille d'un fichier. |
| `isabs(path)` | Retourne `True` si le chemin est un chemin absolu. |
| `isfile(path)` | Retourne `True` si le chemin fait référence à un fichier. |
| `isdir(path)` | Retourne `True` si le chemin fait référence à un répertoire. |
| `join(p1,p2,...)` | Construit un nom de chemin étant donné une liste de répertoires. |
| `split(path)` | Découpe un chemin, isole le nom du fichier ou le dernier répertoire des autres répertoires. |
| `splitext(path)` | Découpe un chemin en nom + extension. |

**Table 7.1 :** *Liste non exhaustive des fonctionnalités offertes par le module* `os.path`. *La page http://docs.python.org/library/os.path.html référence toutes les fonctions de ce module.*

| | |
|---|---|
| `copy(f1,f2)` | (module `shutil`) Copie le fichier `f1` vers `f2` |
| `chdir(p)` | (module `os`) Change le répertoire courant, cette fonction peut être importante lorsqu'on utilise la fonction `system` du module `os` pour lancer une instruction en ligne de commande ou lorsqu'on écrit un fichier sans préciser le nom du répertoire, le fichier sera écrit dans ce répertoire courant qui est par défaut le répertoire où est situé le programme *Python*. C'est à partir du répertoire courant que sont définis les chemins relatifs. |
| `getcwd()` | (module `os`) Retourne le répertoire courant, voir la fonction `chdir`. |
| `mkdir(p)` | (module `os`) Crée le répertoire `p`. |
| `makedirs(p)` | (module `os`) Crée le répertoire `p` et tous les répertoires des niveaux supérieurs s'ils n'existent pas. Dans le cas du répertoire `d : /base/repfinal`, crée d'abord `d : /base` s'il n'existe pas, puis `d : /base/repfinal`. |
| `remove(f)` | (module `os`) Supprime un fichier. |
| `rename(f1,f2)` | (module `os`) Renomme un fichier |
| `rmdir(p)` | (module `os`) Supprime un répertoire |

**Table 7.2 :** *Liste non exhaustive des fonctionnalités offertes par les modules* `shutil` *et* `os`. *Les page http://docs.python.org/library/shutil.html et http://docs.python.org/library/os.html référencent toutes les fonctions de ces modules.*

une fonction plus intéressante qui permet de retourner la liste des éléments d'un répertoire en appliquant un filtre. Le programme suivant permet par exemple de retourner la liste des fichiers et des répertoires inclus dans un répertoire.

```
# coding: latin-1
import glob
import os.path

def liste_fichier_repertoire (folder, filter) :
    # résultats
    file,fold = [], []

    # recherche des fichiers obéissant au filtre
    res = glob.glob (folder + "\\" + filter)

    # on inclut les sous-répertoires qui n'auraient pas été
    # sélectionnés par le filtre
    rep = glob.glob (folder + "\\*")
    for r in rep :
        if r not in res and os.path.isdir (r) :
            res.append (r)

    # on ajoute fichiers et répertoires aux résultats
    for r in res :
        path = r
        if os.path.isfile (path) :
            # un fichier, rien à faire à part l'ajouter
            file.append (path)
        else :
            # sous-répertoire : on appelle à nouveau la fonction
            # pour retourner la liste des fichiers inclus
            fold.append (path)
            fi,fo = liste_fichier_repertoire (path, filter)
            file.extend (fi)  # on étend la liste des fichiers
            fold.extend (fo)  # on étend la liste des répertoires
    # fin
    return file,fold

folder = r"D:\Dupre\_data\informatique"
filter = "*.tex"
file,fold = liste_fichier_repertoire (folder, filter)

for f in file : print "fichier ", f
for f in fold : print "répertoire ", f
```

Le programme repose sur l'utilisation d'une fonction récursive qui explore d'abord le premier répertoire. Elle se contente d'ajouter à une liste les fichiers qu'elle découvre puis cette fonction s'appelle elle-même sur le premier sous-répertoire qu'elle rencontre. La fonction `walk` permet d'obtenir la liste des fichiers et des sous-répertoire. Cette fonction parcourt automatiquement les sous-répertoires inclus, le programme est plus court mais elle ne prend pas en compte le filtre qui peut être alors pris en compte grâce aux expressions régulières (paragraphe 7.6).

```
# coding: latin-1
import os

def liste_fichier_repertoire (folder) :
```

```
    file, rep = [], []
    for r, d, f in os.walk (folder) :
        for a in d : rep.append (r + "/" + a)
        for a in f : file.append (r + "/" + a)
    return file, rep

folder = r"D:\Dupre\_data\informatique"
file,fold = liste_fichier_repertoire (folder)

for f in file : print "fichier ", f
for f in fold : print "répertoire ", f
```

## 7.4 Format binaire

Ecrire et lire des informations au travers d'un fichier texte revient à convertir les informations quel que soit leur type dans un format lisible pour tout utilisateur. Un entier est écrit sous forme de caractères décimaux alors que sa représentation en mémoire est binaire. Cette conversion dans un sens puis dans l'autre est parfois jugée coûteuse en temps de traitement et souvent plus gourmande en terme de taille de fichiers [8] même si elle permet de relire les informations écrites grâce à n'importe quel éditeur de texte. Il est parfois plus judicieux pour une grande masse d'informations d'utiliser directement le format binaire, c'est-à-dire celui dans lequel elles sont stockées en mémoire. Les informations apparaissent dans leur forme la plus simple pour l'ordinateur : une suite d'octets (bytes en anglais). Deux étapes vont intervenir que ce soit pour l'écriture :

1. On récupère les informations dans une suite d'octets (fonction `pack` du module `struct`).
2. On les écrit dans un fichier (méthode `write` affiliée aux fichiers).

Ou la lecture :

1. On lit une suite d'octets depuis un fichier (méthode `read` affiliée aux fichiers).
2. On transforme cette suite d'octets pour retrouver l'information qu'elle formait initialement (fonction `unpack` du module `struct`).

L'utilisation de fichiers binaires est moins évidente qu'il n'y paraît et il faut faire appel à des modules spécialisés alors que la gestion des fichiers texte ne pose aucun problème. Cela vient du fait que *Python* ne donne pas directement accès à la manière dont sont stockées les informations en mémoire contrairement à des langages tels que le *C++*. L'intérêt de ces fichiers réside dans le fait que l'information qu'ils contiennent prend moins de place stockée en binaire plutôt que convertie en chaînes de caractères au format texte [9].

L'écriture et la lecture d'un fichier binaire soulèvent les mêmes problèmes que pour un fichier texte : il faut organiser les données avant de les enregistrer pour savoir comment les retrouver. Les types immuables (réel, entier, caractère) sont assez simples à

8. Un fichier texte compressé, au format `zip` par exemple, est une alternative aux fichiers binaires en terme de taille mais il allonge la lecture et l'écriture par des étapes de compression et de décompression.

9. Par exemple, un réel est toujours équivalent à huit caractères en format binaire alors que sa conversion au format texte va souvent jusqu'à quinze caractères.

gérer dans ce format. Pour les objets complexes, *Python* propose une solution grâce au module `pickle`.

### 7.4.1 Ecriture dans un fichier binaire

L'écriture d'un fichier binaire commence par l'ouverture du fichier en mode écriture par l'instruction `file = open ("<nom_fichier>", "wb")`. C'est le code "`wb`" qui est important (*w* pour *write*, *b* pour *binary*), il spécifie le mode d'ouverture "`w`" et le format "`b`". La fermeture est la même que pour un fichier texte.

Le module `struct` et la fonction `pack` permet de convertir les informations sous forme de chaîne de caractères avant de les enregistrer au format binaire. La fonction `pack` construit une chaîne de caractères égale au contenu de la mémoire. Son affichage avec la fonction `print` produit quelque chose d'illisible le plus souvent. Le tableau suivant montre les principaux formats de conversion [10].

| code | type correspondant |
|---|---|
| `c` | caractère |
| `B` | caractère non signé (octet) |
| `i` | entier (4 octets) |
| `I` | entier non signé (4 octets) |
| `d` | double (8 octets) |

L'utilisation de ces codes est illustrée au paragraphe 7.4.3.

### 7.4.2 Lecture d'un fichier binaire

Le code associé à l'ouverture d'un fichier binaire en mode lecture est "`rb`", cela donne : `file = open ("<nom_fichier>", "rb")`. La lecture utilise la fonction `unpack` pour effectuer la conversion inverse, celle d'une chaîne de caractères en entiers, réels, ... Le paragraphe suivant illustre la lecture et l'écriture au format binaire.

### 7.4.3 Exemple complet

Cet exemple crée un fichier "`info.bin`" puis écrit des informations à l'intérieur. Il ne sera pas possible d'afficher le contenu du fichier à l'aide d'un éditeur de texte. La méthode `pack` accepte un nombre de paramètres variable [11], ceci explique la syntaxe `struct.pack("cccc", *s)`.

```
# coding: latin-1
import struct
# on enregistre un entier, un réel et 4 caractères
i = 10
x = 3.1415692
s = "ABCD"

# écriture
file = open ("info.bin", "wb")
```

10. La liste complète figure à l'adresse http://docs.python.org/library/struct.html.
11. voir également paragraphe 3.4.10, page 81

```
file.write ( struct.pack ("i" , i) )
file.write ( struct.pack ("d" , x) )
file.write ( struct.pack ("cccc" , *s) )
file.close ()

# lecture
file = open ("info.bin", "rb")
i = struct.unpack ("i",      file.read (4))
x = struct.unpack ("d",      file.read (8))
s = struct.unpack ("cccc",   file.read (4))
file.close ()

# affichage pour vérifier que les données ont été bien lues
print i  # affiche (10,)
print x  # affiche (3.1415692000000002,)
print s  # affiche ('A', 'B', 'C', 'D')
```

Les résultats de la méthode `unpack` apparaissent dans un tuple mais les données sont correctement récupérées. Ce programme fait aussi apparaître une des particularité du format binaire. On suppose ici que la chaîne de caractères est toujours de longueur 4. En fait, pour stocker une information de dimension variable, il faut d'abord enregistrer cette dimension puis s'en servir lors de la relecture pour connaître le nombre d'octets à lire. On modifie le programme précédent pour sauvegarder une chaîne de caractères de longueur variable.

```
# coding: latin-1
import struct
# on enregistre un entier, un réel et n caractères
i = 10
x = 3.1415692
s = "ABCDEDF"

# écriture
file = open ("info.bin", "wb")
file.write ( struct.pack ("i" , i) )
file.write ( struct.pack ("d" , x) )
file.write ( struct.pack ("i" , len(s)) )  # on sauve la dimension de s
file.write ( struct.pack ("c" * len(s) , *s) )
file.close ()

# lecture
file = open ("info.bin", "rb")
i = struct.unpack ("i",      file.read (4))
x = struct.unpack ("d",      file.read (8))
l = struct.unpack ("i",      file.read (4)) # on récupère la dimension de s
l = l [0]  # l est un tuple, on s'intéresse à son unique élément
s = struct.unpack ("c" * l,  file.read (l))
file.close ()

# affichage pour contrôler
print i  # affiche (10,)
print x  # affiche (3.1415692000000002,)
print s  # affiche ('A', 'B', 'C', 'D', 'E', 'D', 'F')
```

Cette méthode utilisée pour les chaînes de caractères est applicable aux listes et aux dictionnaires de longueur variable : il faut d'abord stocker leur dimension. Il faut retenir également que la taille d'un réel est de huit octets, celle d'un entier de quatre

octets et celle d'un caractère d'un octet [12]. Cette taille doit être passée en argument à la méthode `read`.

### 7.4.4 Objets plus complexes

Il existe un moyen de sauvegarder dans un fichier des objets plus complexes à l'aide du module `pickle`. Celui-ci permet de stocker dans un fichier le contenu d'un dictionnaire à partir du moment où celui-ci contient des objets standard du langage *Python*. Le principe pour l'écriture est le suivant :

```
import pickle

dico = {'a': [1, 2.0, 3, "e"], 'b': ('string', 2), 'c': None}
lis  = [1, 2, 3]

f = open ('data.bin', 'wb')
pickle.dump (dico, f)
pickle.dump (lis, f)
f.close()
```

La lecture est aussi simple :

```
f = open ('data.bin', 'rb')
dico = pickle.load (f)
lis  = pickle.load (f)
f.close()
```

Un des avantages du module `pickle` est de pouvoir gérer les références circulaires : il est capable d'enregistrer et de relire une liste qui se contient elle-même, ce peut être également une liste qui en contient une autre qui contient la première...

Le module `pickle` peut aussi gérer les classes définies par un programmeur à condition qu'elles puissent convertir leur contenu en un dictionnaire dans un sens et dans l'autre, ce qui correspond à la plupart des cas.

```
import pickle
import copy

class Test :
    def __init__ (self) :
        self.chaine = "a"
        self.entier = 5
        self.tuple  = { "h":1, 5:"j" }

t = Test ()

f = open('data.bin', 'wb')  # lecture
pickle.dump (t, f)
```

12. Cette règle est toujours vrai sur des ordinateurs *32 bits*. Cette taille varie sur les ordinateurs 64 bits qui commencent à se répandre. Le programme suivant donnera la bonne réponse.

```
from struct import pack
print len(pack('i',0)) # longueur d'un entier
print len(pack('d',0)) # longueur d'un réel
print len(pack('c',"e")) # longueur d'un caractère
```

```
f.close()

f = open('data.bin', 'rb')  # écriture
t = pickle.load (f)
f.close()
```

Lorsque la conversion nécessite un traitement spécial, il faut surcharger les opérateurs `__getstate__` et `__setstate__`. Ce cas se produit par exemple lorsqu'il n'est pas nécessaire d'enregistrer tous les attributs de la classe car certains sont calculés ainsi que le montre l'exemple suivant :

```
import pickle
import copy

class Test :
    def __init__ (self) :
        self.x = 5
        self.y = 3
        self.calcule_norme ()   # attribut calculé
    def calcule_norme (self) :
        self.n = (self.x ** 2 + self.y ** 2) ** 0.5
    def __getstate__ (self) :
        """conversion de Test en un dictionnaire"""
        d = copy.copy (self.__dict__)
        del d ["n"]  # attribut calculé, on le sauve pas
        return d
    def __setstate__ (self,dic) :
        """conversion d'un dictionnaire dic en Test"""
        self.__dict__.update (dic)
        self.calcule_norme ()  # attribut calculé

t = Test ()

f = open('data.bin', 'wb')  # lecture
pickle.dump (t, f)
f.close()

f = open('data.bin', 'rb')  # écriture
t = pickle.load (f)
f.close()
```

## 7.5 Paramètres en ligne de commande

Certains développements nécessitent de pouvoir exécuter un programme *Python* en ligne de commande sans intervention extérieure. Ce peut être un programme de synchronisation de deux répertoires ou la récupération du contenu de pages HTML, des tâches souvent exécutées la nuit à une heure où personne n'est présent pour le faire. Prenons par exemple le cas d'un programme très simple de synchronisation qui recopie tous les fichiers d'un répertoire vers un autre. La fonctionne `copie_repertoire` effectue cette tâche.

```
# coding: latin-1
import glob
import shutil
```

```
def copie_repertoire (rep1, rep2) :
    """copie tous les fichiers d'un répertoire rep1 vers un autre rep2"""
    li = glob.glob (rep1 + "/*.*")   # récupère dans une liste fichiers et
                                     # répertoires qui respectent le filtre
    for l in li :
        to = l.replace (rep1, rep2)  # nom du fichier copié
                                     # (on remplace rep1 par rep2)
        shutil.copy (l, to)
copie_repertoire ("c:/original", "c:/backup")
```

Cette tâche est en plus exécutée sur deux répertoires et on ne voudrait pas avoir deux programmes différents alors que la tâche est la même quelque soit le couple de répertoire choisi. On souhaite pouvoir lancer le programme *Python* et lui spécifier les deux répertoires, c'est-à-dire être capable de lancer le programme comme suit (voir également figure 7.1) :

```
c:\python26\python.exe synchro.py c:/original c:/backup
```

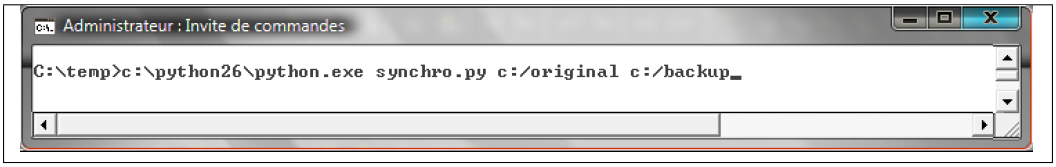



**Figure 7.1 :** *Un programme Python lancé en ligne de commande.*

Pour lancer un programme *Python* en ligne de commande, il faut écrire dans une fenêtre de commande ou un programme d'extension *bat* les instructions suivantes séparées par des espaces :

```
interpréteur_python programme_python argument1 argument2 ...
```

Il faut maintenant récupérer ces paramètres au sein du programme *Python* grâce au module `sys` qui contient une variable `argv` contenant la liste des paramètres de la ligne de commande. S'il y a deux paramètres, cette liste `argv` contiendra trois éléments :

1. nom du programme
2. paramètre 1
3. paramètre 2

Le programme suivante utilise ce mécanisme avec le programme `synchro` contenant la fonction `copie_repertoire` :

```
# coding: latin-1
import glob
import shutil
def copie_repertoire (rep1, rep2) :
    """copie tous les fichiers d'un répertoire rep1 vers un autre rep2"""
    li = glob.glob (rep1 + "/*.*")
    for l in li :
        to = l.replace (rep1, rep2)  # nom du fichier copié
                                     # (on remplace rep1 par rep2)
        shutil.copy (l, to)
```

```
import sys
                      # sys.argv [0] --> nom du programme (ici, synchro.py)
rep1 = sys.argv [1]   # récupération du premier paramètre
rep2 = sys.argv [2]   # récupération du second paramètre
copie_repertoire (rep1, rep2)
```

**Remarque 7.9 : paramètres contenant des espaces**
Lorsque les paramètres écrits sur la ligne de commande contiennent eux-mêmes des espaces, il faut les entourer avec des guillemets.

```
c:\python26\python.exe c:\batch\synchro.py
         "c:\Program Files\source" "c:\Program Files\backup"
```

Pour des besoins plus complexes, le module `getopt` propose des fonctions plus élaborées pour traiter les paramètres de la ligne de commande. Il est également possible de lancer un programme en ligne de commande depuis *Python* grâce à la fonction `system` du module `os` comme ceci :

```
import os
os.system ("c:\python26\python.exe c:\batch\synchro.py " \
           "\"c:\Program Files\source\" \"c:\Program Files\backup\"")
```

Le programme suivant est un autre exemple de l'utilité de la fonction `system`. Il écrit une matrice au format *HTML*[13] puis utilise la fonction `system` pour lancer le navigateur *Firefox* ou *Internet Explorer* afin de pouvoir visualiser la page créée.

```
mat = ["Victor Hugo 6".split (), "Marcel Proust 3".split () ]
f = open ("tableau.html", "w")
f.write ("<body><html>\n")
f.write ("<table border=\"1\">\n")
for m in mat :
    f.write ("<tr>")
    for c in m :
        f.write ("<td>" + c + "</td>")
    f.write ("</tr>\n")
f.write ("</table>")
f.close ()

import os
os.system ("\"C:\\Program Files\\Mozilla Firefox\\firefox.exe\" tableau.html")
os.system ("\"C:\\Program Files\\Internet Explorer\\iexplore.exe\"" \
           " d:\\temp\\tableau.html")
```

## 7.6 Expressions régulières

### 7.6.1 Premiers pas

Chercher un mot dans un texte est une tâche facile, c'est l'objectif de la méthode `find` attachée aux chaînes de caractères, elle suffit encore lorsqu'on cherche un mot

13. La description du langage *HTML* sort du cadre de ce livre mais la requête `HTML syntaxe` effectuée sur n'importe quel moteur de recherche Internet retournera des résultats intéressants.

au pluriel ou au singulier mais il faut l'appeler au moins deux fois pour chercher ces deux formes. Pour des expressions plus compliquées, il est conseillé d'utiliser les *expressions régulières*. C'est une fonctionnalité qu'on retrouve dans beaucoup de langages. C'est une forme de grammaire qui permet de rechercher des expressions.

Lorsqu'on remplit un formulaire, on voit souvent le format "MM/JJ/AAAA" qui précise sous quelle forme on s'attend à ce qu'une date soit écrite. Les expressions régulières permettent de définir également ce format et de chercher dans un texte toutes les chaînes de caractères qui sont conformes à ce format.

La liste qui suit contient des dates de naissance. On cherche à obtenir toutes les dates de cet exemple sachant que les jours ou les mois contiennent un ou deux chiffres, les années deux ou quatre.

```
s = """date 0 : 14/9/2000
date 1 : 20/04/1971     date 2 : 14/09/1913     date 3 : 2/3/1978
date 4 : 1/7/1986     date 5 : 7/3/47     date 6 : 15/10/1914
date 7 : 08/03/1941     date 8 : 8/1/1980     date 9 : 30/6/1976"""
```

Le premier chiffre du jour est soit 0, 1, 2, ou 3 ; ceci se traduit par $[0-3]$. Le second chiffre est compris entre 0 et 9, soit $[0-9]$. Le format des jours est traduit par $[0-3][0-9]$. Mais le premier jour est facultatif, ce qu'on précise avec le symbole ? : $[0-3]?[0-9]$. Les mois suivent le même principe : $[0-1]?[0-9]$. Pour les années, ce sont les deux premiers chiffres qui sont facultatifs, le symbole ? s'appliquent sur les deux premiers chiffres, ce qu'on précise avec des parenthèses : ([0-2][0-9]) ?[0-9][0-9]. Le format final d'une date devient :

```
[0-3]?[0-9]/[0-1]?[0-9]/([0-2][0-9])?[0-9][0-9]
```

Le module `re` gère les expressions régulières, celui-ci traite différemment les parties de l'expression régulière qui sont entre parenthèses de celles qui ne le sont pas : c'est un moyen de dire au module `re` que nous nous intéressons à telle partie de l'expression qui est signalée entre parenthèses. Comme la partie qui nous intéresse - une date - concerne l'intégralité de l'expression régulière, il faut insérer celle-ci entre parenthèses.

La première étape consiste à construire l'expression régulière, la seconde à rechercher toutes les fois qu'un morceau de la chaîne `s` définie plus haut correspond à l'expression régulière.

```
import re
# première étape : construction
expression = re.compile ("([0-3]?[0-9]/[0-1]?[0-9]/([0-2][0-9])?[0-9][0-9])")
# seconde étape  : recherche
res = expression.findall (s)
print res
```

Le résultat de ce programme est le suivant :

```
[('14/9/2000', '20'), ('20/04/1971', '19'),  ('14/09/1913', '19'),
 ('2/3/1978', '19'),  ('1/7/1986', '19'),    ('7/3/47', ''),
 ('15/10/1914', '19'), ('08/03/1941', '19'), ('8/1/1980', '19'),
 ('30/6/1976', '19')]
```

C'est une liste de couples dont chaque élément correspond aux parties comprises entre parenthèses qu'on appelle des *groupes*. Lorsque les expressions régulières sont utilisées, on doit d'abord se demander comment définir ce qu'on cherche puis quelles fonctions utiliser pour obtenir les résultats de cette recherche. Les deux paragraphes qui suivent y répondent.

### 7.6.2 Syntaxe

La syntaxe des expressions régulières est décrite sur le site officiel de *Python*[14]. Comme toute grammaire, celle des expressions régulières est susceptible d'évoluer au fur et à mesure des versions du langage *Python*.

#### 7.6.2.1 Les ensembles de caractères

Lors d'une recherche, on s'intéresse aux caractères et souvent aux classes de caractères : on cherche un chiffre, une lettre, un caractère dans un ensemble précis ou un caractère qui n'appartient pas à un ensemble précis. Certains ensembles sont prédéfinis, d'autres doivent être définis à l'aide de crochets.

Pour définir un ensemble de caractères, il faut écrire cet ensemble entre crochets : [0123456789] désigne un chiffre. Comme c'est une séquence de caractères consécutifs, on peut résumer cette écriture en `[0-9]`. Pour inclure les symboles −, +, il suffit d'écrire : `[-0-9+]`. Il faut penser à mettre le symbole − au début pour éviter qu'il ne désigne une séquence.

Le caractère `^` inséré au début du groupe signifie que le caractère cherché ne doit pas être un de ceux qui suivent. Le tableau suivant décrit les ensembles prédéfinis [15] et leur équivalent en terme d'ensemble de caractères :

| | |
|---|---|
| `.` | désigne tout caractère non spécial quel qu'il soit |
| `\d` | désigne tout chiffre, est équivalent à `[0-9]` |
| `\D` | désigne tout caractère différent d'un chiffre, est équivalent à `[^0-9]` |
| `\s` | désigne tout espace ou caractère approché, est équivalent à [ `\t\n\r\f\v`] [16] |
| `\S` | désigne tout caractère différent d'un espace, est équivalent à [`^\t\n\r\f\v`] |
| `\w` | désigne tout lettre ou chiffre, est équivalent à `[a-zA-Z0-9_]` |
| `\W` | désigne tout caractère différent d'une lettre ou d'un chiffre, est équivalent à `[^a-zA-Z0-9_]` |
| `^` | désigne le début d'un mot sauf s'il est placé entre crochets |
| `$` | désigne la fin d'un mot sauf s'il est placé entre crochets |

A l'instar des chaînes de caractères, comme le caractère \ est un caractère spécial, il faut le doubler : [\\]. Avec ce système, le mot "taxinomie" qui accepte deux orthographes s'écrira : `tax[io]nomie`.

14. La page http://docs.python.org/library/re.html#regular-expression-syntax décrit la syntaxe, la page http://docs.python.org/howto/regex.html#regex-howto décrit comment se servir des expressions régulières, les deux pages sont en anglais.

15. La page http://docs.python.org/library/re.html#regular-expression-syntax les recense tous.

16. Ces caractères sont spéciaux, les plus utilisés sont `\t` qui est une tabulation, `\n` qui est une fin de ligne et qui `\r` qui est un retour à la ligne.

**Remarque 7.10 : caractères spéciaux et expressions régulières**
Le caractère \ est déjà un caractère spécial pour les chaînes de caractères en *Python*, il faut donc le quadrupler pour l'insérer dans un expression régulière. L'expression suivante filtre toutes les images dont l'extension est `png` et qui sont enregistrées dans un répertoire `image`.

```
s = r"D:\Dupre\_data\informatique\support\vba\image/vbatd1_4.png"
print re.compile ( "[\\\\/]image[\\\\/].*[.]png").search(s) # résultat positif
print re.compile (r"[\\/]image[\\/].*[.]png").search(s)     # même résultat
```

#### 7.6.2.2 Les multiplicateurs

Les multiplicateurs permettent de définir des expressions régulières comme : un mot entre six et huit lettres qu'on écrira [\w]{6,8}. Le tableau suivant donne la liste des multiplicateurs principaux [17] :

| | |
|---|---|
| $*$ | présence de l'ensemble de caractères qui précède entre 0 fois et l'infini |
| $+$ | présence de l'ensemble de caractères qui précède entre 1 fois et l'infini |
| ? | présence de l'ensemble de caractères qui précède entre 0 et 1 fois |
| {m,n} | présence de l'ensemble de caractères qui précède entre m et n fois, si m = n, cette expression peut être résumée par {n}. |
| (?!(...)) | absence du groupe désigné par les points de suspensions (voir paragraphe 7.6.4). |

L'algorithme des expressions régulières essaye toujours de faire correspondre le plus grand morceau à l'expression régulière. Par exemple, dans la chaîne `<h1>mot</h1>`, `<.*>` correspond avec `<h1>`, `</h1>` ou encore `<h1>mot</h1>`. Par conséquent, l'expression régulière correspond à trois morceaux. Par défaut, il prendra le plus grand. Pour choisir les plus petits, il faudra écrire les multiplicateurs comme ceci : $*?$, $+?$, $??$.

```
import re
s = "<h1>mot</h1>"
print re.compile ("(<.*>)") .match (s).groups () # ('<h1>mot</h1>',)
print re.compile ("(<.*?>)").match (s).groups () # ('<h1>',)
```

#### 7.6.2.3 Groupes

Lorsqu'un multiplicateur s'applique sur plus d'un caractère, il faut définir un groupe à l'aide de parenthèses. Par exemple, le mot `yoyo` s'écrira : (yo){2}. Les parenthèses jouent un rôle similaire à celui qu'elles jouent dans une expression numérique. Tout ce qui est compris entre deux parenthèses est considéré comme un groupe.

#### 7.6.2.4 Assembler les caractères

On peut assembler les groupes de caractères les uns à la suite des autres. Dans ce cas, il suffit de les juxtaposer comme pour trouver les mots commençant par s : s[a − z]$*$.

17. La page http://docs.python.org/library/re.html#regular-expression-syntax les recense tous.

On peut aussi chercher une chaîne ou une autre grâce au symbole |. Chercher dans un texte l'expression *Xavier Dupont* ou *M. Dupont* s'écrira : (Xavier)|(M[.])Dupont.

### 7.6.3 Fonctions

La fonction compile du module re permet de construire un objet "expression régulière". A partir de cet objet, on peut vérifier la correspondance entre une expression régulière et une chaîne de caractères (méthode match). On peut chercher une expression régulière (méthode search). On peut aussi remplacer une expression régulière par une chaîne de caractères (méthode sub). La table 7.3 récapitule ces méthodes.

| | |
|---|---|
| match(s[, pos[, end]]) | Vérifie la correspondance entre l'expression régulière et la chaîne s. Il est possible de n'effectuer cette vérification qu'entre les caractères dont les positions sont pos et end. La fonction retourne None s'il n'y a pas de correspondance et sinon un objet de type Match. |
| search(s[, pos[, end]]) | Fonction semblable à match, au lieu de vérifier la correspondance entre toute la chaîne et l'expression régulière, cherche la première chaîne de caractères extraite correspondant à l'expression régulière. |
| split(s [, maxsplit = 0]) | Recherche toutes les chaînes de caractères extraites qui vérifient l'expression régulière puis découpe cette chaîne en fonction des expressions trouvées. La méthode split d'une chaîne de caractère permet de découper selon un séparateur. La méthode split d'une expression régulière permet de découper selon plusieurs séparateurs. C'est pratique pour découper une chaîne de caractères en colonnes séparées par ; ou une tabulation. re.compile("[\t;]").split("a;b\tc;g") donne ["a","b","c","g"]. |
| findall(s[, pos[, end]]) | Identique à split mais ne retourne pas les morceaux entre les chaînes extraites qui vérifient l'expression régulière. |
| sub(repl, s [, count = 0]) | Remplace dans la chaîne repl les éléments \1, \2, ... par les parties de s qui valident l'expression régulière. |
| flags | Mémorise les options de construction de l'expression régulière. C'est un attribut. |
| pattern | Chaîne de caractères associée à l'expression régulière. C'est un attribut. |

**Table 7.3 :** *Liste non exhaustive des méthodes et attributs qui s'appliquent à un objet de type "expression régulière" retourné par la fonction* compile *du module* re. *La page http://docs.python.org/library/re.html contient la documentation associée au module* re.

Les méthodes search et match retournent toutes des objets Match dont les méthodes sont présentées par la table 7.4. Appliquées à l'exemple décrit page 185 concernant les dates, cela donne :

```
expression = re.compile ("([0-3]?[0-9]/[0-1]?[0-9]/([0-2][0-9])?[0-9][0-9])[^\d]")
print expression.search (s).group(1,2) # affiche ('14/9/2000', '20')
```

```
c = expression.search (s).span(1)      # affiche (9, 18)
print s [c[0]:c[1]]                    # affiche 14/9/2000
```

| | |
|---|---|
| group([g1, ...]) | Retourne la chaîne de caractères validée par les groupes g1... |
| groups([default]) | Retourne la liste des chaînes de caractères qui ont été validées par chacun des groupes. |
| span([gr]) | Retourne les positions dans la chaîne originale des chaînes extraites validées le groupe gr. |

**Table 7.4 :** *Liste non exhaustive des méthodes qui s'appliquent à un objet de type* `Match` *qui est le résultat des méthodes* `search` *et* `match`*. Les groupes sont des sous-parties de l'expression régulière, chacune d'entre elles incluses entre parenthèses. Le* $n^{ème}$ *correspond au groupe qui suit la* $n^{ème}$ *parenthèse ouvrante. Le premier groupe a pour indice 1. La page http://docs.python.org/library/re.html contient la documentation associée au module* `re`.

### 7.6.4 Exemple plus complet

L'exemple suivant présente trois cas d'utilisation des expressions régulières. On s'intéresse aux titres de chansons *MP3* stockées dans un répertoire. Le module `mutagen`[18] permet de récupérer certaines informations concernant un fichier *MP3* dont le titre, l'auteur et la durée.

Le premier problème consiste à retrouver les chansons sans titre ou dont le titre contient seulement son numéro : *track03*, *track - 03*, *audiotrack 03*, *track 03*, *piste 03*, *piste - 03*, *audiopiste 03*, ... Ce titre indésirable doit valider l'expression régulière suivante : `^(((audio)?track( )?( - )?[0-9]1,2)|(piste [0-9]1,2))$`.

Le second problème consiste à retrouver toutes les chansons dont le titre contient le mot *heart* mais ni *heartland* ni *heartache*. Ce titre doit valider l'expression régulière : `((heart)(?!((ache)|(land))))`.

Le troisième problème consiste à compter le nombre de mots d'un titre. Les mots sont séparés par l'ensemble de caractères [–, ; !'.?& :]. On utilise la méthode `split` pour découper en mots. Le résultat est illustré par le programme suivant.

```
# coding: latin-1
import mutagen.mp3, mutagen.easyid3, os, re

def infoMP3 (file, tags) :
    """retourne des informations sur un fichier MP3 sous forme de
    dictionnaire (durée, titre, artiste, ...)"""
    a = mutagen.mp3.MP3(file)
    b = mutagen.easyid3.EasyID3(file)
    info = { "minutes":a.info.length/60, "nom":file }
    for k in tags :
        try : info [k] = str (b [k][0])
        except : continue
    return info

def all_files (repertoire, tags, ext = re.compile (".mp3$")) :
```

18. http://code.google.com/p/quodlibet/wiki/Development/Mutagen

```
    """retourne les informations pour chaque fichier d'un répertoire"""
    all = []
    for r, d, f in os.walk (repertoire) :
        for a in f :
            if not ext.search (a) : continue
            t = infoMP3 (r + "/" + a, tags)
            if len (t) > 0 : all.append (t)
    return all

def heart_notitle_mots (all, avoid,sep,heart) :
    """retourne trois résultats
    - les chansons dont le titre valide l'expression régulière heart
    - les chansons dont le titre valide l'expression régulière avoid
    - le nombre moyen de mots dans le titre d'une chanson"""
    liheart, notitle  = [], []
    nbmot,nbsong      = 0,0
    for a in all :
        if "title" not in a :
            notitle.append (a)
            continue
        ti = a ["title"].lower ()
        if avoid.match (ti) :
            notitle.append (a)
            continue
        if heart.search (ti) : liheart.append (a)
        nbsong += 1
        nbmot  += len ([ m for m in sep.split (ti) if len (m) > 0 ])
    return liheart, notitle, float (nbmot)/nbsong

tags  = "title album artist genre tracknumber".split ()
all = all_files (r"D:\musique", tags)

avoid = re.compile ("^(((audio)?track( )?( - )?[0-9]{1,2})|(piste [0-9]{1,2}))$")
sep   = re.compile ("[- ,;!'.?&:]")
heart = re.compile ("((heart)(?!((ache)|(land))))")
liheart, notitle, moymot = heart_notitle_mots (all, avoid, sep, heart)

print "nombre de mots moyen par titre ", moymot
print "somme des durée contenant heart ", sum ( [ s ["minutes"] for s in liheart] )
print "chanson sans titre ", len (notitle)
print "liste des titres "
for s in liheart : print "   ",s ["title"]
```

**Remarque 7.11 : Nommer des groupes**

Une expression régulière ne sert pas seulement de filtre, elle permet également d'extraire le texte qui correspond à chaque groupe, à chaque expression entre parenthèses. L'exemple suivant montre comment récupérer le jour, le mois, l'année à l'intérieur d'une date.

```
import re
date = "05/22/2010"
exp  = "([0-9]{1,2})/([0-9]{1,2})/(((19)|(20))[0-9]{2})"
com  = re.compile (exp)
print com.search (date).groups ()    # ('05', '22', '2010', '20', None, '20')
```

Il n'est pas toujours évident de connaître le numéro du groupe qui contient l'information à extraire. C'est pourquoi il paraît plus simple de les nommer afin de les

récupérer sous la forme d'un dictionnaire et non plus sous forme de tableau. La syntaxe `( ?P<nom_du_groupe>expression)` permet de nommer un groupe. Elle est appliquée à l'exemple précédent.

```
exp  = "(?P<jj>[0-9]{1,2})/(?P<mm>[0-9]{1,2})/(?P<aa>((19)|(20))[0-9]{2})"
com  = re.compile (exp)
print com.search (date).groupdict () # {'mm': '22', 'aa': '2010', 'jj': '05'}
```

## 7.7 Dates

Le module `datetime` [19] fournit une classe `datetime` qui permet de faire des opérations et des comparaisons sur les dates et les heures. L'exemple suivant calcule l'âge d'une personne née le 11 août 1975.

```
import datetime
naissance = datetime.datetime (1975,11,8,10,0,0)
jour = naissance.now () # obtient l'heure et la date actuelle
print jour              # affiche 2010-05-22 11:24:36.312000
age = jour - naissance  # calcule une différence
print age               # affiche 12614 days, 1:25:10.712000
```

L'objet `datetime` autorise les soustractions et les comparaisons entre deux dates. Une soustraction retourne un objet de type `timedelta` qui correspond à une durée. qu'on peut multiplier par un réel ou ajouter à un objet de même type ou à un objet de type `datetime`. L'utilisation de ce type d'objet évite de se pencher sur tous les problèmes de conversion.

Le module `calendar` est assez pratique pour construire des calendriers. Le programme ci-dessous affiche une liste de t-uples incluant le jour et le jour de la semaine du mois d'août 1975. Dans cette liste, on y trouve le t-uple $(11, 0)$ qui signifie que le 11 août 1975 était un lundi. Cela permet de récupérer le jour de la semaine d'une date de naissance.

```
import calendar
c = calendar.Calendar ()
for d in c.itermonthdays2 (1975,8) : print d
```

## 7.8 Problème de jeux de caractères

La langue anglaise est la langue dominante en ce qui concerne l'informatique mais cela n'empêche pas que des programmes anglais manipulent du japonais même si le nombre de caractères est beaucoup plus grand. Les jeux de caractères proposent une solution à ce problème : un jeu de caractères définit la façon de décoder une suite d'octets [20]. Les langues latines n'ont besoin que d'un octet pour coder un caractère,

19. Voir également la page http://docs.python.org/library/datetime.html.

20. On confond souvent jeu de caractères et *encodage*. Le jeu de caractère désigne l'ensemble de caractères dont le programme a besoin, l'encodage décrit la manière dont on passe d'une séquence de caractères français, japonais, anglais à une séquence d'octets qui est la seule information manipulée par un ordinateur.

les langues asiatiques en ont besoin de plusieurs. Il n'existe pas qu'un seul jeu de caractères lorsqu'on programme. Ils interviennent à plusieurs endroits différents :

1. Le jeu de caractères utilisé par l'éditeur de texte pour afficher le programme.
2. Le jeu de caractères du programme, par défaut `ascii` mais il peut être changé en insérant une première ligne de commentaire (voir paragraphe 1.5.2, page 22). Les chaînes de caractères du programme sont codées avec ce jeu de caractères. Ce jeu devrait être identique à celui utilisé par l'éditeur de texte afin d'éviter les erreurs.
3. Le jeu de caractères de la sortie, utilisé pour chaque instruction `print`, il est désigné par le code `cp1252` sur un système `Windows`.
4. Le jeu de caractères dans lequel les chaînes de caractères sont manipulées. Un jeu standard qui permet de représenter toutes les langues est le jeu de caractères `utf` $-$ `8`. Il peut être différent pour chaque variable.
5. Le jeu de caractères d'un fichier texte. Il peut être différent pour chaque fichier.

Le langage *Python* offre deux classes pour représenter les chaînes de caractères. La classe `str` qui est adaptée aux langues latines, le jeu de caractères n'est pas précisé. La classe `unicode` représente un caractère sur un à quatre octets avec un jeu de caractères désigné par l'appellation `unicode`. Il est impératif de savoir quel jeu est utilisé à quel endroit et il faut faire des conversions de jeux de caractères pour passer l'information d'un endroit à un autre. Il existe deux méthodes pour cela présentées par la table 7.5.

```
# coding: latin-1
st =  "eé"
su = u"eé"  # raccourci pour su = unicode ("eé", "latin-1")
            # ou encore      su = unicode ("eé".decode ("latin-1"))

print type (st)                                    # affiche <type 'str'>
print type (su)                                    # affiche <type 'unicode'>
print len (st),        ";", st                     # affiche  2 ; eé
print len (repr (st)), ";", repr (st)              # affiche  7 ; 'e\xe9'
print len (su),        ";", su.encode ("latin-1")  # affiche  2 ; eé
print len (repr (su)), ";", repr (su)              # affiche  8 ; u'e\xe9'
```

Lorsqu'on manipule plusieurs jeux de caractères, il est préférable de conserver un unique jeu de référence pour le programme par l'intermédiaire de la classe `unicode`. Il "suffit" de gérer les conversions depuis ces chaînes de caractères vers les entrées sorties du programme comme les fichiers texte. Par défaut, ceux-ci sont écrits avec le jeu de caractères du système d'exploitation. Dans ce cas, la fonction `open` suffit. En revanche, si le jeu de caractères est différent, il convient de le préciser lors de l'ouverture du fichier. On utilise la fonction `open` du module `codecs` qui prend comme paramètre supplémentaire le jeu de caractères du fichier. Toutes les chaînes de caractères seront lues et converties au format `unicode`.

```
import codecs
f = codecs.open ("essai.txt", "r", "cp1252")  # jeu Windows
s = "".join (f.readlines ())
f.close ()
print type (s)              # affiche <type 'unicode'>
print s.encode ("cp1252")   # pour l'afficher,
                            # il faut convertir l'unicode en "cp1252"
```

| | |
|---|---|
| `encode(` [`enc`[, `err`]]) | Cette fonction permet de passer d'un jeu de caractères, celui de la variable, au jeu de caractères précisé par `enc` à moins que ce ne soit le jeu de caractères par défaut. Le paramètre `err` permet de préciser comment gérer les erreurs, doit-on interrompre le programme (valeur `'strict'`) ou les ignorer (valeur `'ignore'`). La documentation *Python* recense toutes les valeurs possibles pour ces deux paramètres aux adresses http://docs.python.org/library/codecs.html#id3 et http://docs.python.org/library/stdtypes.html#id4. Cette fonction retourne un résultat de type `str`. |
| `decode(` [`enc`[, `err`]]) | Cette fonction est la fonction inverse de la fonction `encode`. Avec les mêmes paramètres, elle effectue la transformation inverse. |

**Table 7.5 :** *Conversion de jeux de caractères, ce sont deux méthodes qui fonctionnent de façons identiques pour les deux classes de chaînes de caractères disponibles en Python :* `str` *et* `unicode`.

Le programme suivant permet d'obtenir le jeu de caractères par défaut et celui du système d'exploitation.

```
import sys
import locale
            # retourne le jeu de caractères par défaut
print sys.getdefaultencoding ()    # affiche ascii
            # retourne le jeu de caractères du système d'exploitation
print locale.getdefaultlocale()    # affiche ('fr_FR', 'cp1252')
```

Les problèmes de jeux de caractères peuvent devenir vite compliqués lorsqu'on manipule des informations provenant de plusieurs langues différentes. Il est rare d'avoir à s'en soucier tant que le programme gère les langues anglaise et française. Dans le cas contraire, il est préférable d'utiliser le type `unicode` pour toutes les chaînes de caractères. Il est conseillé de n'utiliser qu'un seul jeu de caractères pour enregistrer les informations dans des fichiers et le jeu de caractères $\texttt{utf} - \texttt{8}$ est le plus indiqué.

# Chapitre 8

# Interface graphique

---

Les interfaces graphiques servent à rendre les programmes plus conviviaux. Elles sont pratiques à utiliser mais elles demandent un peu de temps pour les concevoir. Un programme sans interface exécute des instructions les unes à la suite des autres, le programme a un début - un point d'entrée - et une fin. Avec une interface, le programme fonctionne de manière différente. Il n'exécute plus successivement les instructions mais attend un événement - pression d'une touche du clavier, clic de souris - pour exécuter une fonction. C'est comme si le programme avait une multitude de points d'entrée.

Il existe plusieurs modules permettant d'exploiter les interfaces graphiques. Le plus simple est le module `Tkinter` présent lors de l'installation du langage *Python*. Ce module est simple mais limité. Le module `wxPython` est plus complet mais un peu plus compliqué dans son utilisation [1]. Toutefois, le fonctionnement des interfaces graphiques sous un module ou un autre est identique. C'est pourquoi ce chapitre n'en présentera qu'un seul, le module `Tkinter`. Pour d'autres modules, les noms de classes changent mais la logique reste la même : il s'agit d'associer des événements à des parties du programme *Python*.

Les interfaces graphiques évoluent sans doute plus vite que les autres modules, des composantes de plus en plus complexes apparaissent régulièrement. Un module comme `wxPython` change de version plusieurs fois par an. Il est possible de trouver sur Internet des liens [2] qui donnent des exemples de programme. Une excellente source de documentation sont les forums de discussion qui sont un lieu où des programmeurs échangent questions et réponses. Un message d'erreur entré sur un moteur de recherche Internet permet souvent de tomber sur des échanges de ce type, sur des problèmes résolus par d'autres.

## 8.1 Introduction

Un programme muni d'une interface graphique fonctionne différemment d'un programme classique. Un programme classique est une succession presque linéaire d'instructions. Il y a un début ou *point d'entrée* du programme et aucun événement extérieur ne vient troubler son déroulement. Avec une interface graphique, le point d'entrée du programme est masqué : il est pris en compte automatiquement. Du point de vue du programmeur, le programme a plusieurs points d'entrée : une simple

1. Le paragraphe 1.6.5 page 26 présente d'autres alternatives.
2. L'adresse http://gnuprog.info/prog/python/pwidget.php illustre chaque objet de `Tkinter`, on peut citer également http://effbot.org/tkinterbook/.

fenêtre avec deux boutons (voir figure 8.1) propose deux façons de commencer et il faut prévoir une action associée à chaque bouton.

La conception d'une interface graphique se déroule généralement selon deux étapes. La première consiste à dessiner l'interface, c'est-à-dire choisir une position pour les objets de la fenêtre (boutons, zone de saisie, liste déroulante, ...). La seconde étape définit le fonctionnement de la fenêtre, c'est-à-dire associer à chaque objet des fonctions qui seront exécutées si un tel événement se réalise (pression d'un bouton, pression d'une touche, ...).

Pour le moment, nous allons supposer que ces deux étapes sont scindées même si elles sont parfois entremêlées lorsqu'un événement implique la modification de l'apparence de la fenêtre. La section qui suit décrit des objets que propose le module `Tkinter`. La section suivante présente la manière de les disposer dans une fenêtre. La section d'après décrit les événements et le moyen de les relier à des fonctions du programme. Ce chapitre se termine par quelques constructions courantes à propos des interfaces graphiques.

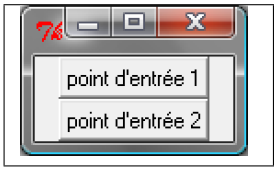



**Figure 8.1 :** *Une fenêtre contenant deux boutons : ce sont deux points d'entrée du programme.*

## 8.2 Les objets

Les interfaces graphiques sont composées d'*objets* ou *widgets*[3]. Ce paragraphe décrit les principales méthodes qui permettent de modifier le contenu et l'apparence des objets. Il faut se reporter à la documentation du langage[4] pour avoir la liste de toutes les options disponibles.

Les exemples de codes des paragraphes qui suivent permettent de disposer les objets dans une fenêtre qui ne s'affichera pas sans les quelques lignes de code supplémentaires présentées au paragraphe 8.4.1 et l'utilisation d'une méthode `pack` (paragraphe 8.3). L'exemple suivant crée un objet :

```
zone_texte = Tkinter.Label (text = "zone de texte")
```

Et pour l'afficher, il faut l'enrober :

```
import Tkinter          # import de Tkinter
root = Tkinter.Tk ()    # création de la fenêtre principale
# ...
obj = Tkinter.Label (text = "zone de texte")
# ...
obj.pack ()             # on ajoute l'objet à la fenêtre principale
root.mainloop ()        # on affiche enfin la fenêtre principal et on attend
                        # les événements (souris, clic, clavier)
```

3. On les appelle aussi *contrôle*. Comme ils reçoivent des événements, en un sens, ce sont ces objets qui pilotent un programme ou qui le contrôlent.

4. La page http://wiki.python.org/moin/Tkinter recense quelques liens utiles autour de `Tkinter` dont la documentation officielle.

### 8.2.1 Zone de texte

Une zone de texte sert à insérer dans une fenêtre graphique une légende indiquant ce qu'il faut insérer dans une zone de saisie voisine comme le montre la figure 8.2. Une zone de texte correspond à la classe `Label` du module `Tkinter`. Pour créer une zone de texte, il suffit d'écrire la ligne suivante :

```
zone_texte = Tkinter.Label (text = "zone de texte")
```

Il est possible que le texte de cette zone de texte doive changer après quelques temps. Dans ce cas, il faut appeler la méthode `config` comme suit :

```
zone_texte = Tkinter.Label (text = "premier texte")
# ...
# pour changer de texte
zone_texte.config (text = "second texte")
```


**Figure 8.2 :** *Exemple de zone de texte ou* Label *associée à une zone de saisie. La seconde image montre une zone de texte dans l'état* `DISABLED`.

La figure 8.2 montre deux zones de texte. La seconde est grisée par rapport à la première. Pour obtenir cet état, il suffit d'utiliser l'instruction suivante :

```
zone_texte.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
zone_texte.config (state = Tkinter.NORMAL)
```

Ces deux dernières options sont communes à tous les objets d'une interface graphique. Cette option sera rappelée au paragraphe 8.2.11.

### 8.2.2 Bouton

Un bouton a pour but de faire le lien entre une fonction et un clic de souris. Un bouton correspond à la classe `Button` du module `Tkinter`. Pour créer un bouton, il suffit d'écrire la ligne suivante :

```
bouton = Tkinter.Button (text = "zone de texte")
```

Il est possible que le texte de ce bouton doive changer après quelques temps. Dans ce cas, il faut appeler la méthode `config` comme suit :

```
bouton = Tkinter.Button (text = "premier texte")
# ...
# pour changer de texte
bouton.config (text = "second texte")
```

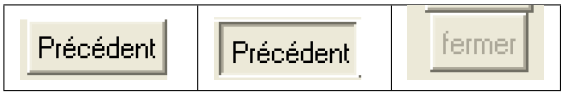



**Figure 8.3 :** *Exemple de boutons, non pressé, pressé, grisé. Le bouton grisé ne peut être pressé.*

La figure 8.3 montre trois boutons. Le troisième est grisé par rapport au premier. Les boutons grisés ne peuvent pas être pressés. Pour obtenir cet état, il suffit d'utiliser l'instruction suivante :

```
bouton.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
bouton.config (state = Tkinter.NORMAL)
```

C'est pour cet objet que cette option est la plus intéressante car elle permet d'interdire la possibilité pour l'utilisateur de presser le bouton tout en le laissant visible.

Il est possible également d'associer une image à un bouton. Par exemple, les trois lignes suivantes créent un bouton, charge une image au format `gif` puis l'associe au bouton `b`. Lors de l'affichage de la fenêtre, le bouton `b` ne contient pas de texte mais une image.

```
b = Tkinter Button ()
im = Tkinter.PhotoImage (file = "chameau.gif")
b.config (image = im)
```

Les images qu'il est possible de charger sont nécessairement au format `GIF`, le seul que le module `Tkinter` puisse lire.

### 8.2.3 Zone de saisie

Une zone de saisie a pour but de recevoir une information entrée par l'utilisateur. Une zone de saisie correspond à la classe `Entry` du module `Tkinter` ; pour en créer une, il suffit d'écrire la ligne suivante :

```
saisie = Tkinter.Entry ()
```

Pour modifier le contenu de la zone de saisie, il faut utiliser la méthode `insert` qui insère un texte à une position donnée.

```
# le premier paramètre est la position
# où insérer le texte (second paramètre)
saisie.insert (pos, "contenu")
```

Pour obtenir le contenu de la zone de saisie, il faut utiliser la méthode `get` :

```
contenu = saisie.get ()
```

Pour supprimer le contenu de la zone de saisie, il faut utiliser la méthode `delete`. Cette méthode supprime le texte entre deux positions.

```
# supprime le texte entre les positions pos1, pos2
saisie.delete (pos1, pos2)
```

Par exemple, pour supprimer le contenu d'une zone de saisie, on peut utiliser l'instruction suivante :

```
saisie.delete (0, len (saisie.get ()))
```


**Figure 8.4 :** *Exemple de zones de saisie, normale et grisée, la zone grisée ne peut être modifiée.*

La figure 8.4 montre deux zones de saisie. La seconde est grisée par rapport à la première. Les zones de saisie grisées ne peuvent pas être modifiées. Pour obtenir cet état, il suffit d'utiliser la méthode `config` comme pour les précédents objets. Cette option sera rappelée au paragraphe 8.2.11.

### 8.2.4 Zone de saisie à plusieurs lignes

Une zone de saisie à plusieurs lignes est identique à la précédente à ceci près qu'elle autorise la saisie d'un texte sur plusieurs lignes. Cette zone correspond à la classe `Text` du module `Tkinter`. Pour créer une telle zone, il suffit d'écrire la ligne suivante :

```
saisie = Tkinter.Text ()
```

Pour modifier le contenu de la zone de saisie, il faut utiliser la méthode `insert` qui insère un texte à une position donnée. La méthode diffère de celle de la classe `Entry` puisque la position d'insertion est maintenant une chaîne de caractères contenant deux nombres séparés par un point : le premier nombre désigne la ligne, le second la position sur cette ligne.

```
# le premier paramètre est la position
# où insérer le texte (second paramètre)
pos = "0.0"
saisie.insert (pos, "première ligne\nseconde ligne")
```

Pour obtenir le contenu de la zone de saisie, il faut utiliser la méthode `get` qui retourne le texte entre deux positions. La position de fin n'est pas connue, on utilise la chaîne de caractères `"end"` pour désigner la fin de la zone de saisie.

```
# retourne le texte entre deux positions
pos1 = "0.0"
pos2 = "end"  # ou Tkinter.END
contenu = saisie.get (pos1, pos2)
```

Pour supprimer le contenu de la zone de saisie, il faut utiliser la méthode `delete`. Cette méthode supprime le texte entre deux positions.

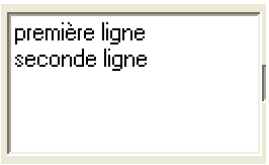
```
# supprime le texte entre les positions pos1, pos2
saisie.delete (pos1, pos2)
```

Par exemple, pour supprimer le contenu d'une zone de saisie à plusieurs lignes, on peut utiliser l'instruction suivante :

```
saisie.delete ("0.0", "end")
# on peut aussi utiliser
# saisie.delete ("0.0", Tkinter.END)
```

Pour modifier les dimensions de la zone de saisie à plusieurs lignes, on utilise l'instruction suivante :

```
# modifie les dimensions de la zone
# width <--> largeur
# height <--> hauteur en lignes
saisie.config (width = 10, height = 5)
```




L'image précédente montre une zone de saisie à plusieurs lignes. Pour griser cette zone, il suffit d'utiliser la méthode `config` rappelée au paragraphe 8.2.11.

### 8.2.5 Case à cocher

Une case à cocher correspond à la classe `Checkbutton` du module `Tkinter`. Pour créer une case à cocher, il suffit d'écrire la ligne suivante :

```
# crée un objet entier pour récupérer la valeur de la case à cocher,
# 0 pour non cochée, 1 pour cochée
v    = Tkinter.IntVar ()
case = Tkinter.Checkbutton (variable = v)
```

En fait, ce sont deux objets qui sont créés. Le premier, de type `IntVar`, mémorise la valeur de la case à cocher. Le second objet, de type `CheckButton`, gère l'apparence au niveau de l'interface graphique. La raison de ces deux objets est plus évidente dans le cas de l'objet `RadioButton` décrit au paragraphe suivant. Pour savoir si la case est cochée ou non, il suffit d'exécuter l'instruction :

```
v.get ()  # égal à 1 si la case est cochée, 0 sinon
```

Pour cocher et décocher la case, il faut utiliser les instructions suivantes :

```
case.select ()      # pour cocher
case.deselect ()    # pour décocher
```

Il est possible d'associer du texte à l'objet case à cocher :

```
case.config (text = "case à cocher")
```

La figure 8.5 montre trois cases. La troisième est grisée par rapport à la première. Les cases grisées ne peuvent pas être cochées. Pour obtenir cet état, il suffit d'utiliser la méthode `config` rappelée au paragraphe 8.2.11.


**Figure 8.5 :** *Exemples de cases à cocher, non cochée, cochée, grisée. Lorsqu'elle est grisée, son état ne peut être modifié. La dernière image montre une case à cocher associée à un texte.*

### 8.2.6 Case ronde ou bouton radio

Une case ronde ou *bouton radio* correspond à la classe `Radiobutton` du module `Tkinter`. Elles fonctionnent de manière semblable à des cases à cocher excepté le fait qu'elles n'apparaissent jamais seules : elles fonctionnent en groupe. Pour créer un groupe de trois cases rondes, il suffit d'écrire la ligne suivante :

```
# crée un objet entier partagé pour récupérer le numéro du bouton radio activé
v     = Tkinter.IntVar ()
case1 = Tkinter.Radiobutton (variable = v, value = 10)
case2 = Tkinter.Radiobutton (variable = v, value = 20)
case3 = Tkinter.Radiobutton (variable = v, value = 30)
```

La variable `v` est partagée par les trois cases rondes. L'option `value` du constructeur permet d'associer un bouton radio à une valeur de `v`. Si `v` == `10`, seul le premier bouton radio sera sélectionné. Si `v` == `20`, seul le second bouton radio le sera. Si deux valeurs sont identiques pour deux boutons radio, ils seront cochés et décochés en même temps. Et pour savoir quel bouton radio est coché ou non, il suffit d'exécuter l'instruction :

```
v.get ()  #  retourne le numéro du bouton radio coché (ici, 10, 20 ou 30)
```

Pour cocher un des boutons radio, il faut utiliser l'instruction suivante :

```
v.set (numero)  # numéro du bouton radio à cocher
                # pour cet exemple, 10, 20 ou 30
```

Il est possible d'associer du texte à un bouton radio (voir figure 8.6) :

```
case1.config (text = "premier bouton")
case2.config (text = "second bouton")
case3.config (text = "troisième bouton")
```

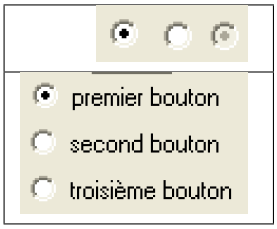



**Figure 8.6 :** *Exemples de cases rondes ou boutons radio, non cochée, cochée, grisée. Lorsqu'elle est grisée, son état ne peut être modifiée. La seconde image présente un groupe de bouton radio. Un seul peut être sélectionné à la fois à moins que deux boutons ne soient associés à la même valeur. Dans ce cas, ils agiront de pair.*

La figure 8.6 montre trois cases. La troisième est grisée par rapport à la première. Si un des boutons radio est grisé parmi les trois, le choix du bouton à cocher s'effectue parmi les deux restants ou aucun si le bouton radio grisé est coché.

### 8.2.7 Liste

Un objet liste contient une liste d'intitulés qu'il est possible de sélectionner. Une liste correspond à la classe `ListBox` du module `Tkinter`. Pour la créer, il suffit d'écrire la ligne suivante :

```
li  = Tkinter.Listbox ()
```

Pour modifier les dimensions de la zone de saisie à plusieurs lignes, on utilise l'instruction suivante :

```
# modifie les dimensions de la liste
# width <--> largeur
# height <--> hauteur en lignes
li.config (width = 10, height = 5)
```

On peut insérer un élément dans la liste avec la méthode `insert` :

```
pos = 0   # un entier, "end" ou Tkinter.END pour insérer ce mot à la fin
li.insert (pos, "première ligne")
```

On peut supprimer des intitulés de cette liste avec la méthode `delete`.

```
pos1 = 0    # un entier
pos2 = None # un entier, "end" ou Tkinter.END pour supprimer tous les éléments
            # de pos1 jusqu'au dernier
li.delete (pos1, pos2 = None)
```

Les intitulés de cette liste peuvent ou non être sélectionnés. Cliquer sur un intitulé le sélectionne mais la méthode `select_set` permet aussi de le faire.

```
pos1 = 0
li.select_set (pos1, pos2 = None)
# sélectionne tous les éléments entre les indices pos1 et
# pos2 inclus ou seulement celui d'indice pos1 si pos2 == None
```

Réciproquement, il est possible d'enlever un intitulé de la sélection à l'aide de la méthode `select_clear`.

```
pos1 = 0
li.select_clear (pos1, pos2 = None)
# retire la sélection de tous les éléments entre les indices
# pos1 et pos2 inclus ou seulement celui d'indice pos1 si pos2 == None
```

La méthode `curselection` permet d'obtenir la liste des indices des éléments sélectionnés.

```
sel = li.curselection ()
```

La méthode `get` permet récupérer un élément de la liste tandis que la méthode `size` retourne le nombre d'éléments :

```
for i in range (0,li.size ()) :
    print li.get (i)
```

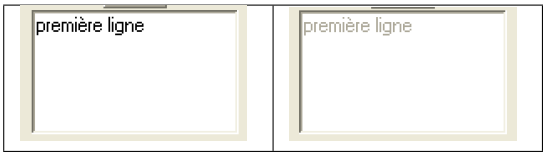



**Figure 8.7 :** *Exemple de liste. La seconde liste est grisée et ne peut être modifiée.*

La figure 8.7 montre deux listes. La seconde est grisée par rapport à la première. Elle ne peut être modifiée. Pour obtenir cet état, il faut appeler la méthode `config` rappelée au paragraphe 8.2.11.

**Remarque 8.1 : liste et barre de défilement**
Il est possible d'adjoindre une barre de défilement verticale. Il faut pour cela inclure l'objet dans une sous-fenêtre `Frame` qui est définie au paragraphe 8.3.2 comme dans l'exemple suivant :

```
frame      = Tkinter.Frame (parent)
scrollbar  = Tkinter.Scrollbar (frame)
li         = Tkinter.Listbox (frame, width = 88, height = 6, \
                              yscrollcommand = scrollbar.set)
scrollbar.config (command = li.yview)
li.pack (side = Tkinter.LEFT)
scrollbar.pack (side = Tkinter.RIGHT, fill = Tkinter.Y)
```

Il suffit de transposer cet exemple pour ajouter une barre de défilement horizontale. Toutefois, il est préférable d'utiliser un objet prédéfini présent dans le module `Tix` qui est une extension du module `Tkinter`. Elle est présentée au paragraphe 8.2.8.

**Remarque 8.2 : plusieurs Listbox**
Lorsqu'on insère plusieurs objets `Listbox` dans une seule fenêtre, ces objets partagent par défaut la même sélection. Autrement dit, lorsqu'on clique sur un élément de la seconde `Listbox`, l'élément sélectionné dans la première ne l'est plus. Afin de pouvoir sélectionner un élément dans chaque `Listbox`, il faut ajouter dans les paramètres du constructeur l'option `exportselection` = `0` comme l'illustre l'exemple suivant :

```
li = Tkinter.Listbox (frame, width = 88, height = 6, exportselection=0)
```

Il existe des méthodes plus avancées qui permettent de modifier l'aspect graphique d'un élément comme la méthode `itemconfigure`. Son utilisation est peu fréquente à moins de vouloir réaliser une belle interface graphique. Le paragraphe 8.6.2 montre l'utilisation qu'on peut en faire.

### 8.2.8 Liste avec barre de défilement, Combobox

Il n'existe pas dans le module `Tkinter` d'objets `List` avec une barre de défilement incluse mais il existe un autre module interne `Tix` qui étend la liste des objets proposés par `Tkinter`. Ce module propose notamment une liste avec une barre de défilement intégrée :

```
# coding: latin-1
import Tix as Tk
root = Tk.Tk ()
```

```
o = Tk.ScrolledListBox (root)
for k in range (0,100) : o.listbox.insert (Tk.END, "ligne " + str (k))
o.pack ()

def print_file () :                     # voir chapitre sur les événements
    print o.listbox.selection_get ()    # idem

b = Tk.Button (root, text = "print")
b.config (command = print_file)         # idem
b.pack ()

root.mainloop ()                        # idem
```

Le module `Tix` quoique assez riche puisqu'il propose des fenêtres permettant de sélectionner un fichier ou de remplir un tableau est mal documenté sur *Python*. Il existe une documentation officielle [5] et des exemples [6]. La plupart du temps, on trouve un exemple approché. Dans l'exemple précédent comme dans le suivant avec une `"ComboBox"`, la pression du bouton `print` écrit la zone sélectionnée. La figure 8.8 illustre graphiquement ce qu'est une `Combobox`.

```
# coding: latin-1
import Tix as Tk
root = Tk.Tk ()

o = Tk.ComboBox (root, label = "label")
o.insert (Tk.END, "ligne 1")
o.insert (Tk.END, "ligne 2")
o.insert (Tk.END, "ligne 3")
o.insert (Tk.END, "ligne 4")
o.pack ()

def print_file () :                     # voir le chapitre sur les événéments
    print o.cget ("value")              # idem

b = Tk.Button (root, text = "print")
b.config (command = print_file)         # idem
b.pack ()

root.mainloop ()                        # idem
```

L'avantage du module `Tix` est d'être installé automatiquement avec le langage *Python*. Il reste malgré tout très peu documenté et arriver à ses fins peut nécessiter quelques heures de recherche.

### 8.2.9 Canevas

Pour dessiner, il faut utiliser un objet canevas, correspondant à la classe `Canvas` du module `Tkinter`. Pour la créer, il suffit d'écrire la ligne suivante :

```
    ca  = Tkinter.Canvas ()
```

5. http://pydoc.org/2.1/Tix.html ou encore http://docs.python.org/library/tix.html
6. http://coverage.livinglogic.de/index.html

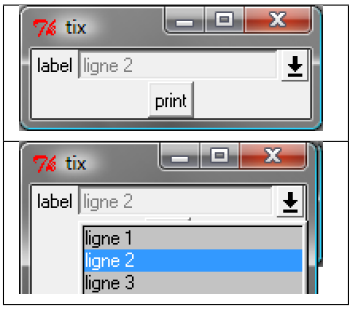



**Figure 8.8 :** *Exemple de fenêtre* `Combobox` *définie par le module interne* `Tix` *qui est une extension du module* `Tkinter`. *La seconde image est la même* `Combobox` *mais dépliée.*

Pour modifier les dimensions de la zone de saisie à plusieurs lignes, on utilise l'instruction suivante :

```
# modifie les dimensions du canevas
# width <--> largeur en pixels
# height <--> hauteur en pixels
ca.config (width = 10, height = 5)
```

Cet objet permet de dessiner des lignes, des courbes, d'écrire du texte grâce aux méthodes `create_line`, `create_rectangle`, `create_text`. La figure 8.9 illustre ce que donnent les quelques lignes qui suivent.

```
  # dessine deux lignes du point 10,10 au point 40,100 et au point 200,60
  # de couleur bleue, d'épaisseur 2
ca.create_line (10,10,40,100, 200,60, fill = "blue", width = 2)
  # dessine une courbe du point 10,10 au point 200,60
  # de couleur rouge, d'épaisseur 2, c'est une courbe de Bézier
  # pour laquelle le point  40,100 sert d'assise
ca.create_line (10,10, 40,100, 200,60, smooth=1, fill = "red", width = 2)
  # dessine un rectangle plein de couleur jaune, de bord noir et d'épaisseur 2
ca.create_rectangle (300,100,60,120, fill = "gray", width = 2)
  # écrit du texte de couleur noire au point 80,80 et avec la police arial
ca.create_text (80,80, text = "écrire", fill = "black", font = "arial")
```

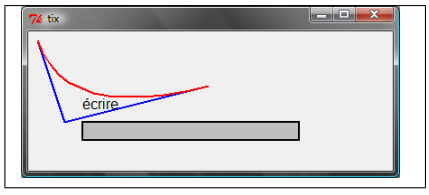



**Figure 8.9 :** *Exemple de canevas.*

### 8.2.10 Menus

Les menus apparaissent en haut des fenêtres. La plupart des applications arborent un menu commençant par *Fichier Edition Affichage...* Le paragraphe 8.4.5 page 214 les décrit en détail.

### 8.2.11 Méthodes communes

Nous avons vu que tous les objets présentés dans ce paragraphe possèdent une méthode `config` qui permet de définir l'état du widget (grisé ou normal) [7].

```
widget.config (state = Tkinter.DISABLED) # grisé
widget.config (state = Tkinter.NORMAL)   # aspect normal
```

Elle permet également de modifier le texte d'un objet, sa position, ... De nombreuses options sont communes à tous les objets et certaines sont spécifiques. L'aide associée à cette méthode [8] ne fournit aucun renseignement. En fait, le constructeur et cette méthode ont les mêmes paramètres optionnels. Il est équivalent de préciser ces options lors de l'appel au constructeur :

```
l = Tkinter.Label (text = "légende")
```

Ou de les modifier à l'aide de la méthode `config` :

```
l = Tkinter.Label ()
l.config (text = "légende")
```

L'aide associée à la méthode `config` est la suivante :

```
Help on method configure in module Tkinter:

configure(self, cnf=None, **kw) unbound Tkinter.Label method
    Configure resources of a widget.

    The values for resources are specified as keyword
    arguments. To get an overview about
    the allowed keyword arguments call the method keys.
```

Tandis que l'aide associée au constructeur de la classe `Label` [9] donne plus d'informations :

```
__init__(self, master=None, cnf={}, **kw) unbound Tkinter.Label method
    Construct a label widget with the parent MASTER.

    STANDARD OPTIONS

        activebackground, activeforeground, anchor,
        background, bitmap, borderwidth, cursor,
        disabledforeground, font, foreground,
        highlightbackground, highlightcolor,
        highlightthickness, image, justify,
        padx, pady, relief, takefocus, text,
        textvariable, underline, wraplength

    WIDGET-SPECIFIC OPTIONS

        height, state, width
```

7. La "disparition" d'un objet est évoquée au paragraphe 8.3.

8. Par exemple pour la classe `Label`, cette aide est affichée grâce à l'instruction `help(Tkinter.Label.config)`.

9. Affichée avec l'instruction `help(Tkinter.Label.__init__)`.

Cette aide mentionne les options communes à tous les objets (ou widgets) et les options spécifiques à cet objet, ici de type `Label`. Toutes ont une valeur par défaut qu'il est possible de changer soit dans le constructeur, soit par la méthode `config`. Quelques-unes ont été décrites, d'autres permettent de modifier entre autres la police avec laquelle est affiché le texte de l'objet (option `font`), la couleur du fond (option `background`), l'alignement du texte, à gauche, à droite, centré (option `justify`), l'épaisseur du bord (option `borderwidth`), le fait qu'un objet reçoive le *focus*[10] après la pression de la touche tabulation (`takefocus`)...

## 8.3 Disposition des objets dans une fenêtre

### 8.3.1 Emplacements

Chacun des objets (ou widgets) présentés au paragraphe précédent possède trois méthodes qui permettent de déterminer sa position dans une fenêtre : `pack`, `grid`, `place`. Les deux premières permettent de disposer les objets sans se soucier ni de leur dimension ni de leur position. La fenêtre gère cela automatiquement. La dernière place les objets dans une fenêtre à l'aide de coordonnées sans utiliser l'aide d'aucune grille. Dans une fenêtre, tous les objets doivent être placés avec la même méthode. Dans le cas contraire, les résultats risquent ne pas être ceux attendus.

#### 8.3.1.1 Méthode `pack`

Cette méthode empile les objets les uns à la suite des autres. Par défaut, elle les empile les uns en dessous des autres. Par exemple, l'exemple suivant produit l'empilement des objets de la figure 8.10.

```
l = Tkinter.Label (text = "première ligne")
l.pack ()
s = Tkinter.Entry ()
s.pack ()
e = Tkinter.Label (text = "seconde ligne")
e.pack ()
```

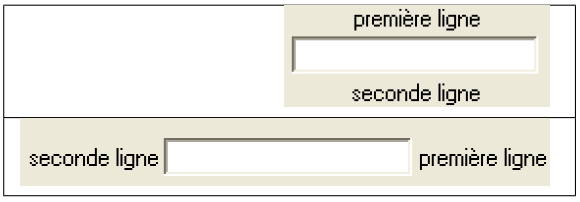



**Figure 8.10 :** *Les objets sont empilés à l'aide de la méthode* `pack` *les uns en dessous des autres pour la première image, les uns à droite des autres pour la seconde image.*

On peut aussi les empiler les uns à droite des autres grâce à l'option `side`.

```
l = Tkinter.Label (text = "première ligne")
l.pack (side = Tkinter.RIGHT)
s = Tkinter.Entry ()
```

10. voir paragraphe 8.4.2

```
s.pack (side = Tkinter.RIGHT)
e = Tkinter.Label (text = "seconde ligne")
e.pack (side = Tkinter.RIGHT)
```

La méthode `pack` possède trois options :

1. `side` : à choisir entre `Tkinter.TOP` (valeur par défaut), `Tkinter.LEFT`, `Tkinter.BOTTOM`, `Tkinter.RIGHT`.
2. `expand` : égale à `True` ou `False` (valeur par défaut), si cette option est vraie, l'objet occupe tout l'espace.
3. `fill` : égale à `None` (valeur par défaut), `X`, `Y`, `BOTH`, l'objet s'étend selon un axe (X ou Y ou les deux).

Il n'est pas toujours évident d'obtenir du premier coup le positionnement des objets souhaités au départ et il faut tâtonner pour y arriver. Lorsque un objet n'est plus nécessaire, il est possible de le faire disparaître en appelant la méthode `pack_forget`. Le rappel de la méthode `pack` le fera réapparaître mais rarement au même endroit.

```
s.pack_forget ()  # disparition
s.pack ()         # insertion à un autre endroit
```

#### 8.3.1.2 Méthode `grid`

La méthode `grid` suppose que la fenêtre qui les contient est organisée selon une grille dont chaque case peut recevoir un objet. L'exemple suivant place trois objets dans les cases de coordonnées $(0, 0)$, $(1, 0)$ et $(0, 1)$. Le résultat apparaît dans la figure 8.11.

```
l = Tkinter.Label (text = "première ligne")
l.grid (column = 0, row = 0)
s = Tkinter.Entry ()
s.grid (column = 0, row = 1)
e = Tkinter.Label (text = "seconde ligne")
e.grid (column = 1, row = 0)
```


**Figure 8.11 :** *Les objets sont placés dans une grille à l'aide de la méthode* `grid`*. Une fois que chaque objet a reçu une position, à l'affichage, il ne sera pas tenu compte des lignes et colonnes vides.*

La méthode `grid` possède plusieurs options, en voici cinq :

1. `column` : colonne dans laquelle sera placée l'objet.
2. `columnspan` : nombre de colonnes que doit occuper l'objet.
3. `row` : ligne dans laquelle sera placée l'objet.
4. `rowspan` : nombre de lignes que doit occuper l'objet.
5. `sticky` : indique ce qu'il faut faire lorsque la case est plus grande que l'objet qu'elle doit contenir. Par défaut, l'objet est centré mais il est possible d'aligner l'objet sur un ou plusieurs bords en précisant que `sticky` = "`N`" ou "`S`" ou "`W`" ou "`E`". Pour

aligner l'objet sur un angle, il suffit de concaténer les deux lettres correspondant aux deux bords concernés. Il est aussi possible d'étendre l'objet d'un bord à l'autre en écrivant `sticky` = "`N` + `S`" ou `sticky` = "`E` + `W`".

Enfin, comme pour la méthode `pack`, il existe une méthode `grid_forget` qui permet de faire disparaître les objets.

```
s.grid_forget ()  # disparition
```

#### 8.3.1.3 Méthode `place`

La méthode `place` est sans doute la plus simple à comprendre puisqu'elle permet de placer chaque objet à une position définie par des coordonnées. Elle peut être utilisée en parallèle avec les méthodes `pack` et `grid`.

```
l = Tkinter.Label (text = "première ligne")
l.place (x=10,y=50)
```

La méthode `place_forget` permet de faire disparaître un objet placer avec cette méthode. L'inconvénient de cette méthode survient lorsqu'on cherche à modifier l'emplacement d'un objet : il faut en général revoir les positions de tous les autres éléments de la fenêtre. On procède souvent par tâtonnement pour construire une fenêtre et disposer les objets. Ce travail est beaucoup plus long avec la méthode `place`.

### 8.3.2 Sous-fenêtre

Les trois méthodes précédentes ne permettent pas toujours de placer les éléments comme on le désire. On souhaite parfois regrouper les objets dans des boîtes et placer celles-ci les unes par rapport aux autres. La figure 8.12 montre deux objets regroupés dans un rectangle avec à sa gauche une zone de texte. Les boîtes sont des instances de la classe `Frame`. Ce sont des objets comme les autres excepté le fait qu'une boîte contient d'autres objets y compris de type `Frame`. Pour créer une boîte, il suffit d'écrire la ligne suivante :

```
f = Tkinter.Frame ()
```

Ensuite, il faut pouvoir affecter un objet à cette boîte `f`. Pour cela, il suffit que `f` soit le premier paramètre du constructeur de l'objet créé :

```
l = Tkinter.Label (f, text = "première ligne")
```

L'exemple qui suit correspond au code qui permet d'afficher la fenêtre de la figure 8.12 :

```
f = Tkinter.Frame ()
l = Tkinter.Label (f, text = "première ligne")
l.pack ()                    # positionne l à l'intérieur de f
s = Tkinter.Entry (f)
```

```
    s.pack ()                     # positionne s à l'intérieur de f
    f.pack (side = Tkinter.LEFT)  # positionne f à l'intérieur
                                  #   de la fenêtre principale
    e = Tkinter.Label (text = "seconde ligne")
    e.pack_forget ()
    e.pack (side = Tkinter.RIGHT) # positionne e à l'intérieur
                                  #   de la fenêtre principale
```

L'utilisation de ces blocs `Frame` est pratique lorsque le même ensemble de contrôles apparaît dans plusieurs fenêtres différentes ou au sein de la même fenêtre. Cette possibilité est envisagée au paragraphe 8.6.3.

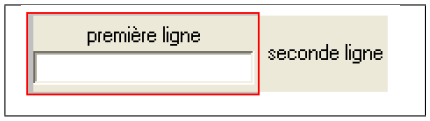



**Figure 8.12 :** *Les deux premiers objets - une zone de texte au-dessus d'une zone de saisie - sont regroupés dans une boîte - rectangle rouge, invisible à l'écran. A droite et centrée, une dernière zone de texte. Cet alignement est plus simple à réaliser en regroupant les deux premiers objets dans un rectangle (objet* `Frame`*).*

## 8.4 Evénements

### 8.4.1 Fenêtre principale

Tous les exemples des paragraphes précédents décrivent les différents objets disponibles et comment les disposer dans une fenêtre. Pour afficher cette fenêtre, il suffit d'ajouter au programme les deux lignes suivantes :

```
root = Tkinter.Tk ()
#  ici, on trouve le code qui définit les objets
#  et leur positionnement
root.mainloop ()
```

La première ligne permet d'obtenir un identificateur relié à la fenêtre principale. La seconde ligne, outre le fait qu'elle affiche cette fenêtre, lance ce qu'on appelle une *boucle de messages*. Cette fonction récupère - intercepte - les événements comme un clic de souris, la pression d'une touche. Elle parcourt ensuite tous les objets qu'elle contient et regarde si l'un de ces objets est intéressé par cet événement. S'il est intéressé, cet objet prend l'événement et le traite. On peut revenir ensuite à la fonction `mainloop` qui attend à nouveau un événement. Cette fonction est définie par `Tkinter`, il reste à lui indiquer quels événements un objet désire intercepter et ce qu'il est prévu de faire au cas où cet événement se produit.

**Remarque 8.3 : instruction** `root` = `Tkinter.Tk`()
Cette première instruction doit se trouver au début du programme. Si elle intervient alors qu'une méthode de positionnement (`pack`, `grid`, ou `place`) a déjà été appelée, le programme affichera deux fenêtres dont une vide.

### 8.4.2 Focus

Une fenêtre peut contenir plusieurs zones de saisie, toutes capables d'intercepter la pression d'une touche du clavier et d'ajouter la lettre correspondante à la zone de saisie. Or la seule qui ajoute effectivement une lettre à son contenu est celle qui a le *focus*. La pression de la touche tabulation fait passer le focus d'un objet à l'autre. La figure 8.13 montre un bouton qui a le focus. Lorsqu'on désire qu'un objet en particulier ait le focus, il suffit d'appeler la méthode `focus_set`.

```
e = Tkinter.Entry ()
e.pack ()
e.focus_set ()
```

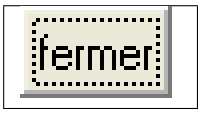



**Figure 8.13 :** *Ce bouton est entouré d'un cercle noir en pointillé, il a le* focus.

### 8.4.3 Lancer une fonction lorsqu'un bouton est pressé

La plupart de temps, le seul événement qu'on souhaite attraper est la pression d'un bouton. Le code suivant permet de créer un bouton dont l'identificateur est `b`. Il a pour intitulé `fonction change_legende`. On définit ensuite une fonction `change_legende` qui change la légende de ce bouton. L'avant-dernière ligne permet d'associer au bouton `b` la fonction `change_legende` qui est alors appelée lorsque le bouton est pressé. La dernière ligne affiche la fenêtre principale et lance l'application.

```
# coding: latin-1
# la première ligne autorise les accents
import Tkinter
root = Tkinter.Tk ()
b = Tkinter.Button (text = "fonction change_legende")
b.pack ()

def change_legende () :
    global b
    b.config (text = "nouvelle légende")

b.config (command = change_legende)
root.mainloop ()
```

Lorsque le bouton `b` est pressé, on vérifie qu'il change bien de légende (voir figure 8.14).

**Remarque 8.4 : événement associé à une méthode**
L'exemple précédent associe une fonction au bouton. Lorsque l'interface devient conséquente, la lisibilité du programme en est réduite car le nombre de fonctions associées à des boutons augmentent rapidement. Pour éviter cela, il est possible d'associer au bouton une méthode de classe comme le suggère l'exemple du paragraphe 8.6.3.


**Figure 8.14 :** *La première fenêtre est celle qui apparaît lorsque le programme de la page 210 est lancé. Comme le bouton change de légende la première fois qu'il est pressé, l'apparence de la fenêtre change aussi, ce que montre la seconde image.*

### 8.4.4 Associer n'importe quel événement à un objet

Le paragraphe précédent s'est intéressé à l'association entre une fonction et la pression d'un bouton mais il est possible de faire en sorte que le programme exécute une fonction au moindre déplacement de la souris, à la pression d'une touche. Il est possible d'associer une fonction au moindre événement susceptible d'être intercepté par l'interface graphique.

On peut regrouper les événements en deux classes. La première classe regroupe les événements provenant du clavier ou de la souris. Ce sont des événements en quelque sorte *bruts*. La seconde classe regroupe des événements produit par des objets tels qu'un bouton. En effet, lorsque celui-ci détecte le clic du bouton droit de la souris, il construit un événement "`pressiondubouton`" et c'est celui-ci qui va déclencher l'exécution d'une fonction. Il n'est pas souvent nécessaire de revenir aux événements *bruts* car les objets proposent d'eux-mêmes de pouvoir attacher des fonctions à des événements liés à leur apparence.

Toutefois, pour un jeu par exemple, il est parfois nécessaire d'avoir accès au mouvement de la souris et il faut revenir aux événements *bruts*. Un événement est décrit par la classe `Event` dont les attributs listés par la table 8.1 décrivent l'événement qui sera la plupart du temps la pression d'une touche du clavier ou le mouvement de la souris.

| | |
|---|---|
| `char` | Lorsqu'une touche a été pressée, cet attribut contient son code, il ne tient pas compte des touches dites muettes comme les touches `shift`, `ctrl`, `alt`. Il tient pas compte non plus des touches `return` ou `suppr`. |
| `keysym` | Lorsqu'une touche a été pressée, cet attribut contient son code, quelque soit la touche, muette ou non. |
| `num` | Contient un identificateur de l'objet ayant reçu l'événement. |
| `x, y` | Coordonnées relatives de la souris par rapport au coin supérieur gauche de l'objet ayant reçu l'événement. |
| `x_root, y_root` | Coordonnées absolues de la souris par rapport au coin supérieur gauche de l'écran. |
| `widget` | Identifiant permettant d'accéder à l'objet ayant reçu l'événement. |

**Table 8.1 :** *Attributs principaux de la classe* `Event`*, ils décrivent les événements liés au clavier et à la souris. La liste complète est accessible en utilisant l'instruction* `help(Tkinter.Event)`.

La méthode `bind` permet d'exécuter une fonction lorsqu'un certain événement donné est intercepté par un objet donné. La fonction exécutée accepte un seul paramètre de type `Event` qui est l'événement qui l'a déclenchée. Cette méthode a pour syntaxe :

**syntaxe 8.5 : méthode `bind`**

```
w.bind (ev, fonction)
```

`w` est l'identificateur de l'objet devant intercepter l'événement désigné par la chaîne de caractères `ev` (voir table 8.2). `fonction` est la fonction qui est appelée lorsque l'événement survient. Cette fonction ne prend qu'un paramètre de type `Event`.

| | |
|---|---|
| `<Key>` | Intercepter la pression de n'importe quelle touche du clavier. |
| `<Button-i>` | Intercepter la pression d'un bouton de la souris. `i` doit être remplacé par 1,2,3. |
| `<ButtonRelease-i>` | Intercepter le relâchement d'un bouton de la souris. `i` doit être remplacé par 1,2,3. |
| `<Double-Button-i>` | Intercepter la double pression d'un bouton de la souris. `i` doit être remplacé par 1,2,3. |
| `<Motion>` | Intercepter le mouvement de la souris, dès que le curseur bouge, la fonction liée à l'événement est appelée. |
| `<Enter>` | Intercepter un événement correspondant au fait que le curseur de la souris entre la zone graphique de l'objet. |
| `<Leave>` | Intercepter un événement correspondant au fait que le curseur de la souris sorte la zone graphique de l'objet. |

**Table 8.2 :** *Chaînes de caractères correspondant aux principaux types d'événements qu'il est possible d'intercepter avec le module* `Tkinter`*. La liste complète est accessible en utilisant l'instruction* `help(Tkinter.Label.bind)`.

L'exemple suivant utilise la méthode `bind` pour que le seul bouton de la fenêtre intercepte toute pression d'une touche, tout mouvement et toute pression du premier bouton de la souris lorsque le curseur est au dessus de la zone graphique du bouton. La fenêtre crée par ce programme ainsi que l'affichage qui en résulte apparaissent dans la figure 8.15.

```
import Tkinter
root = Tkinter.Tk ()
b = Tkinter.Button (text = "appuyer sur une touche")
b.pack ()

def affiche_touche_pressee (evt) :
    print "--------------------------- touche pressee"
    print "evt.char = ", evt.char
    print "evt.keysym = ", evt.keysym
    print "evt.num = ", evt.num
    print "evt.x,evt.y = ", evt.x, ",", evt.y
    print "evt.x_root,evt.y_root = ", evt.x_root, ",", evt.y_root
    print "evt.widget = ", evt.widget
```

```
b.bind ("<Key>", affiche_touche_pressee)
b.bind ("<Button-1>", affiche_touche_pressee)
b.bind ("<Motion>", affiche_touche_pressee)
b.focus_set ()

root.mainloop ()
```


```
evt.char =  ??
evt.keysym =  ??
evt.num =  1
evt.x,evt.y =  105 , 13
evt.x_root,evt.y_root =
                292 , 239
evt.widget =  .9261224
```

```
evt.char =
evt.keysym =  Return
evt.num =  ??
evt.x,evt.y =  105 , 13
evt.x_root,evt.y_root =
                292 , 239
evt.widget =  .9261224
```

**Figure 8.15 :** *Fenêtre affichée par le programme du paragraphe 8.4.4. La pression d'une touche déclenche l'affichage des caractéristiques de l'événement. La seconde colonne correspond à la pression du premier bouton de la souris. La dernière colonne correspond à la pression de la touche* `Return`.

### Remarque 8.6 : focus

L'avant dernière ligne du programme fait intervenir la méthode `focus_set`. Elle stipule que le bouton doit recevoir le `focus`. C'est-à-dire que cet objet est celui qui peut intercepter les événements liés au clavier. Sans cette instruction, cet objet n'y a pas accès, ces événements sont dirigés vers la fenêtre principale qui ne s'en soucie pas.

### Remarque 8.7 : mauvais événement

Les messages d'erreur liés aux événements ne sont pas forcément très explicites. Ainsi l'instruction suivante adresse un événement inexistant.

```
b.bind ("<button-1>", affiche_touche_pressee)
```

Lors de l'exécution, le programme déclenche la succession d'exceptions suivantes qui signifie que l'événement `<button-1>` n'existe pas.

```
Traceback (most recent call last):
  File "exemple_bind.py", line 17, in ?
    b.bind ("<button-1>", affiche_touche_pressee)
  File "c:\python26\lib\lib-tk\Tkinter.py", line 933, in bind
    return self._bind(('bind', self._w), sequence, func, add)
  File "c:\python26\lib\lib-tk\Tkinter.py", line 888, in _bind
    self.tk.call(what + (sequence, cmd))
_tkinter.TclError: bad event type or keysym "button"
```

### Remarque 8.8 : associer un événement à tous les objets

Il arrive parfois qu'un événement ne doive pas être associé à un seul objet mais à tous ceux que la fenêtre contient. C'est l'objectif de la méthode `bind_all`. Sa syntaxe est exactement la même que la méthode `bind`.

```
b.bind_all ("<Button-1>", affiche_touche_pressee)
```

On utilise peu cette fonction, on préfère construire des objets propres à un programme comme suggéré au paragraphe 8.6.2.

**Remarque 8.9 : désactiver un événement**
De la même manière qu'il est possible d'associer un événement à un objet d'une fenêtre, il est possible d'effectuer l'opération inverse qui consiste à supprimer cette association. La méthode `unbind` désactive un événement associé à un objet. La méthode `unbind_all` désactive un événement associé pour tous les objets d'une fenêtre.

**syntaxe 8.10 : méthode `unbind`**

```
w.unbind (ev)
w.unbind_all (ev)
```

`w` est l'identificateur de l'objet interceptant l'événement désigné par la chaîne de caractères `ev` (voir table 8.2). Après l'appel à la méthode `unbind`, l'événement n'est plus intercepté par l'objet `w`. Après l'appel à la méthode `unbind_all`, l'événement n'est plus intercepté par aucun objet.

**Remarque 8.11 : événement spécifique**
Il est possible de définir des événements propres aux programmes. Ceux-ci ne sont générés par aucun périphérique mais explicitement par le programme lui-même. Ce mécanisme est presque toujours couplé à l'utilisation de threads. Le paragraphe 9.3 (page 231) illustre ce principe à l'aide d'un exemple à base de thread. Le paragraphe 8.6.5 page 224 propose un exemple plus simple.

### 8.4.5 Menu

Les menus fonctionnent de la même manière que les boutons. Chaque intitulé du menu est relié à une fonction qui sera exécutée à la condition que l'utilisateur sélectionne cet intitulé. L'objet `Menu` ne désigne pas le menu dans son ensemble mais seulement un niveau. Par exemple, le menu présenté par la figure 8.17 est en fait un assemblage de trois menus auquel on pourrait ajouter d'autres sous-menus.

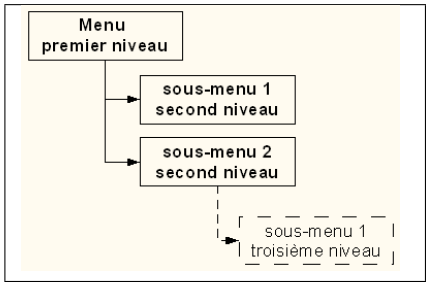



**Figure 8.16 :** *La représentation d'un menu tient plus d'un graphe que d'une liste. Chaque intitulé du menu peut être connecté à une fonction ou être le point d'entrée d'un nouveau sous-menu.*

Pour créer un menu ou un sous-menu, il suffit de créer un objet de type `Menu` :

```
m = Tkinter.Menu ()
```

Ce menu peut être le menu principal d'une fenêtre auquel cas, il suffit de préciser à la fenêtre en question que son menu est le suivant :

```
root.config (menu = m)
```

root est ici la fenêtre principale mais ce pourrait être également une fenêtre de type TopLevel (voir paragraphe 8.5.1). Ce menu peut aussi être le sous-menu associé à un intitulé d'un menu existant. La méthode add_cascade permet d'ajouter un sous-menu associé à un label :

```
mainmenu  = Tkinter.Menu ()
msousmenu = Tkinter.Menu ()
mainmenu.add_cascade (label = "sous-menu 1", menu = msousmenu)
```

En revanche, si on souhaite affecter une fonction à un menu, on utilisera la méthode add_command :

```
def fonction1 () :
    ....
m = Tkinter.Menu ()
mainmenu.add_command (label = "fonction 1", command = fonction1)
```

L'exemple suivant regroupe les fonctionnalités présentées ci-dessus.

```
import Tkinter
root = Tkinter.Tk ()

e = Tkinter.Text (width = 50, height = 10)
e.pack ()

m = Tkinter.Menu ()

sm1 = Tkinter.Menu ()
sm2 = Tkinter.Menu ()

m.add_cascade (label = "sous-menu 1", menu = sm1)
m.add_cascade (label = "sous-menu 2", menu = sm2)

nb = 0

def affiche () : print "fonction affiche"
def calcul () : print "fonction calcul ", 3 * 4
def ajoute_bouton () :
    global nb
    nb += 1
    b = Tkinter.Button (text = "bouton " + str (nb))
    b.pack ()

sm1.add_command (label = "affiche",       command = affiche)
sm1.add_command (label = "calcul",        command = calcul)
sm2.add_command (label = "ajoute_bouton", command = ajoute_bouton)
sm2.add_command (label = "fin",           command = root.destroy)

root.config (menu = m, width = 200)
root.title ("essai de menu")
#help (Tkinter.Tk)
root.mainloop ()
```


**Figure 8.17 :** *Résultat de l'exemple du paragraphe 8.4.5, la sélection de différents intitulés du menu permet d'afficher des choses avec l'instruction* `print` *ou d'ajouter des boutons.*

Chaque intitulé d'un menu est ajouté en fin de liste, il est possible d'en supprimer certains à partir de leur position avec la méthode `delete` :

```
m = Tkinter.Menu ()
m.add_command (...)
m.delete (1, 2) # supprime le second intitulé
                # supprime les intitulés compris entre 1 et 2 exclu
```

### 8.4.6 Fonctions prédéfinies

Il est possible de détruire la fenêtre principale, ce qui mettra fin au programme si celui-ci ne prévoit rien après la fonction `mainloop`. La destruction de la fenêtre s'effectue par la méthode `destroy`. Le programme suivant crée une fenêtre avec un seul bouton qui, s'il est pressé, mettra fin à l'application.

```
import Tkinter
root = Tkinter.Tk ()
Tkinter.Button (text = "fin", command = root.destroy).pack ()
root.mainloop ()
```

La table 8.3 regroupe les fonctions les plus utilisées. Celles-ci s'applique à une fenêtre de type `Toplevel` qui est aussi le type de la fenêtre principale.

| | |
|---|---|
| `destroy()` | Détruit la fenêtre. |
| `deiconify()` | La fenêtre reprend une taille normale. |
| `geometry(s)` | Modifie la taille de la fenêtre. `s` est une chaîne de caractères de type "`wxhxy`". `w` et `h` sont la largeur et la hauteur. `x` et `y` sont la position du coin supérieur haut à l'écran. |
| `iconify()` | La fenêtre se réduit à un icône. |
| `resizable(w,h)` | Spécifie si la fenêtre peut changer de taille. `w` et `h` sont des booléens. |
| `title(s)` | Change le titre de la fenêtre, `s` est une chaîne de caractères. |
| `withdraw()` | Fait disparaître la fenêtre. La fonction inverse est `deiconify`. |

**Table 8.3 :** *Liste non exhaustives des méthodes de la classe* `Toplevel` *qui le type de la fenêtre principale.*

## 8.5 D'autres fenêtres

### 8.5.1 Créer une seconde boîte de dialogues

Lorsqu'un programme doit utiliser plusieurs fenêtres et non pas une seule, l'emploi de l'objet `Toplevel` est inévitable. L'instruction `root = Tkinter.Tk()` crée la fenêtre principale, l'instruction `win = Tkinter.Toplevel()` crée une seconde fenêtre qui fonctionne exactement comme la fenêtre principale puisqu'elle dispose elle aussi d'une boucle de messages via la méthode `mainloop`.

```
import Tkinter
win = Tkinter.Toplevel ()
win.mainloop ()
```

Un cas d'utilisation simple est par exemple un bouton pressé qui fait apparaître une fenêtre permettant de sélectionner un fichier, cette seconde fenêtre sera un objet `Toplevel`. Il n'est pas nécessaire de s'étendre plus sur cet objet, son comportement est identique à celui de la fenêtre principale, les fonctions décrites au paragraphe 8.4.6 s'appliquent également aux objets `Toplevel`. Il reste néanmoins à préciser un dernier point. Tous les objets précédemment décrits au paragraphe 8.2 doivent inclure un paramètre supplémentaire dans leur constructeur pour signifier qu'ils appartiennent à un objet `Toplevel` et non à la fenêtre principale. Par exemple, pour créer une zone de texte, la syntaxe est la suivante :

```
# zone_texte appartient à la fenêtre principale
zone_texte = Tkinter.Label (text = "premier texte")
```

Pour l'inclure à une fenêtre `Toplevel`, cette syntaxe devient :

```
# zone_texte appartient à la fenêtre top
top = Tkinter.Toplevel ()
zone_texte = Tkinter.Label (top, text = "premier texte")
```

Lors de la définition de chaque objet ou *widget*, si le premier paramètre est de type `Toplevel`, alors ce paramètre sera affecté à la fenêtre passée en premier argument et non à la fenêtre principale. Ce principe est le même que celui de la sous-fenêtre `Frame` (voir paragraphe 8.3.2, page 208). La seule différence provient du fait que l'objet `Toplevel` est une fenêtre autonome qui peut attendre un message grâce à la méthode `mainloop`, ce n'est pas le cas de l'objet `Frame`.

Toutefois, il est possible d'afficher plusieurs fenêtre `Toplevel` simultanément. Le programme suivant en est un exemple :

```
# coding: latin-1
import Tkinter

class nouvelle_fenetre :
    resultat = []
    def top (self) :
        sec = Tkinter.Toplevel ()
        Tkinter.Label (sec, text="entrer quelque chose").pack ()
        saisie = Tkinter.Entry (sec)
```

```
        saisie.pack()
        Tkinter.Button (sec, text = "valider", command = sec.quit).pack ()
        sec.mainloop ()
        nouvelle_fenetre.resultat.append ( saisie.get () )
        sec.destroy ()

root = Tkinter.Tk() #fenetre principale
a = Tkinter.Button (text    = "fenêtre Toplevel",
                    command = nouvelle_fenetre ().top)
a.pack()
root.mainloop()

for a in nouvelle_fenetre.resultat :
    print "contenu ", a
```

### 8.5.2 Fenêtres standard

Le module `Tix` propose une fenêtre de sélection de fichiers identique à celle de la figure 8.18. `Tkinter` a l'avantage d'être simple et ne nécessite pas un long apprentissage pour le maîtriser mais il est limité. Pour ce type de fenêtres qu'on retrouve dans la plupart des programmes, il existe presque toujours des solutions toutes faites, via le module `Tix` par exemple. On trouve également de nombreux programmes sur Internet par le biais de moteurs de recherche. Le programme ci-dessous affiche une fenêtre qui permet de sélectionner un fichier.

```
import Tix as Tk
root = Tk.Tk ()

def command_print () : print box.cget("value")

box = Tk.FileSelectBox (root)
box.config (directory="c:\\")
box.pack ()
Tk.Button (root, text = "print", command = command_print).pack ()

root.mainloop ()
```

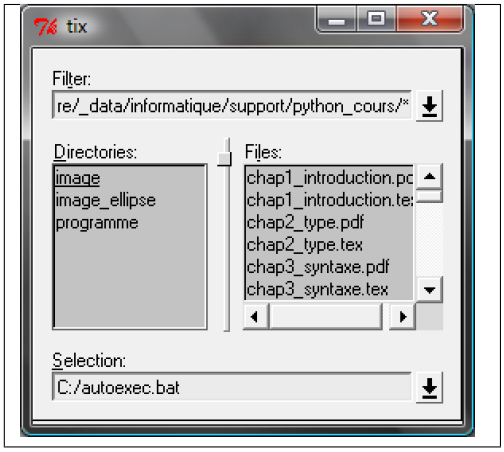



**Figure 8.18 :** *Exemple d'une fenêtre graphique. Celle-ci permet de sélectionner un fichier. C'est une fenêtre prédéfinie du module* `Tix` *qui est une extension de* `Tkinter`.

## 8.6 Constructions classiques

L'objectif des paragraphes qui suivent est d'introduire quelques schémas de construction d'interfaces qui reviennent fréquemment. La première règle de programmation qu'il est préférable de suivre est d'isoler la partie interface du reste du programme. La gestion événementielle a pour défaut parfois de disséminer un traitement, un calcul à plusieurs endroits de l'interface. C'est le cas par exemple de longs calculs dont on souhaite connaître l'avancée. Le calcul est lancé par la pression d'un bouton puis son déroulement est "espionné" par un événement régulier comme un compte à rebours.

Le principal problème des interfaces survient lors du traitement d'un événement : pendant ce temps, l'interface n'est pas réactive et ne réagit plus aux autres événements jusqu'à ce que le traitement de l'événement en cours soit terminé. Pour contourner ce problème, il est possible soit de découper un calcul en petites fonctions chacune très rapide, cela suppose que ce calcul sera mené à bien par une succession d'événements. Il est également possible de lancer un thread, principe décrit au paragraphe 9.3 (page 231).

C'est pourquoi la première règle est de bien scinder interface et calculs scientifiques de façon à pouvoir rendre le programme plus lisible et ainsi être en mesure d'isoler plus rapidement la source d'une erreur. Les paragraphes qui suivent présentent quelques aspects récurrents qu'il est parfois utile d'avoir en tête avant de se lancer.

### 8.6.1 Compte à rebours

Il est possible de demander à un objet d'appeler une fonction après un certains laps de temps exprimé un millisecondes. Le programme suivant crée un objet de type `Label`. Il contient une fonction qui change son contenu et lui affecte un compte à rebours qui impose à l'objet de rappeler cette fonction 1000 millisecondes plus tard. Le résultat est un programme qui crée la fenêtre 8.19 et change son contenu toutes les secondes.

```
import Tkinter
root = Tkinter.Tk ()
l = Tkinter.Label (text = "0 secondes")
l.pack ()
sec = 0
id = None

def change_legende() :
    global l
    global sec
    global id
    sec += 1
    l.config (text = "%d secondes" % sec)
    id = l.after (1000, change_legende)

l.after (1000, change_legende)

root.mainloop ()
```


**Figure 8.19 :** *Résultat de l'exemple du paragraphe 8.6.1, l'intitulé de l'objet* `Label` *change toutes les secondes.*

La méthode `after` retourne un entier permettant d'identifier le compte à rebours qu'il est possible d'interrompre en utilisant la méthode `after_cancel`. Dans l'exemple précédent, il faudrait utiliser l'instruction suivante :

```
    l.after_cancel (id)
```

### 8.6.2 Contrôles personnalisés : utilisation des classes

On peut personnifier un contrôle. Par exemple, on peut mettre en évidence l'intitulé d'une liste sous le curseur de la souris. Le moyen le plus simple est de créer une nouvelle classe qui se substituera au classique `Listbox`. Il suffit que cette nouvelle classe hérite de `Listbox` en prenant soin de lui donner un constructeur reprenant les mêmes paramètres que celui de la classe `Listbox`. De cette façon, il suffit de remplacer `Listbox` par `MaListbox` pour changer l'apparence d'une liste.

```
# coding: latin-1
import Tkinter

class MaListbox (Tkinter.Listbox) :
    def __init__ (self, master = None, cnf={}, **kw) :
        Tkinter.Listbox.__init__ (self, master, cnf, **kw)
        self.bind ("<Motion>", self.mouvement)
        self.pos = None  # mémoire l'ancienne position du curseur
    def mouvement (self, ev) :
        pos = self.nearest (ev.y)  # nouvelle position du curseur
        if pos < 0 or pos >= self.size () : return
        if self.pos != pos :
            if self.pos != None : self.itemconfigure(self.pos, bg='')
            self.itemconfigure (pos, bg='gray')
            self.pos = pos

root = Tkinter.Tk ()
b = MaListbox ()
b.insert ("end", "ligne 1")
b.insert ("end", "ligne 2")
b.insert ("end", "ligne 3")
b.pack ()
b.focus_set ()
root.mainloop ()
```

Dans ce cas précis, on fait en sorte que le contrôle intercepte le mouvement du curseur. Lorsque celui-ci bouge, la méthode `mouvement` est appelée comme le constructeur de `MaListbox` l'a spécifié. La méthode `nearest` permet de définir l'intitulé le plus proche du curseur. La méthode `itemconfigure` permet de changer le fond de cet intitulé en gris après avoir modifié le fond de l'intitulé précédent pour qu'il retrouve sa couleur d'avant. Le résultat est illustré la figure 8.20.

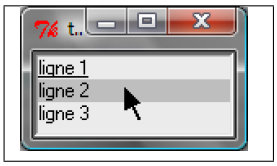



**Figure 8.20 :** *Aspect de la classe* `MaListbox` *définie par le programme 220. L'intitulé sous le curseur de la souris a un fond gris.*

### 8.6.3 Fenêtres personnalisées : utilisation des classes

Cet exemple prolonge l'idée du paragraphe précédent. Lorsque l'interface devient complexe, il peut être utile de créer ses propres fenêtres. Jusqu'à présent, seules des fonctions ont été attachées à événement comme la pression d'un bouton mais il est possible d'attacher la méthode d'une classe ce que développe l'exemple qui suit.

```
# coding: latin-1
import Tkinter as Tk

class MaFenetre :
    def __init__ (self, win) :
        self.win = win
        self.creation ()

    def creation (self) :
        b1 = Tk.Button (self.win, text="bouton 1", command=self.commande_bouton1)
        b2 = Tk.Button (self.win, text="bouton 2", command=self.commande_bouton2)
        b3 = Tk.Button (self.win, text="disparition", command=self.disparition)
        b1.grid (row=0, column=0)
        b2.grid (row=0, column=1)
        b3.grid (row=0, column=2)
        self.lab = Tk.Label (self.win, text = "-")

    def commande_bouton1 (self) :
        # on déplace l'objet lab de type Label
        self.lab.configure (text = "bouton 1 appuyé")
        self.lab.grid (row = 1, column = 0)

    def commande_bouton2 (self) :
        # on déplace l'objet lab de type Label
        self.lab.configure (text = "bouton 2 appuyé")
        self.lab.grid (row = 1, column = 1)

    def disparition (self) :
        # on fait disparaître l'objet lab de type Label
        self.lab.grid_forget ()

if __name__ == "__main__" :
    root = Tk.Tk ()
    f = MaFenetre (root)
    root.mainloop ()
```

Ce programme crée trois boutons et attache à chacun d'entre eux une méthode de la classe `MaFenetre`. Le constructeur de la classe prend comme unique paramètre un pointeur sur un objet qui peut être la fenêtre principale, un objet de type `Frame` ou `Toplevel`. Cette construction permet de considérer cet ensemble de trois boutons

comme un objet à part entière ; de ce fait il peut être inséré plusieurs fois comme le montre l'exemple suivant illustré par la figure 8.21.

```
root = Tk.Tk ()
f = Tk.Frame ()
f.pack ()
MaFenetre (f)      # première instance
g = Tk.Frame ()
g.pack ()
MaFenetre (g)      # seconde instance
root.mainloop ()
```

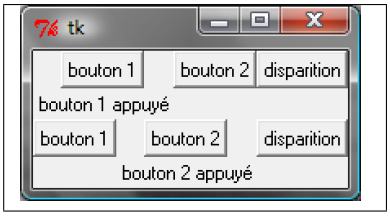



**Figure 8.21 :** *Aspect de la classe* `MaFenetre` *définie au paragraphe 8.6.3. La fenêtre est en fait composée de deux instances de* `MaFenetre`.

### 8.6.4 Séquence d'événements

Il est facile de faire réagir le programme en fonction d'un événement, il suffit d'attacher cet événement à une méthode ou une fonction. En revanche, faire réagir le programme en fonction d'une séquence d'événements est plus complexe. En effet, le premier d'événement de la séquence active une fonction, il n'est pas possible d'attendre le second événement dans cette même fonction, ce dernier ne sera observable que si on sort de cette première fonction pour revenir à la fonction `mainloop`, la seule capable de saisir le prochain événement.

La figure 8.22 précise la gestion des messages. `Tkinter` se charge de la réception des messages puis de l'appel au traitement correspondant indiqué par la méthode ou la fonction attachée à l'événement. Le programmeur peut définir les traitements associés à chaque événement. Ces deux parties sont scindées et à moins de reprogrammer sa boucle de message, il n'est pas évident de consulter les événements intervenus depuis le début du traitement de l'un d'eux.

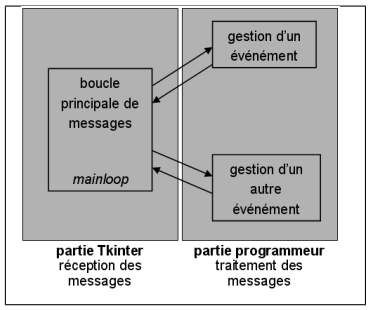


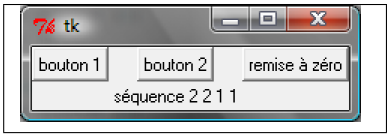

**Figure 8.22 :** *La réception des événements est assurée par la fonction* `mainloop` *qui consiste à attendre le premier événement puis à appeler la fonction ou la méthode qui lui est associée si elle existe.*

Les classes offrent un moyen simple de gérer les séquences d'événements au sein d'une fenêtre. Celle-ci fera l'objet d'une classe qui mémorise les séquences d'événements.

Tous les événements feront appel à des méthodes différentes, chacune d'elles ajoutera l'événement à une liste. Après cette ajout, une autre méthode sera appelée pour rechercher une séquence d'événements particulière. Le résultat est également illustré par la figure 8.23.

```
# coding: latin-1
import Tkinter as Tk

class MaFenetreSeq :
    def __init__ (self, win) :
        self.win = win
        self.creation ()
        self.sequence = []

    def creation (self) :
        b1 = Tk.Button (self.win, text="bouton 1", command=self.commande_bouton1)
        b2 = Tk.Button (self.win, text="bouton 2", command=self.commande_bouton2)
        b3 = Tk.Button (self.win, text="remise à zéro", command=self.zero)
        b1.grid (row=0, column=0)
        b2.grid (row=0, column=1)
        b3.grid (row=0, column=2)
        self.lab = Tk.Label (self.win, text = "-")

    def commande_bouton1 (self) :
        # ajoute 1 à la liste self.sequence
        self.sequence.append (1)
        self.controle ()

    def commande_bouton2 (self) :
        # ajoute 2 à la liste self.sequence
        self.sequence.append (2)
        self.controle ()

    def zero (self) :
        # on vide la liste self.sequence
        self.sequence = []
        self.lab.grid_forget ()

    def controle (self) :
        # on compare la liste sequence à [1,2,1] et [2,2,1,1]
        # dans ce cas, on fait apparaître l'objet lab
        l = len (self.sequence)
        if l >= 3 and self.sequence [l-3:] == [1,2,1] :
            self.lab.configure (text = "séquence 1 2 1")
            self.lab.grid (row = 1, column = 0)
        elif l >= 4 and self.sequence [l-4:] == [2,2,1,1] :
            self.lab.configure (text = "séquence 2 2 1 1")
            self.lab.grid (row = 1, column = 1)

if __name__ == "__main__" :
    root = Tk.Tk ()
    f = MaFenetreSeq (root)
    root.mainloop ()
```

**Figure 8.23 :** *Aspect de la classe* `MaFenetreSeq` *définie au paragraphe 8.6.4.*




### 8.6.5 Communiquer un résultat par message

Le module `Tkinter` permet de définir ses propres messages qui peuvent servir à communiquer des informations. Une fonction est par exemple appelée lorsqu'un bouton est pressé. Celle-ci, une fois terminée, retourne son résultat sous forme de message envoyé à l'interface graphique. Ce message sera ensuite traité comme tout autre message et pourra être intercepté ou non.

Le programme suivant utilise ce concept. La pression d'un bouton appelle une fonction `event_generate` qui génère un message personnalisé `<<perso>>` avec comme paramètre `rooty` $= -5$. A son tour, celui-ci est attrapé et dirigé vers la fonction `perso` qui affiche l'attribut `y_root` de la classe `Event` qui a reçu la valeur $-5$ lors de l'appel de la fonction `event_generate`. Ce procédé ne permet toutefois que de renvoyer que quelques résultats entiers.

```
# coding: latin-1
import Tkinter
def affiche_touche_pressee () : root.event_generate ("<<perso>>", rooty = -5)
def perso (evt) : print "perso", evt.y_root
root = Tkinter.Tk ()
b = Tkinter.Button (text = "clic", \
                    command = affiche_touche_pressee)
b.pack ()
root.bind ("<<perso>>", perso)  # on intercepte un événement personnalisé
root.mainloop ()
```

Ce principe est plus utilisé lorsque l'interface graphique est couplée avec les threads, l'ensemble est présenté au paragraphe 9.3.

# Chapitre 9

# Threads

---

Jusqu'aux années 2003-2004, l'évolution des microprocesseurs était une course vers une augmentation de la puissance, autant en terme de nombre de transistors qu'en fréquence de fonctionnement. Arrivant aux limites de la technologie actuelle, cette évolution s'est tournée maintenant vers la construction de processeurs multicœurs, c'est-à-dire des machines capables d'exécuter des programmes simultanément, de maintenir plusieurs *fils d'exécution* en parallèle.

Les threads ou fils d'exécution ont trois usages principaux. Le premier est relié au calcul distribué ou calcul parallèle. Par exemple, le calcul d'une intégrale sur un intervalle peut être effectué sur deux intervalles disjoints. Le résultat final est la somme des deux résultats sur chacun des intervalles. De plus, ces deux calculs sont indépendants et peuvent être menés de front. Le calcul intégral sera donc deux fois plus rapide puisque les deux intervalles seront traités en même temps. C'est la parallélisation des calculs : les deux calculs sur chaque intervalle seront affectés à deux threads simultanés.

Le second usage est couplé aux interfaces graphiques. Lorsque l'utilisateur entame un processus long après avoir cliqué sur un bouton, l'interface graphique ne réagit plus jusqu'à ce que ce processus s'achève. Afin d'éviter cet inconvénient, l'interface graphique va commencer un thread qui va exécuter ce processus. L'interface graphique n'a plus qu'à attendre la fin du thread, et pendant tout ce temps, elle sera également capable de traiter tout autre événement provenant de l'utilisateur.

Le dernier usage concerne la communication entre ordinateurs ou plus généralement la communication Internet. C'est une communication asynchrone : l'ordinateur effectue des tâches en même temps qu'il écoute un *port* par lequel d'autres ordinateurs communiquent avec lui. Plus précisément, le programme suit deux fils d'exécution : le fil principal et un thread qui ne fait qu'attendre et traiter les messages qu'il reçoit via un port.

La synchronisation est un point commun à ces trois usages. Ce terme désigne la dépendance entre les threads. Lors d'un calcul distribué, le résultat final dépend des résultats retournés par chaque thread, il faut donc attendre que les deux fils d'exécution aient produit le résultat attendu : il faut que les deux fils d'exécution se synchronisent.

Ce document ne s'étendra pas longuement sur les threads bien qu'ils soient amenés à devenir un élément incontournable de tout programme désirant tirer parti des derniers processeurs.

## 9.1 Premier thread

Le premier exemple consiste à exécuter un thread uniquement destiné à faire des affichages. Deux fils d'exécution vont être lancés en parallèle affichant chacun un message différent. Les affichages vont s'entremêler. Il existe plusieurs manières d'exécuter un thread, une seule sera présentée en utilisant la classe `Thread` du module `threading`. Pour créer un thread, il suffit de surcharger la méthode `run` de la classe `Thread`. Si le thread a besoin de données lors de son exécution, il faut surcharger son constructeur sans oublier d'appeler le constructeur de la classe mère. L'exécution de thread commence par la création d'une instance et l'appel à la méthode `start`. En résumé, il faut retenir les éléments suivants :

1. surcharger la classe `threading.Thread`,
2. surcharger le constructeur sans oublier d'appeler le constructeur `threading.Thread.__init__`,
3. surcharger la méthode `run`, c'est le code que devra exécuter le thread,
4. créer une instance de la nouvelle classe et appeler la méthode `start` pour lancer le thread secondaire qui formera le second fil d'exécution.

Le programme principal est appelé le thread principal. Voici ce que cela donne dans un exemple :

```
# coding: cp1252
import threading, time

class MonThread (threading.Thread) :
    def __init__ (self, jusqua) :      # jusqua = donnée supplémentaire
        threading.Thread.__init__(self)# ne pas oublier cette ligne
                                       # (appel au constructeur de la classe mère)
        self.jusqua = jusqua           # donnée supplémentaire ajoutée à la classe

    def run (self) :
        for i in range (0, self.jusqua) :
            print "thread ", i
            time.sleep (0.1)    # attend 100 millisecondes sans rien faire
                                # facilite la lecture de l'affichage

m = MonThread (1000)        # crée le thread
m.start ()                  # démarre le thread,
                            # l'instruction est exécutée en quelques millisecondes
                            # quelque soit la durée du thread

for i in range (0,1000) :
    print "programme ", i
    time.sleep (0.1)            # attend 100 millisecondes sans rien faire
                                # facilite la lecture de l'affichage
```

Le programme affiche des lignes qui proviennent du thread principal et du thread secondaire dont les affichages diffèrent.

```
programme  0
thread  0
thread  1
programme  1
thread  2
```

```
programme  2
programme  3
thread  3
programme  4
thread  4
...
```

Le précédent programme a été adapté pour lancer deux threads secondaires en plus du thread principal. Les lignes modifiées par rapport au programme précédent sont commentées.

```
# coding: cp1252
import threading, time

class MonThread (threading.Thread) :
    def __init__ (self, jusqua, s) :
        threading.Thread.__init__ (self)
        self.jusqua = jusqua
        self.s = s

    def run (self) :
        for i in range (0, self.jusqua) :
            print "thread ", self.s, " : ", i
            time.sleep (0.1)

m = MonThread (1000, "A")
m.start ()

m2 = MonThread (1000, "B")  # crée un second thread
m2.start ()                 # démarre le thread,

for i in range (0,1000) :
    print "programme ", i
    time.sleep (0.1)
```

```
thread  A  :  0
programme  0
thread  B  :  0
thread  A  :  1
thread  B  :  1
programme  1
thread  B  :  2
thread  A  :  2
...
```

**Remarque 9.1 : utilisation le la fonction `sleep`**
Tous les exemples présentés dans ce chapitre font souvent intervenir l'instruction `time.sleep`(...). A moins que ce ne soit explicitement précisé, elle sert la plupart du temps à ralentir l'exécution du programme cité en exemple afin que celle-ci soit humainement observable ou pour exagérer un défaut de synchronisation. Cette fonction est d'ordinaire beaucoup moins fréquente.

## 9.2 Synchronisation

### 9.2.1 Attente

La première situation dans laquelle on a besoin de synchroniser deux threads est l'attente d'un thread secondaire par le thread principal. Et pour ce faire, on a besoin de l'accès par les deux fils d'exécution à une même variable qui indiquera l'état du thread. Dans le programme suivant, on ajoute l'attribut `etat` à la classe `MonThread` qui va indiquer l'état du thread :

– `True` pour en marche
– `False` pour à l'arrêt

Le thread principal va simplement vérifier l'état du thread de temps en temps. Le premier point important est tout d'abord d'attendre que le thread se lance car sans la première boucle, le thread pourrait passer à l'état `True` après être passé dans la seconde boucle d'attente. Le second point important est de ne pas oublier d'insérer la fonction `sleep` du module `time` afin de permettre au thread principal de temporiser. Dans le cas contraire, le thread principal passe l'essentiel de son temps à vérifier l'état du thread secondaire, ce faisant, il ralentit l'ordinateur par la répétition inutile de la même action un trop grand nombre de fois. Ici, le thread principal vérifie l'état du thread secondaire tous les 100 millisecondes. Cette durée dépend de ce que fait le thread secondaire.

```
# coding: latin-1
import threading, time

class MonThread (threading.Thread) :
    def __init__ (self, jusqua) :
        threading.Thread.__init__ (self)
        self.jusqua = jusqua
        self.etat = False       # l'état du thread est soit False (à l'arrêt)
                                # soit True (en marche)

    def run (self) :
        self.etat = True                        # on passe en mode marche
        for i in range (0, self.jusqua) :
            print "thread itération ", i
            time.sleep (0.1)
        self.etat = False                       # on revient en mode arrêt

m = MonThread (10)          # crée un thread
m.start ()                  # démarre le thread,

print "début"

while m.etat == False :
    # on attend que le thread démarre
    time.sleep (0.1)  # voir remarque ci-dessous

while m.etat == True :
    # on attend que le thread s'arrête
    # il faut introduire l'instruction time.sleep pour temporiser, il n'est pas
    # nécessaire de vérifier sans cesse que le thread est toujours en marche
    # il suffit de le vérifier tous les 100 millisecondes
    # dans le cas contraire, la machine passe son temps à vérifier au lieu
    # de se consacrer à l'exécution du thread
    time.sleep (0.1)

print "fin"
```

Ce mécanisme d'attente peut également être codé en utilisation les objets `Condition` et `Event` du module `threading`. Ces deux objets permettent d'éviter l'utilisation de la méthode `sleep`.

```
# coding: latin-1
import threading, time
```

```
class MonThread (threading.Thread) :
    def __init__ (self, jusqua, event) :    # event = objet Event
        threading.Thread.__init__ (self)    #       = donnée supplémentaire
        self.jusqua = jusqua
        self.event  = event                 # on garde un accès à l'objet Event

    def run (self) :
        for i in range (0, self.jusqua) :
            print "thread itération ", i
            time.sleep (0.1)
        self.event.set ()                   # on indique qu'on a fini :
                                            # on active l'object self.event
print "début"

event = threading.Event ()       # on crée un objet de type Event
event.clear ()                   # on désactive l'ojet Event
m = MonThread (10, event)        # crée un thread
m.start ()                       # démarre le thread,
event.wait ()                    # on attend jusqu'à ce que l'objet soit activé
                                 # event.wait (0.1) : n'attend qu'un
print "fin"                      #          seulement 1 dizième de seconde
```

La méthode `wait` de l'objet `Event` attend que l'objet soit activé. Elle peut attendre indéfiniment ou attendre pendant une durée donnée seulement. Pour afficher la durée d'attente, on pourrait utiliser une boucle comme la suivante :

```
m.start ()
while not event.isSet ():
    print "j'attends"
    event.wait (0.1)
print "fin"
```

La méthode `isSet` permet de savoir si l'événement est bloquant ou non. Le programme affiche "`j'attends`" puis attend le thread un dixième de secondes. Au delà de cette durée, il vérifie l'état de l'événement puis recommence si le thread n'est pas fini.

Ces objets de synchronisation sont plus efficaces que le mécanisme décrit dans le premier programme car il fait appel aux fonctions du système d'exploitation.

### 9.2.2 Partage d'informations

La seconde situation dans laquelle on a besoin de synchroniser est l'accès par deux fils d'exécution aux mêmes informations ou plutôt aux mêmes variables. Un problème survient quand parfois un thread lit ou modifie en même temps qu'un autre modifie la même variable. Le second cas de synchronisation est l'ajout de verrous qui permettent de protéger une partie du code d'un programme contre plusieurs accès simultanés. Ce verrou est également un objet du module `threading`.

Dans cet exemple, l'information partagée est la chaîne de caractères `message`, le verrou sert à protéger la fonction `ajoute` contre des ajouts simultanés. Si les deux threads veulent modifier `message` en même temps, un thread va entrer dans la fonction `ajoute` alors que l'autre n'en est pas encore sorti. Les résultats seraient imprévisibles car cette fonction modifie la variable qu'ils utilisent. On aboutit à l'exemple suivant :

```
# coding: latin-1
import threading, time
```

```
message = ""
verrou  = threading.Lock ()

def ajoute (c) :
    global message     # message et verrou sont des variables gloables
    global verrou      # pour ne pas qu'elle disparaisse dès la fin de la fonction
    verrou.acquire ()  # on protège ce qui suit  (*)

    s = message + c    # instructions jamais exécutée simultanément par 2 threads
    time.sleep (0.001) # time.sleep : pour exagérer le défaut de synchronisation
    message = s        # si verrou n'est pas utilisé

    verrou.release ()  # on quitte la section protégée  (*)

class MonThread (threading.Thread) :
    def __init__ (self, jusqua, event, s) :
        threading.Thread.__init__ (self)
        self.jusqua = jusqua
        self.s      = s
        self.event  = event

    def run (self) :
        for i in range (0, self.jusqua) :
            ajoute (self.s)
            time.sleep (0.003)
        self.event.set ()

print "début"

# synchronisation attente
e1 = threading.Event ()
e2 = threading.Event ()
e1.clear ()
e2.clear ()

m1 = MonThread (10, e1, "1")     # crée un thread
m1.start ()                      # démarre le thread,
m2 = MonThread (10, e2, "2")     # crée un second thread
m2.start ()                      # démarre le second thread,

e1.wait ()
e2.wait ()

print "longueur ", len(message) # affiche 20
print "message = ", message     # affiche quelque chose comme 12212112211212121221
```

Les trois instructions protégées pourraient être résumées en une seule : `message += c` ; le résultat resterait inchangé. En revanche, en commentant les instructions `verrou.acquire()` et `verrou.release()` de ce programme [1], la longueur du résultat final `message` est variable alors qu'elle devrait être de 20 puisque les deux threads appellent chacun 10 fois dans la fonction `ajoute`. Le tableau suivant montre l'évolution des variables `message`, `c`, `s` durant deux premiers appels qui s'entremêlent. Le résultat devrait être "12" pour message mais un caractère a été perdu. Il faut retenir que si

1. Celles marquées d'une étoile (∗).

la variable `message` est globale, les deux autres `c`, `s` sont locales et donc différentes pour les deux threads.

| ordre | thread 1 | thread 2 | message | c | s |
|---|---|---|---|---|---|
| 1 | `s = message + c` | | "" | "1" | "1" |
| 2 | | `s = message + c` | "" | "2" | "2" |
| 3 | `time.sleep(0.001)` | | "" | "1" | "1" |
| 4 | | `time.sleep(0.001)` | "" | "2" | "2" |
| 5 | `message = s` | | "1" | "1" | "1" |
| 6 | | `message = s` | "2" | "2" | "2" |

Le verrou empêche d'exécuter une même portion de code en même temps, un code qui modifie des données partagées. C'est pourquoi le verrou est souvent déclaré au même endroit que les données qu'il protège. Le verrou de type `Lock` n'autorise qu'un seul thread à la fois à l'intérieur de la portion de code protégée ce qui aboutit au schéma suivant :

| ordre | thread 1 | thread 2 | message | c | s |
|---|---|---|---|---|---|
| 1 | `s = message + c` | | "" | "1" | "1" |
| 2 | `time.sleep(0.001)` | | "" | "1" | "1" |
| 3 | `message = s` | | "1" | "1" | "1" |
| 4 | | `s = message + c` | "1" | "2" | "12" |
| 5 | | `time.sleep(0.001)` | "1" | "2" | "12" |
| 6 | | `message = s` | "12" | "2" | "12" |

**Remarque 9.2 : réduction des accès à quelques threads**
Le verrou de type `Semaphore` autorise un nombre maximal de thread à parcourir le même code. Ce procédé est parfois utile si le code en question permet d'imprimer un document. Cela permet de limiter sans interdire les accès simultanés aux ressources de l'ordinateur.

**Remarque 9.3 : blocage d'un programme**
Ce mécanisme de verrou peut aboutir à des blocages avec deux threads et deux portions de code protégées. Chaque thread est "coincé" dans une section attendant que l'autre libère la sienne. Dans ce cas de figure, il est conseillé d'utiliser le même verrou pour protéger les deux sections. Ainsi, chaque thread ne pourra pas entrer dans l'une ou l'autre des portions de code protégées tant que l'une d'entre elles est visitée par l'autre thread.

## 9.3 Interface graphique

Un programme bâti autour d'une interface graphique inclut nécessairement une boucle de message. Celle-ci attend les messages en provenance de l'interface. Lorsqu'un de ceux-ci lui commande de lancer un traitement long, l'interface graphique n'est plus en mesure de réagir aux événements qui lui viennent pendant ce temps. Afin de remédier cela, il suffit d'insérer le traitement dans un thread. A la fin de ce dernier, un événement sera envoyé à l'interface afin de lui signifier la fin du traitement.

Le paragraphe 8.4.4 a montré comment associer un événement particulier à une fenêtre. La différence ici est que l'événement accroché à la fenêtre n'est pas prédéfini par le module `Tkinter` mais par le programme lui-même - dans cet exemple `<<thread_fini>>`[2] -. La méthode `event_generate` permet d'insérer un message dans la boucle de messages de façon à ce que celui-ci soit traité au même titre qu'un clic de souris, la pression d'une touche, ...

```
# coding: latin-1
import threading, time, random, copy

# définition du thread
class MonThread (threading.Thread) :
    def __init__ (self, win, res) :
        threading.Thread.__init__ (self)
        self.win = win  # on mémorise une référence sur la fenêtre
        self.res = res

    def run (self) :
        for i in range (0, 10) :
            print "thread ", i
            time.sleep (0.1)

          # afin que le thread retourne un résultat
          # self.res désigne thread_resultat qui reçoit un nombre de plus
        h = random.randint (0,100)
        self.res.append (h)

          # on lance un événement <<thread_fini>> à la fenêtre principale
          # pour lui dire que le thread est fini, l'événement est ensuite
          # géré par la boucle principale de messages
          # on peut transmettre également le résultat lors de l'envoi du message
          # en utilisant un attribut de la classe Event pour son propre compte
        self.win.event_generate ("<<thread_fini>>", x = h)

thread_resultat = []

def lance_thread () :
    global thread_resultat
      # fonction appelée lors de la pression du bouton
      # on change la légnde de la zone de texte
    text .config (text = "thread démarré")
    text2.config (text = "thread démarré")
      # on désactive le bouton pour éviter de lancer deux threads en même temps
    bouton.config (state = TK.DISABLED)
      # on lance le thread
    m = MonThread (root, thread_resultat)
    m.start ()

def thread_fini_fonction (e) :
    global thread_resultat
      # fonction appelée lorsque le thread est fini
    print "la fenêtre sait que le thread est fini"
      # on change la légende de la zone de texte
    text .config (text = "thread fini + résultat " + str (thread_resultat))
    text2.config (text = "thread fini + résultat (e.x) " + str (e.x))
      # on réactive le bouton de façon à pouvoir lancer un autre thread
    bouton.config (state = TK.NORMAL)
```

2. Les symboles << et >> au début et à la fin du nom de l'événement sont la seule contrainte.

```
import Tkinter as TK

# on crée la fenêtre
root   = TK.Tk ()
bouton = TK.Button (root, text = "thread départ", command = lance_thread)
text   = TK.Label (root, text = "rien")
text2  = TK.Label (root, text = "rien")
bouton.pack ()
text.pack ()
text2.pack ()

# on associe une fonction à un événement <<thread_fini>> propre au programme
root.bind ("<<thread_fini>>", thread_fini_fonction)

# on active la boucle principale de message
root.mainloop ()
```

La figure 9.1 contient la fenêtre affichée par le programme lorsqu'elle attend la pression du bouton qui lance le thread et lorsqu'elle attend la fin de l'exécution de ce thread.

**Remarque 9.4 : méthode `event_generate`**
Le programme précédent utilise une astuce pour retourner un résultat autrement qu'un utilisant un paramètre global. On peut adjoindre lors de l'appel à la méthode `event_generate` quelques informations supplémentaires attachées à l'événement en utilisant les attributs prédéfinis de la classe `Event`. Dans cet exemple, on utilise l'attribut `x` pour retourner le dernier entier tiré aléatoirement.

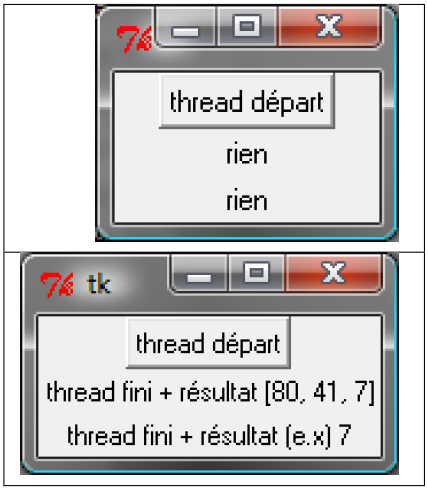



**Figure 9.1 :** *La première image est la fenêtre après trois exécutions du thread. La liste* `thread_resultat` *contient trois nombres. Une fois l'unique bouton pressé, la fenêtre change d'aspect pour devenir comme la seconde image. Cette transition est assurée par la fonction* `lance_thread` *reliée au bouton. La transition inverse est assurée par la fonction* `thread_fini_fonction` *qui est reliée à l'événement que génère le thread lorsqu'il a terminé.*

## 9.4 Files de messages

Les trois usages principaux des threads sont le calcul distribué, la conception d'une interface graphique réactive et l'attente permanente d'événements. En ce qui concernent les deux premiers usages, on peut considérer qu'il existe un thread prin-

cipal qui lance et attend l'exécution de threads secondaires. Les échanges d'informations ont lieu au début et à la fin de chaque thread. Il n'est pas toujours nécessaire de partager des variables en cours d'exécution : l'usage de verrous est peu fréquent pour ces deux schémas sauf pour partager des informations en cours d'exécution.

En ce qui concerne le troisième usage, c'est un cas où tout au long du programme, il y a constamment au moins deux threads actifs : un thread principal et un thread qui attend. Dans ce cas, l'échange et la synchronisation d'informations est inévitable et il est souvent fastidieux de concevoir la synchronisation. C'est pourquoi on la conçoit presque toujours sous forme de messages que les threads s'envoient.

Il existe un objet `Queue` du module `Queue` qui facilite cet aspect. C'est une liste qui possède son propre verrou de sorte que n'importe quel thread peut ajouter ou retirer des éléments de cette liste. Elle est utilisée principalement via quatre méthodes. Deux méthodes `get` sont utilisées au sein du thread qui possède la pile. Deux méthodes `put` sont appelées par des threads étrangers.

| | |
|---|---|
| `get`([`timeout` = ...]) | Retourne un élément de la liste ou attend qu'il y en ait un, le supprime si elle en trouve un. Si `timeout` est renseigné, la fonction attend au plus `timeout` secondes, sinon, elle déclenche l'exception `Queue.Empty`. |
| `get_nowait`() | Retourne un élément de la liste s'il y a en un, dans ce cas, cet élément est supprimé. Dans le cas contraire, la méthode déclenche l'exception `Queue.Empty`. |
| `put`(`e`[, `timeout` = ...]) | Ajoute l'élément `e` à la liste ou attend qu'une place se libère si la liste est pleine. Si `timeout` est renseigné, la fonction attend au plus `timeout` secondes, sinon, elle déclenche l'exception `Queue.Full`. |
| `put_nowait`(`e`) | Ajoute l'élément `e` à la liste ou déclenche l'exception `Queue.Full` si la liste est pleine. |
| `qsize`() | Retourne la taille de la pile. |

Cette pile est utilisée dans l'exemple qui suit pour simuler deux joueurs qui essaye de découvrir le nombre que l'autre joueur a tiré au hasard. A chaque essai, un joueur envoie un message de type ("`essai`", `n`) à l'autre joueur pour dire qu'il joue le nombre `n`. Ce joueur lui répond avec des messages de type ("`dessous`", `n`), ("`dessus`", `n`), ("`gagne`", `n`).

```
# coding: latin-1
import threading, time, Queue, random

class Joueur (threading.Thread) :

    # initialisation
    def __init__ (self, nom, e, nb = 1000, temps = 0.1) :
        threading.Thread.__init__(self)
        self.nb    = nb
        self.queue = Queue.Queue ()
        self.nom   = nom
        self.event = e
        self.temps = temps  # temps de réflexion
    def Joueur (self, autre_joueur) : self.autre = autre_joueur
```

```
    # méthodes : l'adversaire m'envoie un message
    def Joue    (self, nombre) : self.queue.put_nowait ( ("essai", nombre) )
    def Dessus  (self, nombre) : self.queue.put_nowait ( ("dessus", nombre) )
    def Dessous (self, nombre) : self.queue.put_nowait ( ("dessous", nombre) )
    def Gagne   (self, nombre) :
        while not self.queue.empty () :
            try :self.queue.get ()
            except : pass
        self.queue.put ( ("gagne", nombre) )

    # je joue
    def run (self) :
        x = random.randint (0,self.nb)
        print self.nom, " : je joue (", x, ")"
        i = 0
        a = 0
        b = self.nb
        while True :
            time.sleep (self.temps)

            try :
                m,n = self.queue.get_nowait ()       # désynchronisé
                #m,n = self.queue.get (timeout = 0.5)# l'un après l'autre
            except Queue.Empty : m,n = None,None

            # traitement du message --> réponse à l'adversaire
            if m == "essai" :
                if n == x :
                    self.autre.Gagne (n)
                    print self.nom, " : j'ai perdu après ", i, " essais"
                    break
                elif n < x : self.autre.Dessus  (n)
                else       : self.autre.Dessous (n)
            elif m == "dessus" :
                a = max (a, n+1)
                continue  # assure l'équité en mode l'un après l'autre
            elif m == "dessous" :
                b = min (b, n-1)
                continue  # assure l'équité en mode l'un après l'autre
            elif m == "gagne" :
                print self.nom, " : j'ai gagné en ", i, " essais, solution ", n
                break

            # on fait une tentative
            if a == b : n = a
            else : n = random.randint (a,b)
            self.autre.Joue (n)
            i += 1
            print self.nom, " : je tente ", n, " écart ", b-a, \
                  " à traiter ", self.queue.qsize ()

        # fini
        print self.nom, " : j'arrête"
        self.event.set ()

# on crée des verrous pour attendre la fin de la partie
e1 = threading.Event ()
e2 = threading.Event ()
e1.clear ()
```

```
e2.clear ()

# création des joueurs
A = Joueur ("A", e1, 1000, temps = 0.1)
B = Joueur ("B", e2, 1000, temps = 0.3)

# chaque joueur sait qui est l'autre
A.Joueur (B)
B.Joueur (A)

# le jeu commence
A.start ()
B.start ()

# on attend la fin de la partie
e1.wait ()
e2.wait ()
```

Si la méthode `get` est choisie, les joueurs doivent attendre une tentative de l'adversaire avant de proposer la leur. Dans l'autre cas, la méthode `get_nowait` permet de ne pas attendre sa réponse et d'envoyer plusieurs propositions à l'adversaire qui ne répondra pas plus vite pour autant. Dans cette configuration, le joueur `A` est trois fois plus réactif ce qui explique les résultats qui suivent.

```
A  : je joue ( 8 )
B  : je joue ( 569 )
A  : je tente  42  écart  1000  à traiter  0
A  : je tente  791  écart  1000  à traiter  0
...
A  : je tente  528  écart  62  à traiter  0
B  : je tente  20  écart  43  à traiter  57
A  : je tente  508  écart  62  à traiter  0
A  : je tente  548  écart  62  à traiter  0
B  : je tente  8  écart  43  à traiter  59
A  : j'ai perdu après  67  essais
A  : j'arrête
B  : j'ai gagné en  23  essais, solution  8
B  : j'arrête
```

Les affichages se chevauchent parfois, il faudrait pour éviter cela synchroniser l'affichage à l'aide d'un verrou.

Deuxième partie

# ENONCÉS PRATIQUES, EXERCICES

Cette partie regroupe quelques énoncés que les élèves de l'ENSAE ont étudié ou à partir desquels ils ont été évalués entre les années 2005 et 2008. On retient souvent que le boulot d'un informaticien n'est pas compliqué, qu'il est plutôt plutôt ennuyeux et redondant. On oublie parfois que sans l'informatique, certaines tâches seraient tout autant longues et fastidieuses. Cette partie présente certaines tournures d'esprits qui permettent de gagner du temps lors de la conception de programmes et peut-être derendre la programmation moins répétitive, le travail moins fastidieux.

Les premiers énoncés illustrent quelques schémas récurrents de programmation. Les langages de programmation qui nécessitent de reprogrammer un tri sont très rares, la plupart proposent des fonctions qui exécutent un tri rapide - ou quicksort -. Mais connaître certains algorithmes classiques permet de programmer plus rapidement qu'avec la seule connaissance des principes de base - boucle, test -. A l'instar de certains jeux de réflexion - les ouvertures avec les échecs -, il n'est pas nécessaire de réinventer certains algorithmes ou raisonnements fréquemment utilisés.

Cette partie se termine par trois années d'énoncés d'examens d'informatique. Les exercices proposés illustrent parfois le côté ennuyeux d'un programme informatique : sa mise en œuvre dépasse souvent de loin sa conception. La correction des erreurs ou *bugs* est la partie la plus longue et la plus rébarbative. Ces exercices ont pour objectif d'accélérer cette étape en proposant des situations qui reviennent fréquemment. C'est aussi pour cela que lors d'entretiens d'embauche, des exercices similaires sont présentés aux candidats pour vérifier leur expérience en programmation.

# Chapitre 10

# Exercices pratiques pour s'entraîner

---

## 10.1 Montant numérique, montant littéral

L'objectif de cet exercice est d'écrire un programme qui convertit en entier un nombre écrit sous forme littérale. On suppose que tous les nombres sont écrits en minuscules et correctement orthographiés. Cet exercice tourne autour des manipulations de chaînes de caractères et principalement les deux fonctions suivantes :

| **Fonctions utiles :** | |
|---|---|
| `replace(s, old, new)` | A l'intérieur de la chaîne `s`, remplace toutes les occurrences de la chaîne `old` par la chaîne `new`. Pour l'utiliser, il faut ajouter au début du programme la ligne `import string` puis écrire `l = string . replace ("dix - neuf", " - ", " ")`, par exemple. |
| `split(s[, sep])` | Retourne une liste contenant tous les mots de la chaîne de caractères `s`. Si le paramètre facultatif `sep` est renseigné, c'est cette chaîne qui fait office de séparateur entre mot et non les espaces. Pour l'utiliser, il faut ajouter au début du programme la ligne `import string` puis écrire `l = string . split ("vingt et un", " ")`, par exemple. |

L'appel à la fonction `replace` peut se faire en la considérant comme une méthode de la classe `str` ou une fonction du module `string`.

```
import string
s = "dix-neuf"

l = s.replace ("-", " ")
l = string.replace ("-", " ")
```

**1)** Ecrire une fonction qui prend un nombre littéral et retourne une liste contenant tous ses mots.

**2)** Ecrire une fonction qui reçoit une chaîne de caractères correspondant à un nombre compris entre 0 et 16 inclus ou à un nombre parmi 20, 30, 40, 50, 60. La fonction doit retourner un entier correspondant à sa valeur.

**3)** A l'aide de la fonction précédente, écrire une fonction qui reçoit une chaîne de caractères contenant un nombre compris entre 0 et 99 inclus et qui retourne sa valeur.

**4)** Pour vérifier que la fonction précédente marche bien dans tous les cas, écrire une fonction qui écrit un nombre sous sa forme littérale puis vérifier que ces deux fonctions marchent correctement pour les nombres compris entre 0 et 99.

### Correction

A chaque question correspond une fonction utilisant celles décrites dans les questions précédentes. Le programme complet est obtenu en juxtaposant chaque morceau.

**1)**

```
# coding: latin-1
def lire_separation(s):
    """divise un nombre littéral en mots"""
    s = s.replace ("-", " ")     # on remplace les tirets par des espaces
                                 # pour découper en mots même
                                 # les mots composés
    return s.split ()
```

On pourrait écrire cette fonction en utilisant les expressions régulières en utilisant les expressions régulières (voir `split` décrite au paragraphe 7.6.3). page 188.

```
import re
def lire_separation(s):
    """divise un nombre littéral en mots avec les expressions régulières"""
    return re.compile ("[- ]").split (s)
```

**2)** Il n'existe pas qu'une seule manière de rédiger cette fonction. La première, la plus simple, est sans doute la suivante :

```
def valeur_mot (s) :
    """convertit numériquement les nombres inclus entre 0 et 16 inclus,
    20, 30, 40, 50, 60, s est une chaîne de caractères, le résultat est entier"""
    if   s == "zéro"      : return 0
    elif s == "un"        : return 1
    elif s == "deux"      : return 2
    elif s == "trois"     : return 3
    elif s == "quatre"    : return 4
    elif s == "cinq"      : return 5
    elif s == "six"       : return 6
    elif s == "sept"      : return 7
    elif s == "huit"      : return 8
    elif s == "neuf"      : return 9
    elif s == "dix"       : return 10
    elif s == "onze"      : return 11
    elif s == "douze"     : return 12
    elif s == "treize"    : return 13
    elif s == "quatorze"  : return 14
    elif s == "quinze"    : return 15
    elif s == "seize"     : return 16
    elif s == "vingt"     : return 20
    elif s == "trente"    : return 30
    elif s == "quarante"  : return 40
    elif s == "cinquante" : return 50
    elif s == "soixante"  : return 60
    else                  : return 0      # ce cas ne doit normalement pas
                                          # se produire
```

La solution suivante utilise un dictionnaire. Cette écriture peut éviter un copier-coller long pour la dernière question.

```
def valeur_mot (s) :
    dico = {'cinquante': 50, 'quarante': 40, 'onze': 11, 'huit': 8, 'six': 6, \
            'quinze': 15, 'trente': 30, 'douze': 12, 'cinq': 5, 'deux': 2, \
            'quatorze': 14, 'neuf': 9, 'soixante': 60, 'quatre': 4, \
            'zéro': 0, 'treize': 13, 'trois': 3, 'seize': 16, \
            'vingt': 20, 'un': 1, 'dix': 10, 'sept': 7}
    if s not in dico : return 0   # cas imprévu, on peut résumer ces deux lignes
    else : return dico [s]        # par return dico.get (s, 0)
```

**3)** La réponse à cette question est divisée en deux fonctions. La première traduit une liste de mots en un nombre numérique. Ce nombre est la somme des valeurs retournées par la fonction `valeur_mot` pour chacun des mots : soixante dix neuf = $60 + 10 + 9$. Cette règle s'applique la plupart du temps sauf pour *quatre vingt*.

```
def lire_dizaine_liste(s):
    """convertit une liste de chaînes de caractères dont
    juxtaposition forme un nombre littéral compris entre 0 et 99"""
    r       = 0         # contient le résultat final
    dizaine = False     # a-t-on terminé le traitement des dizaines ?
    for mot in s:
        n = lire_unite (mot)
        if n == 20 :
            if not dizaine and r > 0 and r != 60 :
                r       *= n     # cas 80
                dizaine  = True
            else : r += n
        else : r += n
    return r
```

La seconde fonction remplace tous les traits d'union par des espaces puis divise un nombre littéral en liste de mots séparés par des espaces.

```
def lire_dizaine(s):
    li = lire_separation (s)
    return lire_dizaine_liste (li)
```

**4)** Tout d'abord, les deux fonctions réciproques de celles présentées ci-dessus.

```
def ecrit_unite (x):
    """convertit un nombre compris inclus entre 0 et 16 inclus,
    20, 30, 40, 50, 60 en une chaîne de caractères"""
    if   x == 0: return "zéro"
    elif x == 1: return "un"
    elif x == 2: return "deux"
    elif x == 3: return "trois"
    elif x == 4: return "quatre"
    elif x == 5: return "cinq"
    elif x == 6: return "six"
    elif x == 7: return "sept"
    elif x == 8: return "huit"
    elif x == 9: return "neuf"
    elif x == 10: return "dix"
    elif x == 11: return "onze"
```

```
    elif x == 12: return "douze"
    elif x == 13: return "treize"
    elif x == 14: return "quatorze"
    elif x == 15: return "quinze"
    elif x == 16: return "seize"
    elif x == 20: return "vingt"
    elif x == 30: return "trente"
    elif x == 40: return "quarante"
    elif x == 50: return "cinquante"
    elif x == 60: return "soixante"
    elif x == 70: return "soixante-dix"
    elif x == 80: return "quatre-vingt"
    elif x == 90: return "quatre-vingt-dix"
    else        : return "zéro"
```

Cette fonction pourrait être simplifiée en utilisant le même dictionnaire que pour la fonction `valeur_mot` mais inversé : la valeur devient la clé et réciproquement.

```
def mot_valeur (x):
    """convertit un nombre compris inclus entre 0 et 16 inclus,
    20, 30, 40, 50, 60 en une chaîne de caractères"""
    dico = {'cinquante': 50, 'quarante': 40, 'onze': 11, 'huit': 8, 'six': 6, \
            'quinze': 15, 'trente': 30, 'douze': 12, 'cinq': 5, 'deux': 2, \
            'quatorze': 14, 'neuf': 9, 'soixante': 60, 'quatre': 4, \
            'zéro': 0, 'treize': 13, 'trois': 3, 'seize': 16, \
            'vingt': 20, 'un': 1, 'dix': 10, 'sept': 7}
    inv = {}
    for k,v in dico.iteritems () : inv [v] = k
    inv [70] = "soixante-dix"
    inv [80] = "quatre-vingt"
    inv [90] = "quatre-vingt-dix"
    return inv [x]
```

Le programme continue avec l'une ou l'autre des fonctions précédente. La fonction suivante construit le montant littéral pour un nombre à deux chiffres. Elle traite le chiffre des dizaines puis des unités.

```
def ecrit_dizaine(x):
    """convertit un nombre entre 0 et 99 sous sa forme littérale"""

    if x <= 16 : return ecrit_unite(x)

    s       = ""
    dizaine = x / 10
    unite   = x % 10
    s = mot_valeur (dizaine*10)
    s += " "
    s += mot_valeur (unite)
    return s
```

Le morceau de code ci-dessous permet de vérifier que le passage d'un nombre sous sa forme littérale et réciproquement fonctionne correctement pour tous les nombres compris entre 0 et 99 inclus.

```
for i in xrange(0,100):
    s = ecrit_dizaine (i)
```

```
    j = lire_dizaine (s)
    if i != j : print "erreur ", i, " != ", j, " : ", s
```

## 10.2 Représentation des données, partie de dames

Une partie de dames met en jeu quarante pions, vingt noirs, vingt blancs, chacun sur des cases différentes. L'objectif est de savoir si un pion est en mesure d'en prendre un autre. On ne traitera pas le cas des dames. Chaque pion est défini par :

- deux coordonnées entières, chacune comprise entre 1 et 10
- une couleur, noir ou blanc

**1)** On propose deux représentations de l'ensemble de pions :

1. Un tableau de 40 pions indicés de 0 à 39 inclus, chaque pion étant défini par :
   - deux coordonnées comprises entre 1 et 10, ou (0,0) si le pion n'est plus sur le damier
   - un entier qui vaut 1 pour blanc, 2 pour noir
2. Un tableau d'entiers à deux dimensions, chaque case contient :
   - soit 0 s'il n'y a pas de pion
   - soit 1 si la case contient un pion blanc
   - soit 2 si la case contient un pion noir

Y a-t-il d'autres représentations de ces informations ? Si on considère que l'efficacité d'une méthode est reliée à sa vitesse - autrement dit aux coûts des algorithmes qu'elles utilisent -, parmi ces deux représentations, quelle est celle qui semble la plus efficace pour savoir si un pion donné du damier est en mesure d'en prendre un autre ?

**2)** Comment représenter un tableau d'entiers à deux dimensions en langage *Python* à l'aide des types standards qu'il propose, à savoir t-uple, liste ou dictionnaire ?

**3)** On cherche à écrire l'algorithme qui permet de savoir si un pion donné est un mesure de prendre un pion. Quels sont les paramètres d'entrées et les résultats de cet algorithme ?

**4)** Il ne reste plus qu'à écrire cet algorithme.

### Correction

**1)** La seconde représentation sous forme de tableau à deux dimensions est plus pratique pour effectuer les tests de voisinages. Chaque case a quatre voisines aux quatre coins, il est ensuite facile de déterminer si ces quatre voisines sont libres ou si elles contiennent un pion. On sait rapidement le contenu d'une case.

Avec la première représentation - le tableau des pions - pour savoir s'il existe un pion dans une case voisine, il faut passer en revue tous les pions pour savoir si l'un d'eux occupe ou non cette case. Avec la seconde représentation - le tableau à deux dimensions - on accède directement à cette information sans avoir à la rechercher. On évite une boucle sur les pions avec la seconde représentation.

**2)** Pour représenter le tableau en deux dimensions, il existe trois solutions :

1. Une liste de listes, chaque ligne est représentée par une liste. Toutes ces listes sont elles-mêmes assemblées dans une liste globale.

2. Une seule liste, il suffit de numéroter les cases du damier de 0 à 99, en utilisant comme indice pour la case $(i, j)$ : $k = 10 * i + j$. Réciproquement, la case d'indice $k$ aura pour coordonnées $(k/10,\ k\%10)$.
3. Un dictionnaire dont la clé est un couple d'entiers.

**3)** On désire savoir si le pion de la case $(i, j)$ peut en prendre un autre. On suppose que le tableau à deux dimensions est une liste de dix listes appelée `damier`. `damier[i][j]` est donc la couleur du pion de la case $(i, j)$, à savoir 0 si la case est vide, 1 si le pion est blanc, 2 si le pion est noir. Pour ces deux derniers cas, la couleur des pions de l'adversaire sera donc $3 - $ `damier[i][j]`. Les entrées de la fonctions sont donc les indices `i`, `j` et le damier `damier`. La sortie est une variable booléenne qui indique la possibilité ou non de prendre. On ne souhaite pas déplacer les pions.

**4)**

```
def pion_prendre(i,j,damier):
   c = damier [i][j]
   if c == 0: return False  # case vide, impossible de prendre
   c = 3 - c                # couleur de l'adversaire

   if damier [i-1][j-1] == c :     # s'il y a un pion adverse en haut à gauche
       if damier [i-2][j-2] == 0 : # si la case d'après en diagonale est vide
           return True             # on peut prendre

   # on répète ce test pour les trois autres cases
   if damier [i-1][j+1] == c and damier [i-2][j+2] == 0: return True
   if damier [i+1][j-1] == c and damier [i+2][j-2] == 0: return True
   if damier [i+1][j+1] == c and damier [i+2][j+2] == 0: return True

   # si tous les tests ont échoué, on ne peut pas prendre
   return False
```

Voici une fonction équivalente lorsque le damier est un dictionnaire dont la clé est un couple d'entiers.

```
def pion_prendre(i,j,damier):
   c = damier [(i,j)]       # ou encore damier [i,j]
   if c == 0: return False  # case vide, impossible de prendre
   c = 3 - c                # couleur de l'adversaire

   # test pour une prise du pion dans les quatre cases voisines
   if damier [i-1,j-1] == c and damier [i-2,j-2] == 0: return True
   if damier [i-1,j+1] == c and damier [i-2,j+2] == 0: return True
   if damier [i+1,j-1] == c and damier [i+2,j-2] == 0: return True
   if damier [i+1,j+1] == c and damier [i+2,j+2] == 0: return True

   # si tous les tests ont échoué, on ne peut pas prendre
   return False
```

La même fonction lorsque le damier est représenté par une seule liste.

```
def pion_prendre(i,j,damier):
   c = damier [10*i+j]
   if c == 0: return False  # case vide, impossible de prendre
   c = 3 - c                # couleur de l'adversaire

   # test pour une prise du pion dans les quatre cases voisines
```

```
   if damier [10*(i-1)+j-1] == c and damier [10*(i-2)+j-2] == 0: return True
   if damier [10*(i-1)+j+1] == c and damier [10*(i-2)+j+2] == 0: return True
   if damier [10*(i+1)+j-1] == c and damier [10*(i+2)+j-2] == 0: return True
   if damier [10*(i+1)+j+1] == c and damier [10*(i+2)+j+2] == 0: return True

   return False
```

Pour ces trois cas, aucun effet de bord n'a été envisagé. Si la case est trop près d'un des bords, un des indices $i$, $j$, $i-1$, $j-1$, $i+1$, $j+1$, $i-2$, $j-2$, $i+2$, $j+2$ désignera une case hors du damier. Le code de la fonction `pion_prendre` devra donc vérifier que chaque case dont elle vérifie le contenu appartient au damier.

```
def pion_prendre(i,j,damier):
   c = damier [i][j]
   if c == 0: return False  # case vide, impossible de prendre
   c = 3 - c                # couleur de l'adversaire

   # on répète ce test pour les trois autres cases
   if i >= 2                 and j >= 2                  and \
      damier [i-1][j-1] == c and damier [i-2][j-2] == 0: return True
   if i >= 2                 and j < len (damier)-2      and \
      damier [i-1][j+1] == c and damier [i-2][j+2] == 0: return True

   if i < len (damier)-2     and j >= 2                  and \
      damier [i+1][j-1] == c and damier [i+2][j-2] == 0: return True
   if i < len (damier)-2     and j < len (damier)-2      and \
      damier [i+1][j+1] == c and damier [i+2][j+2] == 0: return True

   return False
```

Une autre option consiste à entourer le damier d'un ensemble de cases dont le contenu sera égal à une constante différente de `0`, `-1` par exemple. Dans ce cas, si le damier est représenté par une liste de listes, la première case du damier aura pour coordonnées $(1,1)$ au lieu de $(0,0)$ car les listes n'acceptent pas les indices négatifs. Ce n'est pas le cas lorsque le damier est représenté par un dictionnaire car une case peut tout à fait avoir pour coordonnées $(-1,-1)$.

Avec cette convention, les tests introduits dans le dernier programme ne seront plus nécessaires. Il faudra juste réécrire la seconde ligne de chaque fonction `pion_prendre` par :

```
    if c <= 0 : return False  # au lieu de if c == 0 : return False
```

**Remarque 10.1 : matrice et dictionnaire**
Les structures représentées ici sont des tableaux à deux dimensions : ce sont des matrices. Les dictionnaires sont particulièrement indiqués dans le cas où ces matrices sont *creuses* : elles contiennent beaucoup de valeurs nulles. Pour une liste ou une liste de listes, toutes les cases sont représentées. Dans le cas d'un dictionnaire, il est possible d'adopter comme convention que si un couple de coordonnées n'existe pas en tant que clé, cela veut dire que la case associée est nulle. Il en résulte un gain de place équivalent à la proportion de cases nulles.

**Remarque 10.2 : dictionnaire et clé de type tuple**
Le second exemple utilise un dictionnaire avec comme clé un `tuple`. Dans ce cas, les parenthèses sont superflues.

```
damier [(i-1,j-1)]   # est équivalent à
damier [ i-1,j-1 ]   # cette ligne
```

## 10.3 Reconnaître la langue d'un texte

L'objectif est de distinguer un texte anglais d'un texte français sans avoir à le lire. Le premier réflexe consisterait à chercher la présence de mots typiquement anglais ou français. Cette direction est sans doute un bon choix lorsque le texte considéré est une œuvre littéraire. Mais sur Internet, les contenus mélangent fréquemment les deux langues : la présence de tel mot anglais n'est plus aussi discriminante. Il n'est plus aussi évident d'étiqueter un document de langue anglaise lorsque les mots anglais sont présents partout.

On ne cherche plus à déterminer la langue d'un texte mais plutôt la langue majoritaire. Il serait encore possible de compter les mots de chacune des langues à l'aide d'un dictionnaire réduit de mots anglais et français. La langue majoritaire correspondrait à celle dont les mots sont les plus fréquents. Mais construire un dictionnaire est d'abord fastidieux. Ensuite, il faudrait que celui-ci contienne des mots présents dans la plupart des textes. Il faudrait aussi étudier le problème des mots communs aux deux langues.

Pour ces trois raisons, il paraît préférable d'étudier une direction plus simple quitte à revenir aux dictionnaires plus tard. Dans un premier temps, on cherchera donc à distinguer un texte anglais d'un texte français à partir de la fréquence de certaines lettres dont l'usage est différent dans les deux langues comme la lettre *w*.

La première étape consiste à calculer la fréquence des lettres dans différents textes anglais et français qu'on peut se procurer depuis le site *Gutenberg*[1] par exemple. Il suffit alors de télécharger quelques textes et de les enregistrer dans des fichiers texte.

**1)** Il faut tout d'abord récupérer dans un programme *Python* le contenu de ces textes dont on suppose qu'ils ont été téléchargés. On pourra s'inspirer des exemples du chapitre 7.

**2)** On veut construire une fonction qui compte l'occurrence de chaque lettre dans une chaîne de caractères. Cette fonction prend en entrée une chaîne de caractères et retourne un dictionnaire. Chaque élément contiendra le nombre de fois qu'un caractère apparaît.

**3)** Afin d'obtenir des probabilités, on divise chaque nombre par le nombre total de lettres (attention aux divisions entières).

**4)** Ecrire une fonction qui retourne la probabilité de chaque lettre dans un fichier texte.

**5)** En observant les résultats obtenus sur un texte français et un texte anglais, pensez-vous qu'il soit possible d'écrire une fonction qui détermine automatiquement la langue d'un texte à condition que celle-ci soit l'anglais ou le français.

### Correction

1. http://www.gutenberg.net/

**1)** La fonction répondant à la première question revient régulièrement dans beaucoup de programmes.

```
def lit_fichier (nom) :
    f = open (nom, "r")                  # ouverture du fichier
    l = f.read ()                        # on récupère le contenu
    f.close ()                           # on ferme le fichier
    return l                             # on retourne le contenu
```

On pourrait également souhaiter récupérer un texte directement depuis Internet à condition d'en connaître l'adresse, ce que fait la fonction suivante. La première fonction doit recevoir un nom de fichier, la seconde une adresse Internet.

```
import urllib                        # import du module urllib
def lit_url (nom) :
    f = urllib.urlopen (nom)         # on ouvre l'url
    res = f.read ()                  # on lit son contenu
    f.close ()                       # on termine la lecture
    return res                       # on retourne le résultat
```

On peut regrouper ces deux fonctions et appeler soit l'une soit l'autre selon que le nom du texte est un fichier texte ou une adresse Internet :

```
def lit (texte) :
    if texte.startswith ("http") : s = lit_url (texte)  # Internet
    else : s = lit_fichier (texte)                      # fichier texte
    return s

s = lit ("hugo.txt")
```

Une autre option consiste à utiliser les exceptions : on essaye d'abord d'ouvrir un fichier avec la fonction `open`. Si cela ne fonctionne pas, on peut supposer que le nom du fichier fait référence à une adresse Internet.

```
def lit (texte) :
    try :
        s = lit_fichier (texte)          # fichier texte
        return s
    except :                             # si cela ne marche pas,
        s = lit_url (texte)              # on suppose que texte
        return s                         # est une adresse Internet

s = lit ("hugo.txt")
```

**2)** La fonction suivante compte les occurrences de chaque lettre dans la chaîne de caractères `texte`. Une première solution consiste à appeler 26 fois la méthode `count` des chaînes de caractères.

```
def compte_lettre_count (texte) :
    texte = texte.upper ()           # pour éviter les accents
    res = { }                        # résultat, vide pour le moment
    for c in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" :  # pour tout l'alphabet
        res [c] = texte.count (c)    # on compte les occurrences de c
    return res
```

Une autre solution consiste à parcourir le texte puis à compter les lettres au fur et à mesure qu'elles apparaissent.

```
def compte_lettre (texte) :
    texte = texte.upper ()              # pour éviter les accents
    res = { }                           # résultat, vide pour le moment
    for c in texte :                    # pour tous les caractères du texte
        if not("A" <= c <= "Z") : continue # si ce n'est pas une lettre, on passe
        if c not in res : res [c] = 1   # si elle n'est pas là, on lui affecte 1
        else : res [c] += 1        # sinon, on augmente son nombre d'apparitions
    return res
```

Il n'est pas évident de choisir l'une ou l'autre des méthodes. Calculer le coût algorithmique de chacune des fonctions n'est pas toujours évident car la première utilise la fonction `count` dont le fonctionnement est inconnu. Il paraît plus facile dans ce cas de comparer la vitesse d'exécution de chaque fonction.

Dans l'exemple suivant, la fonction `comparaison` appelle les deux méthodes sur le même texte. Le module `profile` va ensuite mesurer le temps passé dans chacune des fonctions.

```
def comparaison () :
    s = lit_fichier ("hugo.txt")
    compte_lettre_count (s)      # on ne mémorise pas les résultats
    compte_lettre (s)            # car on souhaite mesurer le temps passé

import profile                   # import du module profile
profile.run ("comparaison()")    # mesure du temps passé dans la fonction
                                 # comparaison et les fonctions qu'elle appelle
```

A la fin du programme, le temps passé dans chaque fonction est affiché par le module `profile`. On peut y lire le nombre de fois qu'une fonction a été appelée, le temps moyen passé dans cette fonction.

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.160    0.160    0.174    0.174 langue.py:19(compte_lettre)
        1    0.000    0.000    0.017    0.017 langue.py:31(compte_lettre_count)
        1    0.000    0.000    0.190    0.190 langue.py:43(comparaison)
```

| | |
|---|---|
| ncalls | Nombre d'appels à la fonction |
| tottime | Temps passé uniquement dans la fonction, le temps passé dans une fonction appelée par celle-ci n'est pas compté. |
| percall | Temps moyen passé uniquement dans la fonction. |
| cumtime | Temps cumulé passé dans la fonction, y compris celui passé dans une fonction appelée. |
| percall | Temps cumulé moyen. |
| filename | nom du fichier et nom de la fonction entre parenthèse |

La première solution est clairement la plus rapide. Ce résultat s'explique par la rapidité de la méthode `count`. Mais cela ne permet de pas de conclure dans tous les cas. Il faudrait connaître le coût algorithmique de chaque fonction pour cela. Par exemple, lorsqu'il s'agit de calculer les fréquences de tous les caractères et non plus des seules 26 lettres, on s'aperçoit que la seconde solution n'est plus que 1,5 fois plus lente et non 10 fois.

```
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.120    0.120    0.133    0.133 langue.py:19(compte_lettre)
       1    0.002    0.002    0.082    0.082 langue.py:32(compte_lettre_count)
```

Lorsque le nombre d'éléments dont il faut calculer la fréquence est assez grand, la seconde solution est préférable. La première solution est efficace dans des cas particuliers comme celui-ci. On s'aperçoit également que la fonction `compte_lettre` est plus rapide pour cette seconde expérience ($0.133 < 0.174$) mais cette observation est peu pertinente car cette fonction n'a été exécutée qu'une fois. Il y a en effet peu de chance pour que la fonction s'exécute à chaque fois à la même vitesse car l'ordinateur exécute un grand nombre de tâches simultanément.

Pour obtenir un temps d'exécution fiable, il faut appeler la fonction un grand nombre de fois. Le tableau suivant montre le temps moyen total passé dans chaque fonction après 100 appels à la fonction `comparaison`.

| **fonction** | **26 lettres** | **tous les caractères** |
|---|---|---|
| `compte_lettre` | 0.087 | 0.061 |
| `compte_lettre_count` | 0.017 | 0.073 |

La fonction `compte_lettre` est effectivement plus rapide dans le second cas. Comme on s'intéresse maintenant à tous les caractères, le test `if not` (`"A" <= c <= "Z"`) n'est plus nécessaire : la fonction est d'autant plus rapide que cette instruction était exécutée un grand nombre de fois. On remarque également que la fonction `compte_lettre` ne paraît pas dépendre du nombre de caractères considérés. La fonction `compte_lettre` est un meilleur choix.

**3)** Il suffit de diviser chaque valeur du dictionnaire `res` par la somme de ses valeurs. On insère donc les trois lignes suivantes juste avant l'instruction `return` de la fonction précédente.

```
def compte_lettre (texte) :
    # ...
    s = sum (res.values ())
    for k in res :
        res [k] = float (res [k]) / s
    return res
```

**4)** Le tableau suivant montre les probabilités obtenues pour les quatre lettres H,U,W,Y pour lesquelles les résultats montrent des différences significatives. Les deux textes sélectionnés sont *Le dernier jour d'un condamné* de Victor Hugo et sa traduction anglaise *The Man Who Laughs*.

| **lettre** | Le dernier jour d'un condamné | The Man Who Laughs |
|---|---|---|
| H | 1,22 % | 6,90 % |
| U | 6,79 % | 2,57 % |
| W | 0,11 % | 2,46 % |
| Y | 0,42 % | 1,34 % |

Ces chiffres ont été obtenus avec le programme suivant qui télécharge ces deux textes directement depuis le site *Gutenberg*[2]. Les fréquences de ces quatre lettres montrent des différences significatives, ce que montre de façon plus visuelle la figure 10.1 pour les lettres `W` et `H`.

2. Il est conseillé de vérifier ces adresses depuis le site http://www.gutenberg.net/, ces liens peuvent changer.

```
# le dernier jour d'un condamné
c1 = compte_lettre (lit_url ("http://www.gutenberg.org/dirs/etext04/8ldrj10.txt"))
# the man who laughs
c2 = compte_lettre (lit_url ("http://www.gutenberg.org/files/12587/12587-8.txt"))

car = c1.keys ()
car.sort ()
for k in car :
    print k, " : ", "% 2.2f" % (c1 [k] * 100), "%", " % 2.2f" % (c2 [k] * 100), "%"
```

**5)** La fréquence de la lettre `W` ou `H` devrait suffire à départager un texte français d'un texte anglais. Pour vérifier cette hypothèse, on décrit chaque texte par deux coordonnées correspondant aux fréquences des lettres `H` et `W`, ce que calcule la fonction suivante :

```
def langue_lettre (texte) :
    if "http" in texte : s = lit_url (texte)   # cas URL
    else : s = lit_fichier (texte)             # cas fichier
    c = compte_lettre (s)                      # on compte les lettres
    return c ["W"], c ["H"]                    # on retourne deux fréquences
```

La dernière instruction de cette fonction suppose que tous les textes incluent au moins un `W` et un `H`. Dans le cas contraire, la fonction produirait une erreur car elle ne pourrait pas trouver la clé "`W`" ou "`H`" dans le dictionnaire `c`. Pour remédier à cela, on utilise la méthode `get` des dictionnaires qui permet de retourner une valeur lorsqu'une clé est introuvable.

```
def langue_lettre (texte) :
    ...                                        # lignes inchangées
    return c.get ("W", 0.0), c.get ("H", 0.0)  # on retourne deux fréquences
```

Après cette correction, la fonction retourne une valeur nulle lorsque les lettres `W` ou `H` n'apparaissent pas. La fonction suivante permet de construire deux listes `cx` et `cy` qui contiennent les fréquences des lettres `W` et `H` pour une liste de textes.

```
def curve (li) :
    cx,cy = [], []
    for l in li :                # pour tous les textes de la liste
        x,y = langue_lettre (l)  # coordonnées d'un texte, fréquence W et H
        cx.append (x)            # on ajoute x à la liste des abscisses
        cy.append (y)            # on ajoute y à la liste des ordonnées
    return cx,cy
```

On utilise cette dernière fonction pour faire apparaître sur un graphique plusieurs textes anglais et français. Chaque texte est représenté par deux coordonnées : la fréquence des lettres `W` et `H`. On obtient un nuage de points dans un plan ce que montre le graphe de la figure 10.1. Ce dernier a été tracé à l'aide du module `matplotlib`[3]. Il s'inspire du logiciel *Matlab* dans la manière de constuire des graphes[4].

3. http://matplotlib.sourceforge.net/

4. Il a l'inconvénient de ne pas supporter les accents français mais il est possible de contourner cet obstacle en associant *MatPlotLib* et *Latex* qui permet également d'afficher des formules ma-

```
frcx,frcy = curve (fr)          # on récupère les coordonnées des textes français
encx,ency = curve (en)          # on récupère les coordonnées des textes anglais

import pylab                            # import du module matplotlib
pylab.plot (frcx, frcy, "rx",ms=10,\
                            mew=2.5) # on trace la courbe des textes français
pylab.plot (encx, ency, "bv")        # on trace la courbe des textes anglais
pylab.legend (("francais", "anglais"), loc=2)  # légende (sans accent)
pylab.title ("langue")               # titre
pylab.xlabel ("frequence de W")      # légende de l'axe des abscisses
pylab.ylabel ("frequence de H")      # légende de l'axe des ordonnées
pylab.savefig ("graphe.png")         # enregistrement sous forme d'image
pylab.show ()                        # on fait apparaître le graphique
```

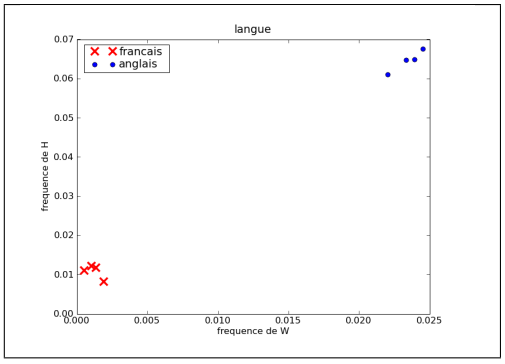



**Figure 10.1 :** *Les textes anglais et français sont cantonnés chacun dans une zone précise du graphe : il apparaît que les fréquences des lettres* `H` *et* `W` *permettent toutes deux de distinguer des textes anglais et français. Une fréquence inférieure à 1% pour la lettre W signifie un texte français.*

La figure 10.1 montre qu'au delà d'une fréquence de 1% pour la lettre `W`, le texte est anglais. On en déduit la fonction qui permet de déterminer la langue d'un texte.

```
def est_anglais (texte) :
    w,h = langue_lettre (texte)
    return w > 0.01
```

La réponse à la dernière question pourrait s'arrêter ici. Toutefois, les paragraphes qui suivent s'intéressent à la fiabilité de la fonction de la fonction `est_anglais` dans un autre contexte : Internet.

### Interprétation

En effet, ce dernier résultat peut être critiqué : ce seuil s'appuie sur seulement quatre textes. Ces derniers sont longs : la fréquence des lettres en est d'autant plus fiable. Sur un texte d'une centaine de lettres, la fréquence de la lettre $W$ est soit nulle, soit supérieure à 1%. Au delà de ce cas difficile et peu fréquent, on vérifie la fiabilité de la fonction `est_anglais` sur des articles de journaux français et anglais.

Internet va nous aider à construire une base de textes : il est facile des récupérer automatiquement de nombreux articles de journaux sur des sites tels que celui du

thématiques sur le graphique. L'utilisation de ce genre de modules est plutôt facile, de nombreux exemples de graphiques sont proposés avec le programme *Python* qui a permis de les réaliser. Il suffit souvent des recopier et de les adapter. Une autre solution consiste à écrire les courbes dans un fichier texte puis à les récupérer avec un tableur tel que *OpenOffice* pour les représenter sous forme de graphe (voir l'exemple page 170).

journal *Le Monde* ou le *New-York Times*. On pourra ensuite calculer la fréquence des lettres `W` et `H` pour chacun d'entre eux.

**Récupération de pages HTML**

Une page Internet contient du texte dont la présentation est décrite par le format HTML. On y trouve tout type de contenu, des images, de la vidéo, du texte, et des hyperliens. Le site du journal *Le Monde*[5] contient beaucoup de liens vers des articles qui eux-mêmes contiennent des liens vers d'autres articles (voir figure 10.2). A partir d'une seule page, on récupére une myriade de liens Internet. L'objectif est ici d'explorer quelques pages pour récupérer des liens vers des articles de ce journal. La récupération des pages Internet d'un site s'inspire de la récupération de la liste de tous les fichiers inclus dans un répertoire[6].

La principale différence provient du fait que les liens ne désignent pas toujours une page nouvelle mais parfois la première page du site ou une autre page déjà vue dont il ne faut pas tenir compte. L'autre différence vient du faite que le nombre de pages sur un site comme *Le Monde* peut s'avérer très grand : il faut bien souvent se limiter aux premières pages et à celles qui appartiennent à ce site.

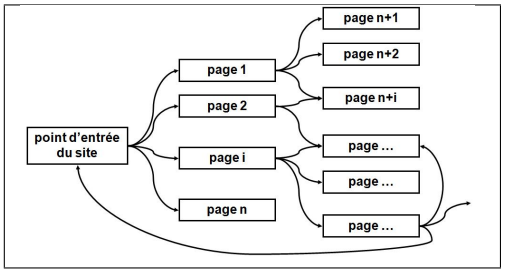



**Figure 10.2 :** *Un site Internet contient généralement une page qui est le moyen d'accéder à toutes les pages du site. C'est un point d'entrée qui mène directement ou indirectement à l'ensemble des pages du site. Il suffit de suivre les liens pour en établir une liste. Certains liens mènent à des pages nouvelles, d'autres à des pages déjà vue : ce graphe est cyclique.*

Le format HTML fonctionne par balises : une information est encadrée par deux balises : `<balise> texte ... </balise>`. Les hyperliens utilisent une syntaxe à peine plus compliquée puisqu'une balise peut aussi recevoir quelques paramètres : `<balise param1="valeur1" param2="valeur2"> texte ... </balise>`. Selon cette syntaxe, un lien vers une autre page est indiqué comme suit :

```
<a href="http://...."> texte qui apparaît à l'écran </a>
```

Les expressions régulières[7] permettent de retrouver aisément tous les liens insérés dans une page sans avoir à se soucier du format HTML. L'algorithme de recherche consiste à recenser tous les liens de la première page. La syntaxe HTML nous apprend que ces liens sont compris entre guillemets. On considère ensuite chacune des pages recensées et on procède de même. On s'arrête après avoir obtenu le nombre de pages souhaité.

```
import urllib       # pour accéder une adresse internet
import re           # pour traiter les expressions régulières
```

5. http://www.lemonde.fr/
6. voir paragraphe 7.3.3 page 177
7. voir paragraphe 7.6 page 184

```
def list_url (site, root, nbpage = 100) :

    # expression régulières
    # tous les liens commençant par root et entre guillemets
    # exemple :  "http://www.lemonde.fr/index,0.26.html"
    # on place entre parenthèses la partie intéressante :
    # c'est-à-dire tout ce qu'il y a entre les guillemets
    s = "\"(" + root + "[-_~a-zA-Z0-9/.?,]*?)\""
    exp  = re.compile (s, re.IGNORECASE)

    res    = [ ]         # résultat
    pile   = [ site ]    # page à explorer

    while len (pile) > 0 and len (res) < nbpage :

        # on bascule toutes les pages de pile vers res
        for u in pile :
            if u not in res : res.append (u)

        u = pile.pop ()  # on s'intéresse à la prochaine page

        try :
            f       = urllib.urlopen (u)      # accès à l'url
            text    = f.read ()               # on lit son contenu
            f.close ()                        # fin de l'accès

            liens = exp.findall (text)        # recherche de tous les liens
            for u in liens :
                if u in pile or u in res :    # on passe au suivant si
                    continue                  # déjà vu

                # on enlève les images et autres fichiers indésirables
                if ".gif" in u or ".png" in u or ".jpg" in u : continue
                if ".cs" in u or ".css" in u or ".js" in u : continue

                # on ajoute le liens à la liste des liens à explorer
                pile.append (u)

        except IOError, exc:
            print "problème avec url ", u
            continue

    return res
```

Il ne reste plus qu'à appeler cette fonction pour récupérer la liste des URLs :

```
url  = "http://www.lemonde.fr/"   # un journal français
res  = list_url (url, url)
for r in res : print r

url  = "http://www.nytimes.com/"  # un journal américain
res  = list_url (url, url)
for r in res : print r
```

### Résultats

On détermine alors la fréquence des lettres sur 88 textes français et 181 textes anglais. Les résultats obtenus sont ceux de la figure 10.3. La frontière est nettement

plus floue et montre à première vue qu'il n'est pas toujours possible de distinguer une page française d'une page anglaise. Toutefois, la plupart des pages mal classées par cette méthode contiennent peu de texte et plus d'images : les fréquences portent plus sur la syntaxe HTML que sur un contenu littéraire. Il faudrait pour améliorer la qualité des résultats filtrer les pages pour ne conserver que leur contenu textuel. Cela montre aussi que cette méthode est plutôt robuste puisqu'on voit deux classes se dessiner nettement malgré les caractères parasites HTML.

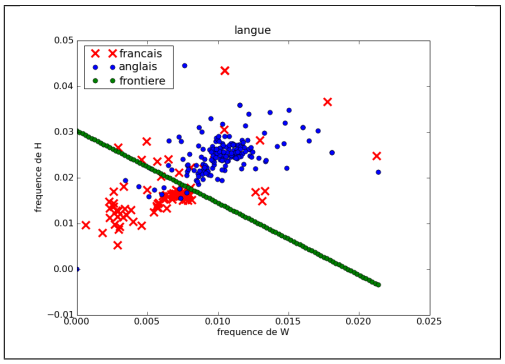



**Figure 10.3 :** *La frontière entre texte anglais et français paraît plus floue et ne dépend plus de la fréquence d'une seule lettre. L'axe des X représente la fréquence de la lettre W, l'axe des Y celle de la lettre H.*

### Pour aller plus loin

Jusqu'à présent, seules les lettres `W` et `H` ont été utilisées. Les deux autres lettres intéressantes `U` et `Y` ont été laissées de côté. On peut se demander quelle est la meilleure façon de procéder avec quatre lettres. Chaque texte est maintenant représenté par quatre fréquences, soit quatre coordonnées et il n'est possible de n'en représenter que deux.

Dans ce nuage de points à quatre dimension, l'analyse en composantes principales (ACP) [8] est une méthode qui permet de calculer un plan de projection, plus exactement le meilleur plan de projection : celui qui minimise les distances de chaque point au plan. On utilise pour cela le module `mdp` [9] qui implémente cet algorithme.

On suppose dans l'exemple suivant que `fr4` et `en4` sont deux matrices de quatre colonnes représentant les coordonnées de chaque document. La figure 10.4 montre que l'utilisation des quatre lettres `H`, `U`, `W`, `Y` permet de construire une frontière plus efficiente qu'avec l'utilisation de `W`, `H` seulement.

```
import mdp,copy,numpy
nbfr,nben = len (fr), len (en)
all = numpy.array (fr + en)    # construction du nuage de points

node = mdp.nodes.PCANode()     # ACP
node.train (all)               # construction de l'ACP
y    = node (all)              # on peut aussi écrire y = mdp.pca (all)

# obtention des coordonnées des points dans le plan de projection
frcx = [ y [i,0] for i in range (0, nbfr) ]
frcy = [ y [i,1] for i in range (0, nbfr) ]
```

8. voir le livre *Probabilités, analyse des données et statistique* de Gilbert Saporta (éditions Technip).

9. http://mdp-toolkit.sourceforge.net/

```
encx = [ y [i,0] for i in range (nbfr, nbfr + nben) ]
ency = [ y [i,1] for i in range (nbfr, nbfr + nben) ]

# dessin des points dans le plan de projection
# c'est le même code que précédemment
# ...
```

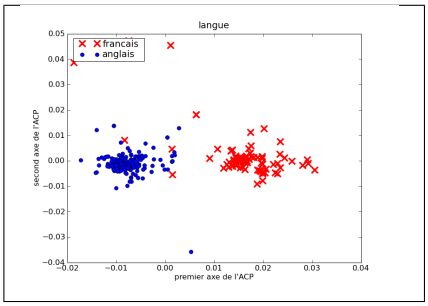



**Figure 10.4 :** *La frontière entre les textes anglais et français est beaucoup plus nette dans ce plan de projection issu de l'analyse en composantes principales.*

L'ACP donne de bons résultats dans la mesure où les deux classes sont visiblement disctinctes et concentrées. Dans un cas comme celui-ci pourtant, il serait plus adéquat d'utiliser une *Analyse Discriminante Linéaire* ou *Fisher Discriminant Analysis* (FDA) en anglais. Il s'agit de déterminer la meilleure séparation linéaire entre ces deux classes. Les résultats sont dans ce cas assez proches.

```
# même début que le programme précédent
allfr = numpy.array (fr)    # points français
allen = numpy.array (en)    # points anglais

node = mdp.nodes.FDANode ()
node.train (allfr, "fr")
node.train (allen, "en")
node.stop_training ()
node.train (allfr, "fr")
node.train (allen, "en")
y    = node (all)
```

## 10.4 Carrés magiques

Un carré magique est un carré 3x3 dont chaque case contient un nombre entier et pour lequel les sommes sur chacune des lignes, chacune des colonnes, chacune des diagonales sont égales, soit huit sommes égales. Le plus simple des carrés magiques est celui dont tous les chiffres sont égaux :

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 1 |
| 1 | 1 | 1 |

**1)** Un carré magique n'est a priori qu'une matrice de trois lignes et trois colonnes. Dans un premier temps, on va donc construire une classe `CarreMagique` qui contiendra une liste de trois listes, chacune de trois colonnes. Compléter le programme pour créer un carré magique `cm`.

```
class CarreMagique :
    def __init__ (self, nb) :
        self.nb = nb

m = [ [9, 3, 3], [ 4,5,6] , [3,8,2] ]
```

**2)** On désire afficher le carré magique. Que donne l'instruction `print cm` ? On ajoute la méthode `__str__`. Que produit l'instruction `print cm` maintenant ? Que faudrait-il changer pour que le programme affiche le carré en entier ?

```
    def __str__ (self) :
        s = ""
        s += str (self.nb [0][0])
        return s
```

**3)** Ecrire une méthode qui calcule la somme des chiffres sur la ligne `i`.

**4)** Ajouter une ligne dans la méthode `__str__` afin d'afficher aussi la somme des chiffres sur la ligne 0. Cette information correspond à la somme des chiffres sur chaque ligne ou chaque colonne si le carré est magique.

**5)** Ecrire une méthode qui calcule la somme des chiffres sur la colonne `j`.

**6)** Sans tenir compte des diagonales, écrire une méthode qui détermine si un carré est magique.

**7)** Il ne reste plus qu'à inclure les diagonales.

**8)** Ecrire une fonction qui détermine si tous les nombres du carré magique sont différents.

**9)** Terminer le programme pour déterminer tous les carrés magiques 3x3 dont les nombres sont tous différents.

**Remarque 10.3 : différence entre `print` et `return`**
A la fin d'un calcul, afin de voir son résultat, on utilise souvent l'instruction `print`. On peut se demander alors si à la fin de chaque fonction, il ne faudrait pas utiliser l'instruction `print`. A quoi servirait alors l'instruction `return` ? On suppose qu'un calcul est en fait le résultat de trois calculs à la suite :

```
a = calcul1 (3)
b = calcul2 (a)
c = calcul3 (b) # c résultat souhaité et affiché
```

Chaque terme `calculx` cache une fonction or seul le résultat de la dernière nous intéresse et doit être affiché. Pour les deux premières, la seule chose importante est que leur résultat soit transmis à la fonction suivante et ceci ne peut se faire que grâce à l'instruction `return`. L'instruction `print` insérée dans le code de la fonction `calcul1` ou `calcul2` permettra d'afficher le résultat mais ne le transmettra pas et il sera perdu. L'instruction `return` est donc indispensable, `print` facultative.

En revanche, dans la dernière fonction `calcul3`, il est possible de se passer de `return` et de se contenter uniquement d'un `print`. Cependant, il est conseillé d'utiliser quand même `return` au cas où le résultat de la fonction `calcul3` serait utilisé par une autre fonction.

### Correction

**1)** Il suffit de créer instance de type `CarreMagique`.

```
cm = CarreMagique (m)
```

**2)** Le message affiché par l'instruction `print cm` avant d'ajouter la méthode `__str__` correspond au suivant :

```
<__main__.CarreMagique instance at 0x01A254E0>
```

La réponse à cette question et aux suivantes sont présentées d'un seul tenant. Des commentaires permettent de retrouver les réponses aux questions de l'énoncé excepté pour la dernière question.

```
# coding: latin-1

class CarreMagique :

    def __init__ (self, nb) :
        """on place la matrice nb (liste de listes)
        dans la classe accessible par le mot-clé self"""
        self.nb = nb

    # réponse à la question 2
    def __str__ (self) :
        """méthode appelée lorsque on cherche à afficher
        un carré magique avec l'instruction print"""
        s = ""
        for i in range (0, len (self.nb)) :
            for j in range (0, len (self.nb)) : s += str ( self.nb [i][j]) + " "
            s += "\n" # pour passer à la ligne
        # réponse à la question 4
        s += "somme " + str (self.somme_ligne (0))
        return s

    # réponse à la question 3
    def somme_ligne (self, i) :
        s = 0
        for j in range (0, len (self.nb)) : s += self.nb [i][j]
        return s

    # réponse à la question 5
    def somme_colonne (self, j) :
        s = 0
        for i in range (0, len (self.nb)) : s += self.nb [i][j]
        return s

    # réponse à la question 6
    def est_magique (self) :
        # on stocke toutes les sommes
        l = []
        for i in range (0, len (self.nb)) : l.append ( self.somme_ligne (i) )
        for j in range (0, len (self.nb)) : l.append ( self.somme_colonne (j) )

        # réponse à la question 7
        l.append ( self.somme_diagonale (0))
```

```
        l.append ( self.somme_diagonale (1))

        # on trie la liste
        l.sort ()

        # on compare le plus petit et le plus grand, s'il sont égaux,
        # le carré est magique
        return l [0] == l [ len(l)-1 ]

    # réponse à la question 7
    def somme_diagonale (self, d) :
        """d vaut 0 ou 1, première ou seconde diagonale"""
        s = 0
        if d == 0 :
            for i in range (0, len (self.nb)) : s += self.nb [i][i]
        else :
            for i in range (0, len (self.nb)) :
                s += self.nb [i][len(self.nb)-i-1]
        return s

    # réponse à la question 8
    def nombre_differents (self) :
        """retourne True si tous les nombres sont différents,
        on place les nombres un par un dans un dictionnaire,
        dès que l'un d'eux s'y trouve déjà,
        on sait que deux nombres sont identiques, le résultat est False"""
        k = { }
        for i in range (0, len (self.nb)) :
            for j in range (0, len (self.nb)) :
                c = self.nb [i][j]
                if c in k : return False     # pas besoin d'aller plus loin
                                             # il y a deux nombres identiques
                else : k [c] = 0
        return True


m = [ [9, 3, 7],  [ 4,5,6] , [1,8,2] ]
cm = CarreMagique (m)           # réponse à la question 1
print cm                        # affiche 15
print cm.est_magique ()         # affiche False
print cm.nombre_differents ()   # affiche True

m = [ [9, 9, 9],  [9, 9, 9], [9, 9, 9] ]
cm = CarreMagique (m)
print cm                        # affiche 15
print cm.est_magique ()         # affiche True
print cm.nombre_differents ()   # affiche False
```

La dernière question consiste à trouver tous les carrés magiques dont les nombres sont tous différents. Il faut donc passer en revue tous les carrés 3x3 et ne conserver que ceux pour lesquels les méthodes `est_magique` et `nombre_differents` retournent un résultat positif. On suppose que le programme précédent contenant la classe `CarreMagique` porte le nom de `carre_magique.py`. Ceci explique la ligne `from`....

```
from carre_magique import CarreMagique
res = []
for a in range (1,10) :
    for b in range (1,10) :
```

```
        for c in range (1,10) :
            for d in range (1,10) :
                for e in range (1,10) :
                    for f in range (1,10) :
                        for g in range (1,10) :
                            for h in range (1,10) :
                                for i in range (1,10) :
                                    l  = [ [a,b,c], [d,e,f], [g,h,i] ]
                                    cm = CarreMagique (l)
                                    if cm.nombre_differents () and \
                                                cm.est_magique () :
                                        res.append (cm)

print len (res)
for r in res :
    print r
```

L'inconvénient de cette solution est qu'elle est inadaptée pour des carrés d'une autre dimension que 3x3. Il faudrait pouvoir inclure un nombre de boucles variable ce qui est impossible. L'autre solution est de programmer un compteur en reproduisant l'algorithme d'une addition. C'est l'objectif du programme suivant.

```
# coding: latin-1
from carre_magique import CarreMagique

dim = 3
nb  = dim*dim
res = []         # contiendra la liste des carrés magiques

# le compteur : neuf nombres 1
ind = [1 for i in range (0,nb) ]

while ind [0] <= nb :

    # transformation d'une liste en une liste de listes
    # [1,2,3,4,5,6,7,8,9] --> [[1,2,3],[4,5,6],[7,8,9]]
    l = []
    for i in range (0, dim) :
        l.append ( ind [i*dim:(i+1)*dim] )

    # on vérifie que le carré est magique et
    # a des nombres tous différents
    cm = CarreMagique (l)
    if cm.nombre_differents () and cm.est_magique () :
        res.append (cm)

    # on passe au carré suivant :
    i        = nb-1      # dernier indice (9 ici)
    ind [i] += 1         # addition de 1 au dernier indice
    while i > 0 and ind [i] > nb :
        ind [i-1] += 1   # un des indices est supérieur à nb (9 ici)
        ind [i]    = 1   # on le remet à 1
        i         -= 1   # et on propage l'information à l'indice inférieur

# résultat final
print len (res)
for r in res : print r
```

Il existe une autre manière de parcourir les carrés magiques en décomposant un entier compris entre 1 et $9^9$ en une succession de neuf entiers. On utilise pour cela la décomposition d'un nombre en base neuf : $n-1 = \sum_{k=1}^{9} \alpha_k(i_k - 1)$ où tous les nombres $i_k$ sont dans l'ensemble $\{1, \ldots, 9\}$.

```
# coding: latin-1
from carre_magique import CarreMagique

dim = 3
nb  = dim*dim
M   = 9 ** nb    # on va tester 9^9 carrés possibles
res = []         # contiendra la liste des carrés magiques

for n in xrange (0,M) :

    # on décompose n en liste
    ind = []
    k   = n
    for t in range (0,nb) :
        dec = k % nb
        k   = k / nb
        ind.append (dec+1)

    # transformation d'une liste en une liste de listes
    # [1,2,3,4,5,6,7,8,9] --> [[1,2,3],[4,5,6],[7,8,9]]
    l = []
    for i in range (0, dim) :
        l.append ( ind [i*dim:(i+1)*dim] )

    # on vérifie que le carré est magique et
    # a des nombres tous différents
    cm = CarreMagique (l)
    if cm.nombre_differents () and cm.est_magique () :
        res.append (cm)

# résultat final
print len (res)
for r in res : print r
```

Il faut noter l'utilisation de la fonction `xrange` au lieu de la fonction `range` lors de la première boucle. La fonction `xrange` ne crée pas de liste, elle simule un itérateur qui parcourt une liste. Dans notre cas, la fonction `range` crée une liste de $9^9$ éléments qui ne tient pas en mémoire, l'erreur suivante est déclenchée :

```
Traceback (most recent call last):
  File "carre_magique_tous5.py", line 9, in <module>
    for n in range (0,M) :
MemoryError
```

Ces trois solutions sont très lentes car elles explorent $9^9$ solutions. Comme on ne cherche que les carrés magiques dont les éléments sont tous différents, on pourrait n'explorer que les permutations soit 9! configurations. On crée une fonction `permutation` qui parcourt toutes les permutations possibles par récurrence (voir figure 10.5).

```
# coding: latin-1
from carre_magique import CarreMagique
```

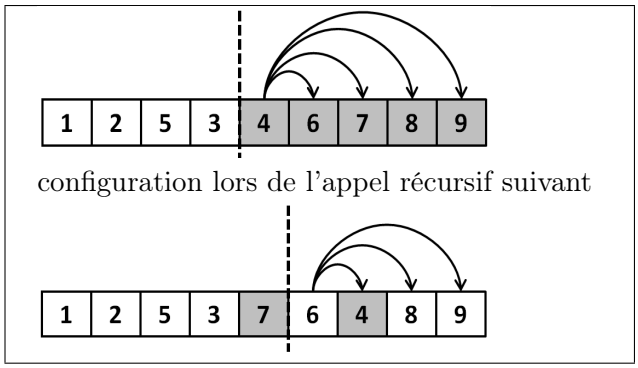



**Figure 10.5 :** *C'est une illustration de l'idée permettant de parcourir les permutations. On suppose que la partie du côté gauche du trait en pointillé (relié à la variable* pos*, ici égale à 4) est figée. On l'échange tour à tour avec les chiffre à droite (*pos + 1,pos + 2*, ...). A chaque échange, on appelle à nouveau la fonction* permutation *récursive en décalant le trait en pointillé d'un cran vers la droite. Après cet appel, on effectue l'échange inverse.*

```
def echange (p, i, pos) :
    # échange de deux éléments d'indice i et pos du tableau p
    if i != pos :
        e       = p [i]
        p [i]   = p [pos]
        p [pos] = e

def permutation (res, dim, pos = 0, p = None) :
    # parcours des permutations par récurrence
    # pour le premier appel à la fonction permutation
    if p == None : p = range (1, dim*dim+1)

    if pos < len (p) :
        # on organise les permutations de l'élément p [pos], p [pos+1], ...
        for i in range (pos, len (p)) :
            echange (p, i, pos)
            permutation (res, dim, pos+1, p)
            echange (p, i, pos)
    else :
        # pos correspond au dernier élément : il n'y a plus de permutation
        # possible avec les éléments qui suivent
        # on teste donc le nouveau carré
        l = []
        for i in range (0, len (p)/dim) :
            l.append ( p [i*dim:(i+1)*dim] )
        cm = CarreMagique (l)
        if cm.est_magique () : res.append (cm)

# permutations
res = []
permutation ( res, 3 )
# résultats
print "nombre de carrés ", len (res)
for r in res : print r
```

Le programme indique huit carrés magiques qui sont en fait huit fois le même carré obtenus par rotation ou symétrie. La somme des chiffres est 15.

## 10.5 Tri rapide ou quicksort

Cet énoncé a pour objectif de présenter l'algorithme de tri *quicksort* qui permet de trier par ordre croissant un ensemble d'éléments (ici des chaînes de caractères) avec un coût moyen [10] en $O(n \ln n)$ où $n$ est le nombre d'éléments à classer. Le tri *quicksort* apparaît rarement sous la forme d'un graphe : il est plus simple à programmer sans les graphes mais il est plus simple à appréhender avec les graphes. Dans cette dernière version, l'algorithme insère un à un les éléments d'une liste à trier dans un graphe comme celui de la figure 10.6 (page 265). Chaque nœud de ce graphe est relié à deux autres nœuds :

1. Un nœud `avant` ou " < " qui permet d'accéder à des éléments classés avant celui de ce nœud.
2. Un nœud `apres` ou " > " qui permet d'accéder à des éléments classés après celui de ce nœud.

Les nœuds `avant` et `apres` sont appelés les successeurs. Le terme opposé est prédécesseur. Ces deux nœuds ont nécessairement un prédécesseur mais un nœud n'a pas forcément de successeurs. S'il en avait toujours un, l'arbre serait infini.

**1)** On cherche à construire une classe ayant pour nom `NoeudTri` et qui contient une chaîne de caractères initialisée lors de la création de la classe : `n = NoeudTri("essai")`.

**2)** On écrit la méthode `__str__` de sorte que l'instruction `print n` affiche la chaîne de caractères que contient `n`.

**3)** On cherche maintenant à définir d'autres nœuds, reliés à des attributs `avant` et `apres`. On suppose que les nœuds utilisent l'attribut `mot`, on crée alors une méthode `insere(s)` qui :

- Si `s < self.mot`, alors on ajoute l'attribut `avant = NoeudTri(s)`.
- Si `s > self.mot`, alors on ajoute l'attribut `apres = NoeudTri(s)`.

| **Fonction utile :** | |
|---|---|
| `cmp(s1,s2)` | Compare deux chaînes de caractères, retourne -1,0,1 selon que `s1` est classée avant, est égale ou est classée après `s2`. |

**4)** La méthode `__str__` n'affiche pour le moment qu'un mot. Il s'agit maintenant de prendre en compte les attributs `avant` et `apres` afin que l'instruction `print n` affiche `avant.__str__()` et `apres.__str__()`. Il faudra également faire en sorte que la méthode `avant.__str__()` ne soit appelée que si l'attribut `avant` existe. Comme la liste des mots à trier est finie, il faut bien que certains nœuds n'aient pas de successeurs. On pourra s'inspirer du programme page 99 (attribut `__dict__`). Qu'est-ce qu'affiche le programme suivant ?

10. Lorsqu'on parle de coût moyen, cela signifie que le coût n'est pas constant en fonction de la dimension du problème. Ici, le coût moyen désigne le coût moyen d'un tri *quicksort* obtenu en faisant la moyenne du coût du même algorithme sur toutes les permutations possibles de l'ensemble de départ.

```
racine = NoeudTri ("un")
racine.insere ("unite")
racine.insere ("deux")
print racine
```

**5)** Est-il possible de trier plus de trois mots avec ce programme ? Que faut-il modifier dans la méthode `insere` afin de pouvoir trier un nombre quelconque de mots ?

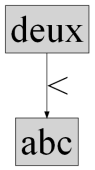
**6)** Ajouter le code nécessaire afin que la méthode `insere` génère une exception lorsqu'un mot déjà présent dans l'arbre est à nouveau inséré.

**7)** On se propose de construire une image représentant l'arbre contenant les mots triés par l'algorithme *quicksort*. Cette représentation utilise le module `pydot` qui utilise l'outil *Graphviz*[11] ainsi que le module `pyparsing`[12]. Leur installation est assez facile sous *Windows* puisqu'elle consiste seulement à exécuter un programme d'installation. Il faut ensuite installer manuellement le module `pydot`[13]. Après avoir décompressé les fichiers de ce module, il faut se placer dans le même répertoire et utiliser la ligne de commande suivante :

```
c:\python26\python setup.py install
```

Le tracé d'un graphe passe par l'écriture d'un fichier dans lequel on indique les nœuds et les arcs. Un nœud est un numéro suivi de [`label` = "`deux`"] qui indique que ce nœud contient le mot `deux`. Si on ajoute , `style` = `filled`, `shape` = `record`, le nœud est représenté par un rectangle au fond grisé. Un arc est un couple de numéros séparés par `->`. Le texte `[label="<"]` permet d'étiqueter cet arc. Selon cette syntaxe, le texte suivant décrit l'image située à droite :

```
digraph GA {
    2 [label="deux",style=filled,shape=record]
    3 [label="abc" ,style=filled,shape=record]
    2 -> 3 [label="<"]
}
```




| **Fonction** | **utile :** |
|---|---|
| `id(obj)` | Retourne un identifiant entier unique pour chaque variable `obj`. |

Une fois que cette chaîne de caractères a été construite, il suffit de l'écrire dans un fichier puis d'appeler le module `pydot` pour le convertir en image avec le code suivant :

```
g = open ("graph.txt", "w")  #
g.write (graph)              # partie écriture dans un fichier
g.close ()                   #
dot = pydot.graph_from_dot_file ("graph.txt") # partie graphe
dot.write_png ("graph.png", prog="dot")       # avec pydot
```

11. http://www.graphviz.org/
12. http://pyparsing.wikispaces.com/
13. http://code.google.com/p/pydot/

L'objectif de cette question est de construire une chaîne de caractères pour l'ensemble du graphe. La correction de cette exercice envisage également de construire une page HTML contenant le graphe et la liste triée ainsi que la création automatique d'un fichier au format PDF par le biais de *Latex*[14].

### Correction

**1)** La chaîne de caractères que contient `NoeudTri` s'appelle `mot`.

```
class NoeudTri (object):
    def __init__(self,s):
        self.mot = s
```

**2)**

```
class NoeudTri (object):
    def __init__(self,s): self.mot = s
    def __str__(self)   : return self.mot + "\n"  # \n : passage à la ligne
```

**3)**

```
class NoeudTri (object):
    def __init__(self,s): self.mot = s
    def __str__(self)   : return self.mot + "\n"

    def insere (self,s):
        c = cmp (s, self.mot)
        if   c == -1 : self.avant = NoeudTri (s)  # ajout d'un successeur
        elif c ==  1 : self.apres = NoeudTri (s)  # ajout d'un successeur
```

La méthode `insere` prévoit de ne rien faire dans le cas où le mot `s` passé en argument est égal à l'attribut `mot` : cela revient à ignorer les doublons dans la liste de mots à trier.

**4)**

```
class NoeudTri (object):
    def __init__(self,s): self.mot = s

    def __str__(self):
        s = ""
        if "avant" in self.__dict__: s += self.avant.__str__ ()
        s += self.mot + "\n"
        if "apres" in self.__dict__: s += self.apres.__str__()
        return s

    def insere (self,s):
        c = cmp (s, self.mot)
        if   c == -1 : self.avant = NoeudTri (s)
        elif c ==  1 : self.apres = NoeudTri (s)
```

14. *Latex* est un langage adapté aux publications scientifiques, également à l'écriture de livres comme celui-ci. En utilisant sa syntaxe, il permet de créer rapidement un fichier au format PDF.

L'insertion des mots donnés dans l'énoncé produit le code suivant :

```
deux
un
unite
```

**5) 6)** Il reste à compléter la fonction `insere` afin qu'elle puisse trouver le bon nœud où insérer un nouveau mot. Cette méthode est récursive : si un nœud contient deux attributs `avant` et `apres`, cela signifie que le nouveau mot doit être inséré plus bas, dans des nœuds reliés soit à `avant` soit à `apres`. La méthode `insere` choisit donc un des attributs et délègue le problème à la méthode `insere` de ce nœud.

```
# coding: latin-1
import string

class SecondeInserstion (AttributeError):
    "insertion d'un mot déjà inséré"

class NoeudTri :

    def __init__(self,s): self.mot = s

    # la création d'un nouveau noeud a été placée dans une méthode
    def nouveau_noeud (self, s) :
        return self.__class__ (s)
        #return NoeudTri (s)

    def __str__(self):
        s = ""
        if "avant" in self.__dict__: s += self.avant.__str__ ()
        s += self.mot + "\n"
        if "apres" in self.__dict__: s += self.apres.__str__()
        return s

    def insere (self,s):
        c = cmp (s, self.mot)
        if c == -1:
            if "avant" in self.__dict__ : self.avant.insere (s) # délégation
            else :  self.avant = self.nouveau_noeud (s)         # création
        elif c == 1:
            if "apres" in self.__dict__ : self.apres.insere (s) # délégation
            else: self.apres = self.nouveau_noeud (s)           # création
        else:
            raise SecondeInsertion, "mot : " + s

l = ["un", "deux", "unite", "dizaine", "exception", "dire", \
     "programme", "abc", "xyz", "opera", "quel"]

racine = None
for mot in l :
    if racine == None :
        # premier cas : aucun mot --> on crée le premier noeud
        racine = NoeudTri (mot)
    else :
        # second cas : il y a déjà un mot, on ajoute le mot suivant
        # à l'arbre
        racine.insere (mot)
```

```
print racine
```

Chaque nouveau mot va partir du tronc pour s'accrocher à une feuille de l'arbre pour devenir à son tour une feuille. La méthode `nouveau_noeud` crée un nouveau nœud dans le graphe. Son utilité est mise en évidence par le prochain programme.

**7)** La figure 10.6 détient le graphe obtenu par le programme qui suit. Plutôt que de modifier la classe `NoeudTri`, une seconde est créée qui hérite de la première. On lui adjoint la méthode `chaine_graphe` qui convertit un graphe en une chaîne de caractères dont le format reprend celui énoncé plus haut. Cette fonction s'occupe de construire récursivement cette chaîne de caractères. Pour identifier chaque nœud, on utilise la fonction `id` qui retourne un identifiant distinct pour chaque instance de classe.

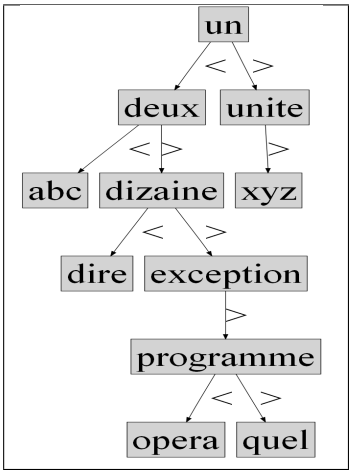



**Figure 10.6 :** *Graphe de tri obtenu lors du tri quicksort. Chaque nœud du graphe inclut un mot. Les symboles "<" et ">" des arcs désignent les membres* `avant` *et* `apres` *de la classe* `NoeudTri`*. Tous les mots attachés à un arc "<" d'un nœud sont classés avant le mot de ce nœud. De même, tous les mots attachés à un arc ">" d'un nœud sont classés après le mot de ce nœud.*

```
# coding: latin-1
import string
import pydot
import quicksort

class NoeudTri2 (quicksort.NoeudTri):

    def chaine_graphe (self):
        # le principe est le même que pour la méthode __str__
        # excepté que le format est différent
        g = str (id (self)) + ' [label="' + self.mot \
                            + '",style=filled,shape=record,fontsize=60]\n'
        if "avant" in self.__dict__:
            h  = self.avant.chaine_graphe ()
            g += h + str (id (self)) + " -> " + str (id (self.avant)) \
                   + ' [label="<",fontsize=60]' + '\n'
        if "apres" in self.__dict__:
            h  = self.apres.chaine_graphe ()
            g += h + str (id (self)) + " -> " + str (id (self.apres)) \
                   + ' [label=">",fontsize=60]' + "\n"
        return g

    #def nouveau_noeud (self, s) : return NoeudTri2 (s)
```

```
    def image (self, file, im) :
        # on crée un graphe : on récupère le code du graphe
        graph = self.chaine_graphe ()
        # auquel on ajoute un début et une fin
        graph = "digraph GA {\n" + graph + "}\n"

        # puis on l'écrit dans le fichier file
        g = open (file, "w")
        g.write (graph)
        g.close ()

        # enfin, on convertit ce fichier en image
        dot = pydot.graph_from_dot_file (file)
        dot.write_png (im, prog="dot")

def construit_arbre () :
    # même code que dans le programme précédent
    # mais inclus dans une fonction
    l = ["un", "deux", "unite", "dizaine", "exception", "dire", \
         "programme", "abc", "xyz", "opera", "quel"]
    racine = None
    for mot in l :
        if racine == None : racine = NoeudTri2 (mot)
        else : racine.insere (mot)
    return racine

racine = construit_arbre ()
print racine
racine.image ("graph.txt", "graph.png")
```

La méthode `nouveau_noeud` permet de s'assurer que tous les nœuds insérés lors de la création du graphe seront bien du type `NoeudTri2` qui inclut la méthode `chaine_graphe`. Cette méthode serait inutile s'il n'y avait qu'une seule classe `NoeudTri` contenant toutes les méthodes. Si on la met en commentaire, le message d'erreur suivant apparaît :

```
Traceback (most recent call last):
  File "quicksort2.py", line 53, in <module>
    racine.image ("graph.txt", "graph.png")
  File "quicksort2.py", line 27, in image
    graph = self.chaine_graphe ()
  File "quicksort2.py", line 14, in chaine_graphe
    h  = self.avant.chaine_graphe ()
AttributeError: NoeudTri instance has no attribute 'chaine_graphe'
```

L'erreur signifie que le programmeur cherche à appeler une méthode qui n'existe pas dans la classe `NoeudTri` parce que seul le premier nœud de l'arbre est de type `NoeudTri2` contrairement aux nœuds insérés par la méthode `nouveau_noeud` de la classe `NoeudTri`. En surchargeant cette méthode, on s'assure que tous les nœuds sont du même type `NoeudTri2`. Il existe néanmoins une façon d'éviter de surcharger cette fonction à chaque fous. Il suffit qu'elle crée automatiquement la bonne classe, que l'objet soit une instance de `NoeudTri` ou `NoeudTri2`. C'est ce que fait l'exemple suivant où `self.__class__` correspond à la classe de l'objet.

```
def nouveau_noeud (self, s) : return self.__class__ (s)
```

Au final, la méthode `image` construit l'image du graphe. Le fichier `graphe.txt` doit ressembler à ce qui suit :

```
digraph GA {
18853120 [label="un",style=filled,shape=record]
28505472 [label="deux",style=filled,shape=record]
28505712 [label="abc",style=filled,shape=record]
28505472 -> 28505712 [label="<"]
28505552 [label="dizaine",style=filled,shape=record]
28505592 [label="dire",style=filled,shape=record]
28505552 -> 28505592 [label="<"]
28505632 [label="exception",style=filled,shape=record]
28505672 [label="programme",style=filled,shape=record]
28505792 [label="opera",style=filled,shape=record]
28505672 -> 28505792 [label="<"]
28505832 [label="quel",style=filled,shape=record]
28505672 -> 28505832 [label=">"]
28505632 -> 28505672 [label=">"]
28505552 -> 28505632 [label=">"]
28505472 -> 28505552 [label=">"]
18853120 -> 28505472 [label="<"]
28505512 [label="unite",style=filled,shape=record]
28505752 [label="xyz",style=filled,shape=record]
28505512 -> 28505752 [label=">"]
18853120 -> 28505512 [label=">"]
}
```

La numérotation des nœuds importe peu du moment que chaque nœud reçoit un identifiant unique. C'est pour cela que la fonction `id` est pratique dans ce cas-là. Le programme suivant construit une sortie au format HTML mélangeant image et texte. Il commence par importer le programme `quicksort2` qui n'est autre que celui incluant la classe `NoeudTri2`. Il termine en appelant le navigateur *Mozilla Firefox* afin d'afficher le résultat automatiquement.

```
# coding: latin-1
import quicksort2

# construction de l'arbre
racine = quicksort2.construit_arbre ()
# construction de l'image du graphe
racine.image ("graph.txt", "graph.png")

# création d'un fichier HTML
f = open ("page.html", "w")
f.write ("<body><html>\n")              # début

f.write ("<H1> liste triée </H1>\n")    # titre pour la liste triée
s = str (racine)                        # on récupère la liste triée
s = s.replace ("\n", "<BR>\n")          # <BR> permet de passer à la ligne
f.write (s)

f.write ("<H1> graphe </H1>\n")         # titre pour l'image
f.write ('<img src="graph.png" width=400/>\n')    # image

f.write ("<H1> code du graphe </H1>\n") # titre pour le code du graphe
s = racine.chaine_graphe ()             # on récupère le code du graphe
f.write ("<pre>\n")                     # on l'affiche tel quel
f.write (s)
```

```
f.write ("</pre>\n")

f.write ("</html></body>\n")            # fin
f.close ()

# on lance le navigateur automatiquement pour afficher la page
import os
os.system (r'"C:\Program Files\Mozilla Firefox\firefox" page.html')
```

Le fichier `page.html` contient les lignes suivantes excepté les points de suspension qui remplacent la partie tronquée qu'on peut aisément deviner.

```
<body><html>
<H1> liste triée </H1>
abc<BR>
deux<BR>
...
unite<BR>
xyz<BR>
<H1> graphe </H1>
<img src="graph.png" width=400/>
<H1> code du graphe </H1>
<pre>
13697312 [label="un",style=filled,shape=record,fontsize=60]
13697192 [label="deux",style=filled,shape=record,fontsize=60]
34692472 [label="abc",style=filled,shape=record,fontsize=60]
...
13697232 -> 34692592 [label=">",fontsize=60]
13697312 -> 13697232 [label=">",fontsize=60]
</pre>
</html></body>
```

On veut modifier ce programme pour que la sortie ne soit plus au format HTML mais au format PDF. Pour cela, on utilise comme intermédiaire le langage *Latex* qui s'occupe de créer ce fichier au format PDF. Les dernières lignes sont propres au système *Windows* où on suppose que la version de *Latex* installée est *Miktex 2.7*[15]. Il faut également le logiciel *Adobe Reader*[16].

```
# coding: latin-1
import quicksort2

# construction de l'arbre
racine = quicksort2.construit_arbre ()
# construction de l'image du graphe
racine.image ("graph.txt", "graph.png")

# construction du début du fichier tex
package = """a4 amsmath amssymb subfigure float latexsym amsfonts
epic eepic makeidx multido varindex moreverb alltt fancyvrb fancyhdr
color eurosym tabularx placeins url shorttoc""".split ()

header = """\\documentclass[french,11pt]{article}\n\\usepackage[french]{babel}
\\usepackage[usenames]{color}\\usepackage{""" + \
                          "}\n\\usepackage{".join (package) + \
```

15. http://miktex.org/, ce logiciel est volumineux et assez long à télécharger.
16. http://www.adobe.com/fr/products/acrobat/readstep2.html

```
"""}\\usepackage[small,normal]{caption2}\\urlstyle{sf}
\\usepackage[pdftex]{graphicx}\usepackage[T1]{fontenc}
\DefineVerbatimEnvironment{verbatimx}{Verbatim}{frame=single,
framerule=.1pt, framesep=1.5mm, fontsize=\\footnotesize,xleftmargin=0pt}
\\begin{document}\n"""

# création d'un fichier tex
f = open ("page.tex", "w")
f.write (header)

f.write ("\\title{Tri quicksort}\n")    # définit le titre
f.write ("\\maketitle\n")               # écrit le titre
f.write ("\\tableofcontents\n")         # table des matières

f.write ("\\section{liste triée}\n")    # titre pour la liste triée
s = str (racine)                        # on récupère la liste triée
s = s.replace ("\n", "\\\\  \n")        # \\ passe à la ligne
f.write ("\\begin{tabular}{|l|}\n")
f.write (s)
f.write ("\\end{tabular}\n")

f.write ("\\section{graphe}\n")         # titre pour l'image
f.write ('\\includegraphics[height=5cm]{graph.png}\n')    # image

f.write ("\\section{code du graphe}\n") # titre pour le code du graphe
s = racine.chaine_graphe ()             # on récupère le code du graphe
f.write ("\\begin{verbatimx}\n")        # on l'affiche tel quel
f.write (s)
f.write ("\\end{verbatimx}\n")

f.write ("\\end{document}\n")            # fin
f.close ()

# on compile deux fois le fichier pour que la table des matières
# soit bien prise en compte
import os
os.system (r'"C:\Program Files\MiKTeX 2.7\miktex\bin\pdflatex" page.tex')
os.system (r'"C:\Program Files\MiKTeX 2.7\miktex\bin\pdflatex" page.tex')

# on affiche le résultat avec Adobe Reader
os.system (r'"C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe" page.pdf')
```

La partie `header` est longue dans cet exemple, elle inclut des packages Latex qui ne sont pas utilisés mais qui pourraient l'être dans le cas d'un rapport plus long. Il faut bien sûr connaître quelques rudiments de ce langage pour construire le document PDF. L'avantage de ce système est de pouvoir retravailler manuellement le document final. Il est également indiqué lorsque le rapport mélange tableaux de chiffres récupérés depuis différentes sources et graphiques qui peuvent être générés par *Python* via un module comme `matplotlib` [17].

Sous *Linux*, il suffit de modifier les chemins d'accès aux différentes applications. Une dernière remarque, il existe sous *Windows* un éditeur convenable qui est *TeXnicCenter* [18].

17. http://matplotlib.sourceforge.net/
18. http://www.toolscenter.org/

# Chapitre 11

# Exercices pratiques pour s'évaluer

---

Les trois énoncés qui suivent sont prévus pour être réalisés en deux heures sur un ordinateur. Ils ne font référence à aucun algorithme ou méthode connue. Il suffit de connaître les types standard du *Python*, les boucles, les tests et les fonctions. Le second énoncé nécessite également quelques notions sur les classes.

## 11.1 Recherche dichotomique

L'objectif de cet exercice est de programmer une recherche dans une liste triée.

**1)** Il faut d'abord récupérer un fichier texte disponible sur Intranet [1]. Ce fichier contient un mot par ligne. Il faut lire ce fichier et construire une liste avec tous ces mots.

**2)** Construire une fonction qui vérifie que la liste chargée à la question précédente est triée.

**3)** Construire une fonction qui recherche un mot X dans la liste et qui retourne sa position ou -1 si ce mot n'y est pas. Cette fonction prend deux paramètres : la liste et le mot à chercher. Elle retourne un entier. On précise que pour savoir si deux chaînes de caractères sont égales, il faut utiliser l'opérateur ==.

**4)** Quels sont les positions des mots "UN" et "DEUX" ? La réponse doit figurer en commentaire dans le programme. Il faudra écrire aussi le nombre de comparaisons effectuées pour trouver ces deux positions.

**5)** Lorsqu'une liste est triée, rechercher un élément est beaucoup plus rapide. Si on cherche le mot X dans la liste, il suffit de le comparer au mot du milieu pour savoir si ce mot est situé dans la partie basse (X inférieur au mot du milieu), la partie haute (X supérieur au mot du milieu). S'il est égal, le mot a été trouvé. Si le mot n'a pas été trouvé, on recommence avec la sous-liste inférieure ou supérieure selon les cas jusqu'à ce qu'on ait trouvé le mot ou qu'on soit sûr que le mot cherché n'y est pas.

Le résultat de la recherche est la position du mot dans la liste ou -1 si ce mot n'a pas été trouvé. Cette recherche s'appelle une recherche dichotomique.

Ecrire la fonction qui effectue la recherche dichotomique d'un mot dans une liste triée de mots. Vérifiez que les deux fonctions retournent bien les mêmes résultats. Cette fonction peut être récursive ou non. Elle prend au moins les deux mêmes paramètres que ceux de la question 3, si elle en a d'autres, il faudra leur donner

1. http://www.xavierdupre.fr/enseignement/initiation/td_note_texte.txt

une valeur par défaut. On précise que les comparaisons entre chaînes de caractères utilisent aussi les opérateurs $<$, $==$, $>$.

**6)** Normalement, les positions des mots "UN" et "DEUX" n'ont pas changé mais il faut de nouveau déterminer le nombre d'itérations effectuées pour trouver ces deux positions avec la recherche dichotomique.

**7)** Quel est, au pire [2], le coût d'une recherche non dichotomique ? La réponse doit figurer en commentaire dans le programme.

**8)** Quel est, au pire, le coût d'une recherche dichotomique ? La réponse doit figurer en commentaire dans le programme.

### Correction

```
# coding: latin-1
# question 1
def lit_fichier (file) :
    f = open (file, "r")
    mot = []
    for l in f :
        mot.append ( l.replace ("\n", "") )
    f.close ()
    return mot

mot = lit_fichier ("td_note_texte.txt")
print mot

# question 2
def est_trie (mot) :
    for i in range (1, len (mot)) :
        if mot [i-1] > mot [i] :
            return False
    return True

tri = est_trie (mot)
print "liste triée ", tri

# question 3
def cherche (mot, m) :
    for i in range (0, len (mot)) :
        if mot [i] == m :
            return i
    return -1

print "mot ACHATS ", cherche (mot, "ACHATS")
print "mot achats ", cherche (mot, "achats")

# question 4
un   = cherche (mot, "UN")
deux = cherche (mot, "DEUX")
print "recherche normale ", un, deux
print "nombre d'itérations", un + deux

# question 5, 6, nbun et nbdeux contiennent le nombre de comparaisons
def cherche_dicho (mot, m) :
```

2. On cherche la meilleure majoration du coût de la recherche non dichotomique en fonction de la taille $n$ de la liste.

```
    a = 0
    b = len (mot)-1
    nb = 0
    while a < b :
        nb += 1
        p = (a+b)/2
        if   mot [p] == m :  return p,nb
        elif mot [p] >  m :  b = p-1
        else :               a = p+1
    return -1,nb

un,nbun     = cherche_dicho (mot, "UN")
deux,nbdeux = cherche_dicho (mot, "DEUX")
print "recherche dichotomique ", un, deux
print "nombre d'itérations ", nbun + nbdeux

# question 7
"""
Lors d'une recherche simple, au pire, l'élément cherche sera
en dernière position, ce qui signifie n itérations pour le trouver.
Le coût de la recherche simple est en O(n).
"""

# question 8
"""
Lors de la recherche dichotomique, à chaque itération, on divise par deux
l'ensemble dans lequel la recherche s'effectue,
au départ n, puis n/2, puis n/4 jusqu'àã ce que n/2^k soit nul
c'est-à-dire k = partie entière de ln n / ln 2
il y a au plus k itérations donc le coût de l'algorithme est en O (ln n).
"""
```

## 11.2 Ajouter un jour férié

### 11.2.1 Avec des fonctions uniquement

Le gouvernement désire ajouter un jour férié mais il voudrait le faire à une date éloignée des jours fériés existant. On suppose également que ce jour ne sera pas inséré entre Noël et le jour de l'an. On va donc calculer le nombre de jours qui sépare deux jours fériés dont voici la liste pour l'année 2007 :

| | |
|---|---|
| Jour de l'an | 1er janvier 2007 |
| Lundi de Pâques | 9 avril 2007 |
| Fête du travail | 1er mai 2007 |
| Victoire de 1945 | 8 mai 2007 |
| Ascension | 17 mai 2007 |
| Lundi de Pentecôte | 4 juin 2007 |
| Fête nationale | 14 juillet 2007 |
| Assomption | 15 août 2007 |
| Toussaint | 1er novembre 2007 |
| Armistice de 1918 | 11 novembre 2007 |
| Noël | 25 décembre 2007 |

On rappelle que l'année 2007 n'est pas une année bissextile et qu'en conséquence, le mois de février ne comporte que 28 jours.

**1)** Afin de simplifier la tâche, on cherche à attribuer un numéro de jour à chaque jour férié : l'année a 365 jours, pour le numéro du lundi de Pâques, soit 31 (mois de janvier) + 28 (février) + 31 (mars) + 9 = 89. La première question consiste à construire une fonction qui calcule le numéro d'une date étant donné un jour et un mois. Cette fonction prend comme entrée :

- un numéro de jour
- un numéro de mois
- une liste de 12 nombres correspondant au nombre de jours dans chacun des douze mois de l'année

**2)** Si on définit la liste des jours fériés comme étant une liste de couples (jour, mois) triée par ordre chronologique, il est facile de convertir cette liste en une liste de nombres correspondant à leur numéro dans l'année. La fonction à écrire ici appelle la précédente et prend une liste de couples en entrée et retourne comme résultat une liste d'entiers.

**3)** A partir de cette liste d'entiers, il est facile de calculer l'écart ou le nombre de jours qui séparent deux jours fériés. Il ne reste plus qu'à écrire une fonction qui retourne l'écart maximal entre deux jours fériés, ceux-ci étant définis par la liste de numéros définie par la question précédente. Un affichage du résultat permet de déterminer les deux jours fériés les plus éloignés l'un de l'autre. Quels sont-ils ?

### 11.2.2 Programme équivalent avec des classes

Le programme précédent n'utilise pas de classe. L'objectif de ce second exercice est de le réécrire avec une classe.

**1)** Une fonction du programme précédent effectue la conversion entre un couple jour-mois et un numéro de jour. Les calculs sont faits avec le numéro mais le résultat désiré est une date : les numéros ne sont que des intermédiaires de calculs qui ne devraient pas apparaître aussi explicitement. La première question consiste à créer une classe `Date` :

```
class Date :
    def __init__ (self, jour, mois) :
        ...
```

**2)** A cette classe, on ajoute une méthode qui retourne la conversion du couple jour-mois en un numéro de jour de l'année.

**3)** On ajoute maintenant une méthode calculant le nombre de jours séparant deux dates (ou objet de type `Date` et non pas numéros). Cette méthode pourra par exemple s'appeler `difference`.

**4)** Il ne reste plus qu'à compléter le programme pour obtenir les mêmes résultats que le programme de l'exercice 1.

**5)** Avec ce programme, lors du calcul des écarts entre tous les jours fériés consécutifs, combien de fois effectuez-vous la conversion du couple jour-mois en numéro pour

le second jour férié de l'année ? Est-ce le même nombre que pour le programme précédent (en toute logique, la réponse pour le premier programme est 1) ?

**6)** La réponse à la question précédente vous suggère-t-elle une modification de ce second programme ?

### Correction

```
# coding: latin-1
###################################
# exercice 1
###################################

# question 1
def numero (jour, mois, duree = [31, 28, 31,30,31,30,31,31,30,31,30,31] ) :
    s = 0
    for i in range (0,mois-1) :
        s += duree [i]
    s += jour - 1
    return s+1

# question 2
def conversion_liste (li) :
    res = []
    for jour,mois in s : res.append ( numero (jour, mois))
    # pareil que
    # for i in range (0, len (s)) : res.append ( numero (s [i][0], s [i][1]))
    return res

def ecart (num) :
    res = []
    for i in range (1, len (num)) :
        d = num [i] - num [i-1]
        res.append (d)
    return res


s = [ (1,1), (9,4), (1,5), (8,5), (17,5), (4,6), (14,7), \
        (15,8), (1,11), (11,11), (25,12) ]
r = conversion_liste (s)
ec = ecart (r)

# question 3
pos = ec.index ( max (ec) )
print "position de l'écart le plus grand ", pos
print "jour ", s [pos], " --> ", s [pos+1]

###################################
# exercice 2
###################################

# question 4
class Date :
    def __init__ (self, jour, mois) :
        self.jour = jour
        self.mois = mois
        self.duree = [31, 28, 31,30,31,30,31,31,30,31,30,31]
```

```
    # question 5
    def numero (self) :
        s = 0
        for i in range (0,self.mois-1) :
            s += self.duree [i]
        s += self.jour - 1
        return s+1

    # question 6
    def difference (self, autre) :
        return self.numero () - autre.numero ()

def conversion_date (s) :
    res = []
    for jour,mois in s :
        res.append ( Date (jour, mois) )
    return res

def ecart_date (date) :
    ec = []
    for i in range (1, len (date)) :
        ec.append ( date [i].difference ( date [i-1] ) )
    return ec


# question 7
s = [ (1,1), (9,4), (1,5), (8,5), (17,5), (4,6), \
            (14,7), (15,8), (1,11), (11,11), (25,12) ]

r = conversion_date (s)
ec = ecart_date (r)
pos = ec.index ( max (ec) )
print "position de l'ecart le plus grand ", pos
print "jour ", s [pos], " --> ", s [pos+1]

# question 8
"""
La conversion en Date est faite une fois pour les dates (1,1) et (25,12)
et 2 fois pour les autres en effet, la méthode difference effectue
la conversion en numéros des dates self et autre
la fonction ecart_date calcule date [i].difference ( date [i-1] ) et
                                        date [i+1].difference ( date [i] )
            --> la date [i] est convertie 2 fois
"""

# question 9
"""
On peut par exemple stocker la conversion en numéro
dans le constructeur comme suit :
"""

class Date :
    def __init__ (self, jour, mois) :
        self.jour = jour
        self.mois = mois
        self.duree = [31, 28, 31,30,31,30,31,31,30,31,30,31]
        self.num = self.numero ()

    # question 5
```

```
    def numero (self) :
        s = 0
        for i in range (0,self.mois-1) :
            s += self.duree [i]
        s += self.jour - 1
        return s+1

    # question 6
    def difference (self, autre) :
        return self.num - autre.num


r = conversion_date (s)
ec = ecart_date (r)
pos = ec.index ( max (ec) )
print "position de l'écart le plus grand ", pos
print "jour ", s [pos], " --> ", s [pos+1]
```

On pourrait encore améliorer la dernière classe `Date` en créant un attribut statique pour l'attribut `duree` qui est identique pour toutes les instances de la classe `Date`.

## 11.3 Fréquentation d'un site Internet

Lorsqu'on se connecte à un site internet, celui-ci enregistre votre adresse IP, l'heure et la date de connexion ainsi que la page ou le fichier désiré. Ainsi le fichier *logpdf.txt*[3] recense ces quatre informations séparées par des espaces pour les fichiers d'extension *pdf* téléchargés sur le site http://www.xavierdupre.fr/.

**1)** La première étape consiste à charger les informations depuis le fichier texte dans une matrice de 4 colonnes (liste de listes, liste de t-uples). On utilisera pour cela les fonctions `open` et les méthodes `split`, `replace` associées aux chaînes de caractères. On pourra s'aider des corrections des TD précédents.

**2)** On souhaite dans un premier temps faire des statistiques sur les dates : on veut compter le nombre de fichiers téléchargés pour chaque date présente dans les données. On pourra pour cela utiliser un dictionnaire dont la clé sera bien choisie. Le résultat devra être inséré dans une fonction prenant comme entrée une matrice (ou liste de listes, liste de t-uples) et retournant un dictionnaire comme résultat.

**3)** On s'intéresse au programme suivant :

```
l = [ (1, "un"), (3, "deux",), (2, "deux"), (1, "hun"), (-1, "moinsun") ]
l.sort (reverse = True)
print l
```

Donc le résultat est :

```
[(3, 'deux'), (2, 'deux'), (1, 'un'), (1, 'hun'), (-1, 'moinsun')]
```

Que s'est-il passé ?

**4)** On désire maintenant connaître les 10 dates pour lesquelles il y a eu le plus de téléchargements ces jours-là. L'inconvénient est que le dictionnaire élaboré à la

3. Il faut télécharger ce fichier depuis l'adresse http://www.xavierdupre.fr/enseignement/initiation/logpdf.txt

question 2 ne retourne pas les réponses dans l'ordre souhaité : il faut classer les dates par nombre de téléchargements croissants. Il faut ici imaginer une fonction qui retourne ces dix meilleures dates et des dix fréquentations correspondantes en se servant de la remarque de la question 3.

**5)** Effectuez le même travail pour déterminer les dix documents les plus téléchargés.

**6)** Ecrire une fonction qui retourne l'heure sous forme d'entier à partir d'une date définie par une chaîne de caractères au format "`hh:mm:ss`". Par exemple, pour "`14:55:34`", la fonction doit retourner 14 sous forme d'entier. L'instruction `int("14")` convertit une chaîne de caractères en un entier.

**7)** Calculer le nombre de documents téléchargés pour chaque heure de la journée. Le site est-il consulté plutôt le matin ou le soir ? Serait-il possible de conclure aussi rapidement pour un site d'audience internationale ?

### Correction

```
# coding: latin-1
# la première ligne autorise les accents dans un programme Python
# la langue anglaise est la langue de l'informatique,
# les mots-clés de tous les langages
# sont écrits dans cette langue.

####################################
# exercice 1
####################################
#

# question 1
def lit_fichier (file) :
    f = open (file, "r")
    li = f.readlines ()           # découpage sous forme de lignes
    f.close ()
    res = []
    for l in li :
        s = l.replace ("\n", "")
        s = s.split (" ")         # le séparateur des colonnes est l'espace
        res.append (s)
    return res


mat = lit_fichier ("logpdf.txt")
for m in mat [0:5] :       # on affiche les 5 premières lignes
    print m                # parce que sinon, c'est trop long

# question 2
def compte_date (mat) :
    d = { }
    for m in mat :
        date = m [1]   # clé
        if date in d : d [date] += 1
        else : d [date] = 1
    return d

dico_date = compte_date (mat)
print dico_date
```

```
# remarque générale : si le fichier logpdf.txt contient des lignes
# vides à la fin, il se produira une erreur à la ligne 34 (date = m [1])
# car la matrice mat contiendra des lignes avec une seule colonne et non quatre.
# Il suffit soit de supprimer les lignes vides du fichier logpdf.txt
# soit de ne pas les prendre en compte lors de la lecture de ce fichier.

# question 3
# La méthode sort trie la liste mais comment ?
# Il est facile de trier une liste de nombres mais une liste de couples de
# nombres ? L'exemple montre que la liste est triée selon le premier élément de
# chaque couple. Pour les cas où deux couples ont un premier élément en commun,
# les éléments semblent triés selon le second élément. L'exemple suivant le monte :

l = [(1, "un"), (3, "deux",), (2, "deux"), (1, "hun"), (1, "un"), (-1, "moinsun")]
l.sort (reverse = True)
print l  # affiche [(3, 'deux'), (2, 'deux'), (1, 'un'),
         #          (1, 'un'),   (1, 'hun'), (-1, 'moinsun')]

# question 4
def dix_meilleures (dico) :
    # dans cette fonction on crée une liste de couples (valeur,clé) ou
    # la clé représente une date et valeur le nombre de téléchargement
    # pour cette date
    li = []
    for d in dico :
        cle = d
        valeur = dico [cle]
        li.append ( ( valeur, cle ) )
    li.sort (reverse = True)
    return li [0:10]

dix = dix_meilleures (dico_date)
print dix # la première date est (283, '26/Sep/2007')

# les quatre premières dates correspondent aux quatre premiers TD en 2007 à l'ENSAE

# question 5
# la date est en colonne 1, le document en colonne 3
# on fait un copier-coller de la fonction compte_date en changeant un paramètre
def compte_document (mat) :
    d = { }
    for m in mat :
        doc = m [3]   # clé, 3 au lieu de 1 à la question 2
        if doc in d : d [doc] += 1
        else : d [doc] = 1
    return d

dix = dix_meilleures ( compte_document (mat) )
print dix   # le premier document est
            # (323, '/mywiki/Enseignements?.....target=python_cours.pdf'),

# question 6
def heure (s) :
    hs = s [0:2]           # on extrait la partie correspondant à l'heure
    return int (hs)        # on retourne la conversion sous forme d'entiers

# question 7
# on recommence avec un copier-coller
def compte_heure (mat) :
```

```
    d = { }
    for m in mat :
        h = m [2]   # clé, 2 au lieu de 1 à la question 2
        cle = heure (h)
        if cle in d : d [cle] += 1
        else : d [cle] = 1
    return d

h = compte_heure (mat)
dix = dix_meilleures ( h )
print dix   # la première heure est  (432, 17), ce qui correspond à l'heure des TD

for i in h :
    print i, "h  ", h [i]

# Il y a beaucoup plus de téléchargement entre 20h et 2h du matin
# que le matin avant 10h.
# Le site est plutôt consulté le soir.
# La conclusion ne serait pas aussi évidente avec un site consulté par des gens
# du monde entier puisque 6h du matin est une heure de l'après midi au Japon.
# Il faudrait croiser l'heure avec la position géographique de la personne
# qui consulte le site.
```

# Chapitre 12

# Exercices écrits

---

Chacun des énoncés qui suit est une série d'exercices auxquels une personne expérimentée peut répondre en une heure. C'est souvent le temps imparti à un test informatique lors d'un entretien d'embauche pendant lequel le candidat propose des idées plus qu'il ne rédige des solutions. Chaque énoncé est un panel représentatif des problèmes que la programmation amène inévitablement. La difficulté de chaque exercice est notée du plus facile au plus difficile : *, **, ***. Le sixième énoncé 12 est structuré sous forme de problème. Il cherche à reproduire la démarche suivie lors de la conception d'un algorithme.

## 12.1 Premier énoncé

### 12.1.1 Programme à deviner *

Que fait le programme suivant ? Que contiendra `res` à la fin du programme ?

```
l = [ 0,1,2,3,4,6,5,8,9,10]
res = True
for i in range (1,len (l)) :
    if l[i-1] > l[i] :
        res = False
```

#### Correction

Le programme vérifie que la liste `l` est triée, `res` vaut `True` si elle est triée, `False` sinon. Et dans ce cas précis, elle vaut `False` car la liste n'est pas triée (4,6,5).

### 12.1.2 Somme des chiffres d'un nombre *

Ecrire une fonction qui calcule la somme des chiffres d'un entier positif.

#### Correction

Tout d'abord, obtenir la liste des chiffres d'un nombre entier `n` n'est pas immédiat bien qu'elle puisse s'écrire en une ligne :

```
def somme (n) :
    return sum ( [ int (c) for c in str (n) ] )
```

Cette écriture résumée cache une conversion en chaîne de caractères. Une boucle est inévitable à moins de procéder par récurrence. Commençons par une version plus étendue et qui passe par les chaînes de caractères. Pour cette version, il ne faut pas oublier de faire la conversion inverse, c'est-à-dire celle d'un caractère en nombre.

```
def somme (n) :
    l = str (n)           # il ne faut pas confondre l=str (n) avec l = "n"
    s = 0
    for c in l :          # ou   for i in range (0, len (c)) :
        s += int (c)      # ou       s += int (c [i])
    return s
```

Une version numérique maintenant, celle qui utilise l'opération % ou modulo pour obtenir le dernier chiffre :

```
def somme (n) :
    s = 0
    while n > 0 :
        s += n % 10
        n /= 10    # ici, c'est une division entière, si vous n'êtes pas sûr :
                   #       n = int (n/10)
    return n
```

Enfin, une autre solution utilisant la récurrence et sans oublier la condition d'arrêt :

```
def somme (n) :
    if n <= 0 : return 0
    else : return (n % 10) + somme ( n / 10 )
```

Parmi les autres solutions, certaines, exotiques, ont utilisé la fonction `log` en base 10 ou encore la fonction `exp`. En voici une :

```
import math
def somme (n) :
    k = int (math.log (n) / math.log (10) + 1)
    s = 0
    for i in range (1,k+1) :
        d = 10 ** i   # ou encore d = int (exp ( k * log (10) ) )
        c = n / d
        e = n - c * d
        f = e / (d / 10)
        s += f
    return s
```

L'idée principale permet de construire une fonction retournant le résultat souhaité mais il faut avouer qu'il est plus facile de faire une erreur dans cette dernière fonction que dans les trois précédentes.

### 12.1.3 Calculer le résultat d'un programme **

Que vaut `n` à la fin de l'exécution du programme suivant ? Expliquez en quelques mots le raisonnement suivi.

```
n = 0
for i in range (0,10) :
    if (n + i) % 3 == 0 :
        n += 1
```

**Correction**

L'ensemble `range(0,10)` est égale à la liste $[0,1,2,3,4,5,6,7,8,9]$. Dans ce programme, `n` est incrémenté lorsque `i+n` est un multiple de 3. La première incrémentation a lieu lors de la première itération puisque 0 est un multiple de 3. On déroule le programme dans le tableau suivant :

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| n avant le test | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 |
| i+n | 0 | 2 | 3 | 5 | 6 | 8 | 9 | 11 | 12 | 14 |
| n après le test | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 |

`range(0,10)` contient les nombres de 0 à 10 **exclu**. La réponse cherchée est donc 5.

### 12.1.4 Suite récurrente (Fibonacci) *

Ecrire un programme qui calcule l'élément $u_{13}$ de la suite définie par :

$$\begin{array}{rrcl} & u_1 &=& 1 \\ & u_2 &=& 2 \\ \forall n \geqslant 2, & u_n &=& u_{n-1} + u_{n-2} \end{array}$$

$u_{13}$ est le nombre de manières possibles pour une grenouille de monter en haut d'une échelle de 13 barreaux si elle peut faire des bonds de un ou deux barreaux seulement. Si cette affirmation, qui n'est pas à démontrer, vous semble douteuse, avant d'y revenir, faites d'abord le reste de l'énoncé.

**Correction**

Tout d'abord, la grenouille : elle peut faire des bonds de un ou deux barreaux à chaque fois. Donc, lorsqu'elle est sur le barreau $n$, elle a pu venir du barreau $n-1$ et faire un saut de 1 barreau ou venir du barreau $n-2$ et faire un bond de 2 barreaux. Par conséquent, le nombre de manières qu'a la grenouille d'atterrir sur le barreau $n$, c'est la somme des manières possibles de venir jusqu'au barreau $n-1$ et du barreau $n-2$.

Maintenant, comment le programmer ? Tout d'abord les solutions récursives, les plus simples à programmer :

```
def grenouille (n) :
    if   n == 2 : return 2
    elif n == 1 : return 1
    else :        return grenouille (n-1) + grenouille (n-2)
print grenouille (13)
```

Cette solution n'est pas très rapide car si `grenouille(13)` va être appelée une fois, `grenouille(12)` une fois aussi, mais `grenouille(11)` va être appelée deux fois... Cette solution implique un grand nombre de calculs inutiles.

Après la solution récursive descendante (`n` décroît), la solution récursive montante (`n` croît) :

```
def grenouille (fin, n = 2, u1 = 1, u2 = 2) :
    if   fin == 1   : return u1
    elif fin == 2   : return u2
    elif n   == fin : return u2
    u = u1 + u2
    return grenouille (fin, n+1, u2, u)
print grenouille (13)
```

La méthode la plus simple est non récursive mais il ne faut pas se tromper dans les indices :

```
def grenouille (n) :
    if n == 1 : return 1
    u1 = 1
    u2 = 2
    for i in range (3,n+1) :
        u  = u1 + u2   # il est impossible de
        u1 = u2        # résumer ces trois lignes
        u2 = u         # en deux
    return u2
print grenouille (13)
```

Quelques variantes utilisant les listes :

```
def grenouille (n) :
    if n == 1 : return 1
    u = [1,2]
    for i in range (2,n) :
        u.append (u [i-1] + u [i-2])
    return u [n-1]
print grenouille (12)
```

Ou encore :

```
def grenouille (n) :
    if n == 1 : return 1
    u = range (0, n)  # il ne faut pas oublier de créer le tableau
                      # avec autant de cases que nécessaire
                      # ici 13
    u [0] = 1
    u [1] = 2
    for i in range (2,n) :
        u [i] = u [i-1] + u [i-2]
    return u [n-1]
print grenouille (12)
```

### 12.1.5 Comprendre une erreur d'exécution *

On écrit le programme suivant :

```
a = "abcdefghijklmnopqrstuvwxyz"
print len (a)
d = {}
for i in range (0,len (a)) :
    d [ a [ i ] ] = i

print d ["M"]
```

Une erreur est déclenchée lors de l'exécution :

```
examen.py:14:  KeyError: 'M'
```

Que faudrait-il écrire pour que le programme marche ? Qu'affichera-t-il alors ?

**Correction**

L'erreur signifie que la clé "M" n'est pas présente dans le dictionnaire d. Elle n'est présente qu'en minuscule. La modification proposée est la suivante :

```
a = "abcdefghijklmnopqrstuvwxyz"
print len (a)
d = {}
for i in range (0,len (a)) :
    d [ a [ i ] ] = i

print d ["m"]         ######   ligne modifiée
```

Mais il était également possible de procéder comme suit :

```
a = "abcdefghijklmnopqrstuvwxyz"
a = a.upper ()        #######   ligne ajoutée
print len (a)
d = {}
for i in range (0,len (a)) :
    d [ a [ i ] ] = i

print d ["M"]
```

Dans les deux cas, le programme affiche la position de M dans l'alphabet qui est 12 car les indices commencent à 0.

### 12.1.6 Copie de variables **

```
def somme (tab) :
    l = tab[0]
    for i in range (1, len (tab)) :
        l += tab [i]
    return l
ens = [[0,1],[2,3]]
print somme ( ens )   # affiche [0,1,2,3]
print ens             # affiche [ [0,1,2,3], [2,3] ]
```

La fonction somme est censée faire la concaténation de toutes les listes contenues dans ens. Le résultat retourné est effectivement celui désiré mais la fonction modifie également la liste ens, pourquoi ?

### Correction

Le problème vient du fait qu'une affectation en *Python* (seconde ligne de la fonction `somme`) ne fait pas une copie mais crée un second identificateur pour désigner la même chose. Ici, `l` et `tab[0]` désignent la même liste, modifier l'une modifie l'autre. Ceci explique le résultat. Pour corriger, il fallait faire une copie explicite de `tab[0]` :

```
import copy                       ###### ligne ajoutée
def somme (tab) :
    l = copy.copy (tab[0])        ###### ligne modifiée
    for i in range (1, len (tab)) :
        l += tab [i]
    return l
ens = [[0,1],[2,3]]
print somme ( ens )   # affiche [0,1,2,3]
print ens             # affiche [ [0,1,2,3], [2,3] ]
```

Il était possible, dans ce cas, de se passer de copie en écrivant :

```
def somme (tab) :
    l = []                            ###### ligne modifiée
    for i in range (0, len (tab)) :   ###### ligne modifiée
        l += tab [i]
    return l
ens = [[0,1],[2,3]]
print somme ( ens )   # affiche [0,1,2,3]
print ens             # affiche [ [0,1,2,3], [2,3] ]
```

## 12.1.7 Comprendre une erreur d'exécution **

On écrit le programme suivant :

```
li  = range (0,10)
sup = [0,9]
for i in sup :
    del li [i]
print li
```

Mais il produit une erreur :

```
examen.py:44:  IndexError: list assignment index out of range
```

Que se passe-t-il ? Comment corriger le programme pour qu'il marche correctement ? Qu'affichera-t-il alors ?

### Correction

L'erreur signifie qu'on cherche à accéder à un élément de la liste mais l'indice est soit négatif, soit trop grand. Pour comprendre pourquoi cette erreur se produit, il faut suivre les modifications de la liste `li` dans la boucle `for`. Au départ, `li` vaut `range(0,10)`, soit :

```
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Lors du premier passage de la boucle, on supprime l'élément d'indice 0, la liste `li` est donc égale à :

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Et elle ne contient plus que 9 éléments au lieu de 10, donc le dernier élément est celui d'indice 8. Or lors du second passage dans la boucle `for`, on cherche à supprimer l'élément d'indice 9, ce qui est impossible. L'idée la plus simple pour corriger l'erreur est de supprimer les éléments en commençant par ordre d'indices décroissant :

```
li  = range (0,10)
sup = [9,0]      ####### ligne modifiée
for i in sup :
    del li [i]
print li
```

Mais on pouvait tout à fait écrire aussi $\texttt{sup} = [0, 8]$ même si ce n'est la réponse que je conseillerais. L'objectif du programme n'était pas de supprimer toute la liste `li`.

Une dernière précision, pour supprimer l'élément d'indice `i` de la liste `li`, on peut écrire soit `del li[i]` soit `del li[i :i+1]`.

### 12.1.8 Comprendre une erreur de logique **

Le programme suivant fonctionne mais le résultat n'est pas celui escompté.

```
l = ["un", "deux", "trois", "quatre", "cinq"]
for i in range (0,len (l)) :
    mi = i
    for j in range (i, len (l)) :
        if l[mi] < l [j] : mi = j
    e      = l [i]
    l [mi] = l [i]
    l [i]  = e
print l
```

Le résultat affiché est :

```
['un', 'deux', 'deux', 'deux', 'cinq']
```

Qu'est censé faire ce programme ? Quelle est l'erreur ?

#### Correction

Ce programme est censé effectuer un tri par ordre alphabétique **décroissant**. Le problème intervient lors de la permutation de l'élément `l[i]` avec l'élément `l[mi]`. Il faut donc écrire :

```
l = ["un", "deux", "trois", "quatre", "cinq"]
for i in range (0,len (l)) :
    mi = i
    for j in range (i, len (l)) :
        if l[mi] < l [j] : mi = j
    e      = l [mi]       ######## ligne modifiée
    l [mi] = l [i]
    l [i]  = e
print l
```

### 12.1.9 Coût d'un algorithme **

Le coût d'un algorithme ou d'un programme est le nombre d'opérations (additions, multiplications, tests, ...) qu'il effectue. Il s'exprime comme un multiple d'une fonction de la dimension des données que le programme manipule. Par exemple : $O(n)$, $O(n^2)$, $O(n \ln n)$, ...

Quel est le coût de la fonction `variance` en fonction de la longueur de la liste `tab` ? N'y a-t-il pas moyen de faire plus rapide ?

```
def moyenne (tab) :
    s = 0.0
    for x in tab :
        s += x
    return s / len (tab)

def variance (tab) :
    s = 0.0
    for x in tab :
        t  = x - moyenne (tab)
        s += t * t
    return s / len (tab)

l = [ 0,1,2, 2,3,1,3,0]
print moyenne (l)
print variance (l)
```

#### Correction

Tout d'abord, le coût d'un algorithme est très souvent exprimé comme un multiple de la dimension des données qu'il traite. Ici, la dimension est la taille du tableau `tab`. Par exemple, si on note `n` = `len(tab)`, alors le coût de la fonction `moyenne` s'écrit $O(n)$ car cette fonction fait la somme des $n$ éléments du tableau.

La fonction `variance` contient quant à elle un petit piège. Si elle contient elle aussi une boucle, chacun des $n$ passages dans cette boucle fait appel à la fonction `moyenne`. Le coût de la fonction `variance` est donc $O(n^2)$.

Il est possible d'accélérer le programme car la fonction `moyenne` retourne le même résultat à chaque passage dans la boucle. Il suffit de mémoriser son résultat dans une variable avant d'entrer dans la boucle comme suit :

```
def variance (tab) :
    s = 0.0
    m = moyenne (tab)
    for x in tab :
        t  = x - m
        s += t * t
    return s / len (tab)
```

Le coût de la fonction `variance` est alors $O(n)$.

**Remarque 12.1 : coût d'un algorithme**
Le coût d'un algorithme peut être évalué de manière plus précise et nécessiter un résultat comme $n^2 + 3n + 2$ mais cette exigence est rarement utile pour des langages comme *Python*. L'expression `for x in tab :` cache nécessairement un test qu'il

faudrait prendre en compte si plus de précision était exigée. Il faudrait également se tourner vers un autre langage de programmation, plus précis dans sa syntaxe. Par exemple, lorsqu'on conçoit un programme avec le langage C ou C++, à partir du même code informatique, on peut construire deux programmes exécutables. Le premier (ou version *debug*), lent, sert à la mise au point : il inclut des tests supplémentaires permettant de vérifier à chaque étape qu'il n'y a pas eu d'erreur (une division par zéro par exemple). Lorsqu'on est sûr que le programme marche, on construit la seconde version (ou *release*), plus rapide, dont ont été ôtés tous ces tests de conception devenus inutiles.

*Python* aboutit à un programme lent qui inclut une quantité de tests invisibles pour celui qui programme mais qui détecte les erreurs plus vite et favorise une conception rapide. Il n'est pas adapté au traitement d'information en grand nombre et fait une multitude d'opérations cachées.

### 12.1.10 Héritage **

On a besoin dans un programme de créer une classe `carre` et une classe `rectangle`. Mais on ne sait pas quelle classe doit hériter de l'autre. Dans le premier programme, `rectangle` hérite de `carre`.

```
class carre :
    def __init__ (self, a) :
        self.a = a
    def surface (self) :
        return self.a ** 2

class rectangle (carre) :
    def __init__ (self, a,b) :
        carre.__init__(self,a)
        self.b = b
    def surface (self) :
        return self.a * self.b
```

Dans le second programme, c'est la classe `carre` qui hérite de la classe `rectangle`.

```
class rectangle :
    def __init__ (self, a,b) :
        self.a = a
        self.b = b
    def surface (self) :
        return self.a * self.b

class carre (rectangle) :
    def __init__ (self, a) :
        rectangle.__init__ (self, a,a)
    def surface (self) :
        return self.a ** 2
```

**1)** Dans le second programme, est-il nécessaire de redéfinir la méthode `surface` dans la classe `carre` ? Justifiez.

**2)** Quel est le sens d'héritage qui vous paraît le plus censé, `class rectangle(carre)` ou `class carre(rectangle)` ? Justifiez.

**3)** On désire ajouter la classe `losange`. Est-il plus simple que `rectangle` hérite de la classe `carre` ou l'inverse pour introduire la classe `losange` ? Quel ou quels attributs supplémentaires faut-il introduire dans la classe `losange` ?

### Correction

**1)** Le principe de l'héritage est qu'une classe `carre` héritant de la classe `rectangle` hérite de ses attributs et méthodes. L'aire d'un carré est égale à celle d'un rectangle dont les côtés sont égaux, par conséquent, la méthode `surface` de la classe retourne la même valeur que celle de la classe `rectangle`. Il n'est donc pas nécessaire de la redéfinir.

**2)** D'après la réponse de la première question, il paraît plus logique de considérer que `carre` hérite de `rectangle`.

**3)** Un losange est défini par un côté et un angle ou un côté et la longueur d'une de ses diagonales, soit dans les deux cas, deux paramètres. Dans la première question, il paraissait plus logique que la classe la plus spécifique hérite de la classe la plus générale afin de bénéficier de ses méthodes. Pour introduire le losange, il paraît plus logique de partir du plus spécifique pour aller au plus général afin que chaque classe ne contienne que les informations qui lui sont nécessaires.

```
class carre :
    def __init__ (self, a) :
        self.a = a
    def surface (self) :
        return self.a ** 2

class rectangle (carre) :
    def __init__ (self, a,b) :
        carre.__init__(self,a)
        self.b = b
    def surface (self) :
        return self.a * self.b

class losange (carre) :
    def __init__ (self, a,theta) :
        carre.__init__(self,a)
        self.theta = theta
    def surface (self) :
        return self.a * math.cos (self.theta) * self.a * math.sin (self.theta) * 2
```

Le sens de l'héritage dépend de vos besoins. Si l'héritage porte principalement sur les méthodes, il est préférable de partir du plus général pour aller au plus spécifique. La première classe sert d'interface pour toutes ses filles. Si l'héritage porte principalement sur les attributs, il est préférable de partir du plus spécifique au plus général. Dans le cas général, il n'y a pas d'héritage plus sensé qu'un autre mais pour un problème donné, il y a souvent un héritage plus sensé qu'un autre.

### 12.1.11 Précision des calculs ***

**1)** On exécute le programme suivant :

```
x = 1.0
for i in range (0,15) :
    x = x / 10
    print i, "\t", 1.0 - x, "\t", x, "\t", x **(0.5)
```

Il affiche à l'écran le résultat suivant :

```
0    0.90000000000000002220     0.1              0.316227766017
1    0.98999999999999999112     0.01             0.1
2    0.99899999999999999911     0.001            0.0316227766017
3    0.99990000000000001101     0.0001           0.01
4    0.99999000000000004551     1e-05            0.00316227766017
5    0.99999899999999997124     1e-06            0.001
6    0.99999990000000005264     1e-07            0.000316227766017
7    0.99999998999999994975     1e-08            0.0001
8    0.99999999900000002828     1e-09            3.16227766017e-05
9    0.99999999899999999173     1e-10            1e-05
10   0.99999999998999999917     1e-11            3.16227766017e-06
11   0.99999999999900002212     1e-12            1e-06
12   0.99999999999989996891     1e-13            3.16227766017e-07
13   0.99999999999999000799     1e-14            1e-07
14   0.99999999999999900080     1e-15            3.16227766017e-08
15   0.99999999999999988898     1e-16            1e-08
16   1.00000000000000000000     1e-17            3.16227766017e-09
17   1.00000000000000000000     1e-18            1e-09
18   1.00000000000000000000     1e-19            3.16227766017e-10
19   1.00000000000000000000     1e-20            1e-10
```

Que peut-on en déduire ?

**2)** On écrit une classe `matrice_carree_2` qui représente une matrice carrée de dimension 2.

```
class matrice_carree_2 :
    def __init__ (self, a,b,c,d) :
        self.a, self.b, self.c, self.d = a,b,c,d

    def determinant (self) :
        return self.a * self.d - self.b * self.c

m1 = matrice_carree_2 (1.0,1e-6,1e-6,1.0)
m2 = matrice_carree_2 (1.0,1e-9,1e-9,1.0)
print m1.determinant ()
print m2.determinant ()
```

Qu'affichent les deux dernières lignes ?

**3)** On considère la matrice $M = \begin{pmatrix} 1 & 10^{-9} \\ 10^{-9} & 1 \end{pmatrix}$.

On pose $D = \det(M) = 1 - 10^{-18}$ et $T = tr(M) = 2$. $\Delta$ est le déterminant de $M$ et $T$ sa trace. On sait que les valeurs propres de $M$ notées $\lambda_1, \lambda_2$ vérifient :

$$\begin{aligned} D &= \lambda_1 \lambda_2 \\ T &= \lambda_1 + \lambda_2 \end{aligned}$$

On vérifie que $(x-\lambda_1)(x-\lambda_2) = x^2 - x(\lambda_1+\lambda_2) + \lambda_1\lambda_2$. Les valeurs propres de $M$ sont donc solutions de l'équation : $x^2 - Tx + D = 0$.

Le discriminant de ce polynôme est $\Delta = T^2 - 4D$. On peut donc exprimer les valeurs propres de la matrice $M$ par :

$$\begin{array}{rcl} \lambda_1 & = & \frac{T-\sqrt{\Delta}}{2} \\ \lambda_2 & = & \frac{T+\sqrt{\Delta}}{2} \end{array}$$

On ajoute donc la méthode suivante à la classe `matrice_carree_2` :

```
        def valeurs_propres (self) :
            det   = self.determinant ()
            trace = self.a + self.d
            delta = trace ** 2 - 4 * det
            l1    = 0.5 * (trace - (delta ** (0.5)) )
            l2    = 0.5 * (trace + (delta ** (0.5)) )
            return l1,l2
```

D'après la précédente question, que retourne cette méthode pour la matrice $M$ ? (à justifier)

**4)** On décompose la matrice $M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 10^{-9} \\ 10^{-9} & 0 \end{pmatrix} = I + M'$.

On peut démontrer que si $\lambda$ est une valeur propre de $M'$, alors $1+\lambda$ est une valeur propre de $M$. Que donne le calcul des valeurs propres de $M'$ si on utilise la méthode `valeurs_propres` pour ces deux matrices ?

**5)** On considère maintenant la matrice $M'' = \begin{pmatrix} 1 & 10^{-9} \\ -10^{-9} & 1 \end{pmatrix}$. En décomposant la matrice $M''$ de la même manière qu'à la question 4, quelles sont les valeurs propres retournées par le programme pour la matrice $M''$ ? Quelles sont ses vraies valeurs propres ?

### Correction

L'exercice a pour but de montrer que l'ordinateur ne fait que des calculs approchés et que la précision du résultat dépend de la méthode numérique employée.

**1)** Le programme montre que l'ordinateur affiche `1` lorsqu'il calcule $1 - 10^{-17}$. Cela signifie que la précision des calculs en *Python* est au mieux de $10^{-16}$.

**2)** Il s'agit ici de donner le résultat que calcule l'ordinateur et non le résultat théorique qui sera toujours exact. On cherche à calculer ici le déterminant des matrices $M_1$ et $M_2$ définies par :

$$M_1 = \begin{pmatrix} 1 & 10^{-6} \\ 10^{-6} & 1 \end{pmatrix} \text{ et } M_2 = \begin{pmatrix} 1 & 10^{-9} \\ 10^{-9} & 1 \end{pmatrix}$$

Or, le programme proposé calcule les déterminants comme suit :

$$\det M_1 = 1 - 10^{-12} \text{ et } \det M_2 = 1 - 10^{-18}$$

D'après les affichages du programme de la question 1, le programme de la question 2 donnera comme réponse :

```
0.99999999999900002212
1.00000000000000000000
```

La seconde valeur est donc fausse.

**3)** Le déterminant est utilisé pour calculer les valeurs propres d'une matrice. Cette question s'intéresse à la répercussion de l'approximation faite pour le déterminant sur les valeurs propres. D'après l'énoncé, les deux valeurs propres sont calculées comme étant :

```
l1    = 0.5 * (trace - ((trace ** 2 - 4 * det) ** (0.5)) )
l2    = 0.5 * (trace + ((trace ** 2 - 4 * det) ** (0.5)) )
```

Pour la matrice $M_1$, on obtient donc en remplaçant le déterminant par 0.99999999999900002212, on obtient les réponses données par le petit programme de la question 1 :

```
l1 = 1,000001
l2 = 0.99999899999999997124  # égale à 1 - 1e-6
```

Pour la matrice $M_2$, le déterminant vaut 1. En remplaçant `trace` par 2 et `det` par 1, on obtient :

```
l1 = 1
l2 = 1
```

**4)** On change la méthode de calcul pour la matrice $M_2$, on écrit que $M_2 = I + \begin{pmatrix} 0 & 10^{-9} \\ 10^{-9} & 0 \end{pmatrix} = I + M_2'$. Cette fois-ci le déterminant calculé par *Python* est bien $1e-18$. La trace de la matrice $M_2'$ est nulle, on applique les formules suivantes à la matrice $M_2'$ pour trouver :

```
l1    = 0.5 * (trace - ((trace ** 2 - 4 * det) ** (0.5)) ) = - det ** 0.5 = -1e-9
l2    = 0.5 * (trace + ((trace ** 2 - 4 * det) ** (0.5)) ) = det ** 0.5 = 1e-9
```

D'après l'énoncé, les valeurs propres de la matrice $M_2$ sont les sommes de celles de la matrice $I$ et de la matrice $M_2'$. Par conséquent, ce second calcul mène au résultat suivant :

```
l1    = 1-1e-9  = 0.99999999900000002828
l2    = 1+ 1e-9 = 1.000000001
```

**5)** La matrice $M''$ n'est en fait pas diagonalisable, c'est-à-dire que $tr\,(M'')^2 - 4 * \det M'' = 4 - 4(1 + 10^{-18}) < 0$. Or le calcul proposé par la question 3 aboutit au même résultat faux que pour la matrice $M_2$, les deux valeurs propres trouvées seront égales à 1. Si on applique la décomposition de la question 4 :

$$M'' = I + \begin{pmatrix} 0 & -10^{-9} \\ 10^{-9} & 0 \end{pmatrix} = I + N''$$

Le programme calcule sans erreur le discriminant négatif de la matrice $N''$ qui n'est pas diagonalisable. Il est donc impossible d'obtenir des valeurs propres réelles pour la matrice $M''$ avec cette seconde méthode.

Cette question montre qu'une erreur d'approximation peut rendre une matrice diagonalisable alors qu'elle ne l'est pas. Il est possible d'accroître la précision des calculs mais il faut faire appel à des modules externes [1].

## 12.2 Second énoncé

### 12.2.1 Logarithme en base deux **

On veut écrire une fonction qui retourne $n$ tel que $n$ soit le premier entier qui vérifie $2^n \geqslant k$. Cette fonction prend comme paramètre d'entrée $k$ et retourne également un entier. Elle ne devra utiliser ni logarithme ni exponentielle. On précise qu'en langage *Python*, `2**n` signifie 2 à la puissance $n$. Une boucle `for` ne semble pas indiquée dans ce cas.

#### Correction

L'indication qui préconisait d'utiliser autre chose qu'une boucle `for` ne voulait pas dire ne pas utiliser de boucle du tout mais une boucle `while`. La majorité des élèves ont réussi à trouver la fonction suivante :

```
def fonction_log2 (k) :
    n = 0
    while 2**n < k :
        n += 1
    return n
```

Même s'il est possible d'utiliser malgré tout une boucle `for` :

```
def fonction_log2 (k) :
    for i in range (0,1000) :
        if 2**i >= k :
            return i
```

Voici un exemple de fonction récursive :

```
def fonction_log2 (k) :
    if k <= 1 : return 0
    else : return fonction_log2 ((k+1)/2)+1
```

### 12.2.2 Calculer le résultat d'un programme **

On définit la fonction suivante :

```
def parcours (n) :
    i = 1
    j = 1
```

1. comme le module `GMPy`, http://code.google.com/p/gmpy/

```
    while i+j < n :
        print (i,j)
        i += 1
        j -= 1
        if j < 1 :
            j = i+j
            i = 1
```

Quelles sont les 6 lignes qu'affiche cette fonction si `n = 5` ?

**Correction**

La solution est :

```
(1,1)
(1,2)
(2,1)
(1,3)
(2,2)
(3,1)
```

Et si on continue :

```
(1,4)
(2,3)
(3,2)
(4,1)
(1,5)
(2,4)
(3,3)
...
```

Il fallait voir deux choses importantes dans l'énoncé de l'exercice, tout d'abord, les deux lignes suivantes :

```
        i += 1
        j -= 1
```

Elles signifient que lorsque $i$ augmente de 1, $j$ diminue de 1. L'autre information est que $j$ ne devient jamais inférieur à 1 :

```
        if j < 1 :
            j = i+j
            i = 1
```

Et lorsque ce cas arrive, $i$ devient égal à 1, $j$ devient égale à $i+j$, soit $i+1$ puisque la condition est vérifiée lorsque $j == 1$. Ce programme propose une façon de parcourir l'ensemble des nombres rationnels de la forme $\frac{i}{j}$.

### 12.2.3 Calculer le résultat d'un programme **

Que vaut `n` à la fin du programme suivant ? Il n'est en principe pas nécessaire d'aller jusqu'à `i = 10`.

```
def suite (n) :
    n = n ** 2 / 2 + 1
    return n

n = 3
for i in range (0,10) :
    n = suite (n) % 17
print n
```

**Correction**

Il est difficile d'arriver au bout des calculs correctement lors de cet exercice. La première chose à faire est d'identifier la relation entre $n$ et $n+1$. On rappelle que le symbole % désigne le reste d'une division entière et que / désigne une division entière car tous les nombres manipulés dans ce programme sont entiers. La fonction `suite` aboutit à $f(n) = E\left[\frac{n^2}{2}\right] + 1$ où $E[x]$ désigne la partie entière de $x$. Ajouté aux trois dernières lignes, on obtient :

$$u_{n+1} = f(u_n)\%17 = \left(E\left[\frac{n^2}{2}\right] + 1\right)\%17$$

Même si $i$ n'est pas utilisé dans la boucle `for`, celle-ci s'exécute quand même 10 fois. Pour trouver la valeur finale de $n$, on calcule donc les premiers termes comme suit :

| $i$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $u_i = n$ | 3 | 5 | 13 | 0 | 1 |
| $n^2$ | 9 | 25 | 169 | 0 | 1 |
| $y = E\left[\frac{n^2}{2}\right] + 1$ | 4 | 12 | 85 | **1** | 1 |
| $u_{i+1} = y\%17$ | 5 | 13 | 0 | 1 | 1 |

A partir de $i = 3$, la suite reste constante et égale à 1. Il n'est pas nécessaire d'aller plus loin.

### 12.2.4 Comprendre une erreur d'exécution *

Le programme suivant est incorrect.

```
def compte_lettre (s) :
    nombre = {}
    for c in s :
        nombre [c] += 1
    return nombre

print compte_lettre ( "mysteres" )
```

Il retourne l'erreur suivante :

```
KeyError: 'm'
```

**1)** Quelle est la cause de l'erreur ?

**2)** Quelle est la correction à apporter pour que cette fonction compte les lettres d'un mot ?

**3)** Qu'afficherait le programme une fois corrigé ?

**4)** La fonction `compte_lettre` est-elle réservée aux chaînes de caractères ? Voyez-vous d'autres types de données auxquels elle pourrait s'appliquer ?

### Correction

**1)** L'erreur provient de la ligne `nombre [c] += 1` pour `c` $=$' `m`', cette ligne équivaut à `nombre`[`c`] = `nombre`[`c`] + `1`. Cela signifie que `nombre`[`c`] doit exister avant l'exécution de la ligne, c'est-à-dire que le dictionnaire `nombre` doit avoir une valeur associée à la clé `c` $=$' `m`', ce qui n'est pas le cas ici.

Dans ce programme, `nombre` est un dictionnaire et non une liste (ou un tableau) et les dictionnaires acceptent des indices autres que des entiers, ils acceptent comme indice ou clé toute variable de type immuable : les nombres (entiers, réels), les caractères, les t-uples.

**2) 3)** La solution compte le nombre d'occurrences de chaque lettre dans le mot `s`.

```
def compteur (s) :
    nombre = {}
    for c in s : nombre [c] = 0        # ligne ajoutée
    for c in s :
        nombre [c] += 1
    return nombre
```

Ou encore :

```
def compteur (s) :
    nombre = {}
    for c in s :
        if c not in nombre : nombre [c] = 0       # ligne ajoutée
        nombre [c] += 1
    return nombre
```

Ces deux programmes retournent :

```
{'e': 2, 'm': 1, 's': 2, 'r': 1, 't': 1, 'y': 1}
```

La dernière solution pour ceux qui n'aiment pas les dictionnaires :

```
def compteur (s) :
    alpha = "abcdefghijklmnopqrstuvwxyz"
    nombre = [ 0 for c in alpha ]
    for c in s :
        i = alpha.index (c)
        nombre [i] += 1
    return nombre
```

Le programme retourne :

```
[0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 2, 1, 0, 0, 0, 0, 1, 0]
```

**4)** La fonction `compteur` accepte d'autres types de données à condition que les deux lignes `for c in s :` et `nombre[c]+=1` aient un sens et elles ne sont valables que si le paramètre `s` contient des éléments susceptibles de servir de clé pour le dictionnaire `s`, c'est-à-dire un autre dictionnaire, un tuple, une liste d'éléments de type immuable. Autrement dit, les lignes suivantes sont correctes :

```
print compteur ( "mysteres" )
print compteur ( compteur ("mysteres") )
print compteur ( [0,1,1,4,-1, (6,0), 5.5, "ch"] )
print compteur ( { 1:1, 2:2, 1:[] } )
```

Mais pas celle-ci :

```
print compteur ( [0, [0,0] ] )
```

Toutefois, cette dernière ligne est valide si la fonction `compteur` se contente seulement de compter le nombre d'éléments, c'est-à-dire la première solution citée aux questions 2) et 3).

### 12.2.5 Comprendre une erreur de logique ***

On précise que l'instruction `random.randint(0,1)` retourne un nombre aléatoire choisi dans l'ensemble $\{0,1\}$ avec des probabilités équivalentes ($\mathbb{P}(X=0) = \frac{1}{2}$ et $\mathbb{P}(X=1) = \frac{1}{2}$). La fonction `ligne_nulle` doit compter le nombre de lignes nulles de la matrice `mat` donnée comme paramètre.

```
def ligne_nulle (mat) :
    nb = 0
    for i in range (0, len (mat)) :
        lig = 0
        for j in range (0, len (mat [i])) :
            if mat [i][j] > 0 : lig += 1
            if lig == 0 : nb += 1
    return nb

matri = [ [ random.randint (0,1) for i in range (0,4) ] for j in range (0,20) ]
print ligne_nulle (matri)
```

Après avoir exécuté le programme trois fois de suite, les résultats affichés sont successivement `15, 19, 17`. Bien que l'exécution du programme ne provoque aucune erreur, le concepteur de la fonction s'interroge quand même sur ces résultats.

**1)** Sachant que chaque case de la matrice reçoit aléatoirement 0 ou 1, quelle est la probabilité qu'une ligne soit nulle ? Quelle est la probabilité d'avoir 15 lignes nulles dans la matrice sachant que cette matrice a 20 lignes ? Ces deux réponses peuvent être littérales. Donnez-vous raison à celui qui a écrit le programme (il pense s'être trompé) ?

**2)** Si vous lui donnez raison, ce qui est fort probable, où serait son erreur ?

#### Correction

**1)** La probabilité d'avoir une ligne nulle est la probabilité d'avoir 4 zéros, c'est donc :

$$\mathbb{P}(\textit{ligne nulle}) = \left(\frac{1}{2}\right)^4 = \frac{1}{16}$$

Comment calculer la probabilité d'avoir 15 lignes nulles parmi 20 ? Cela revient à estimer la probabilité de tirer 15 fois sur 20 une boule blanche lors d'un tirage à

remise sachant que dans l'urne, il y a 1 boule blanche et 15 boules noires. Le nombre de boules blanches tirées suit une loi binomiale de paramètre $p = \frac{1}{16}$. On en déduit que :

$$\mathbb{P}\left(15 \textit{ lignes nulles sur } 20\right) = C_{20}^{15} \left(\frac{1}{16}\right)^{15} \left(1 - \frac{1}{16}\right)^{5} \leqslant \frac{2^{20}}{10^{15}}$$

Cette probabilité est très faible, il est donc presque impossible d'obtenir trois fois de suite un nombre de lignes supérieur à 15. Le programme est sans aucun doute faux.

**2)** La construction de la matrice est manifestement correcte, c'est donc le comptage des lignes nulles qui est faux. Cette erreur intervient lors de la ligne `if lig == 0 : nb += 1`. Celle-ci est incluse dans la seconde boucle `for` ce qui a pour effet d'incrémenter `lig` dès qu'un premier zéro est rencontré sur une ligne. Au final, la fonction retourne le nombre de lignes contenant au moins un zéro. Voici donc la correction à apporter :

```
def ligne_nulle (mat) :
    nb = 0
    for i in range (0, len (mat)) :
        lig = 0
        for j in range (0, len (mat [i])) :
            if mat [i][j] > 0 : lig += 1
        if lig == 0 : nb += 1                    # ligne décalée vers la gauche
    return nb
```

### 12.2.6 Récursivité *

Le programme suivant provoque une erreur dont le message paraît sans fin.

```
class erreur :
    def __init__ (self) :
        self.e = erreur ()
e = erreur ()
```

Auriez-vous une explication pour ce qui suit ?

```
Traceback (most recent call last):
  File "examen2007.py", line 4, in ?
    e = erreur ()
  File "examen2007.py", line 2, in __init__
    self.e = erreur ()
  File "examen2007.py", line 2, in __init__
    self.e = erreur ()
  File "examen2007.py", line 2, in __init__
    self.e = erreur ()
  ...
```

#### Correction

L'erreur retournée est une erreur d'exécution et non une erreur de syntaxe. Cela veut dire que le programme de l'exercice est syntaxiquement correct. Il n'est donc pas possible de dire que l'attribut `e` n'est pas défini et de corriger le programme comme suit pour expliquer l'erreur :

```
class erreur :
    def __init__ (self,e) :
        self.e = e
```

En fait, la classe `erreur` définit un attribut qui est également de type `erreur`. Cet attribut va lui aussi définir un attribut de type `erreur`. Ce schéma va se reproduire à l'infini puisqu'à aucun moment, le code du programme ne prévoit la possibilité de s'arrêter. Le programme crée donc des instances de la classe `erreur` à l'infini jusqu'à atteindre une certaine limite dépendant du langage *Python*. C'est à ce moment-là que se produit l'erreur citée dans l'énoncé.

### 12.2.7 Compléter un programme **

11 (base décimale) s'écrit 102 en base 3 puisque $11 = 1 * 3^2 + 0 * 3^1 + 2 * 3^0$. L'objectif de cet exercice est d'écrire une fonction qui écrit un nombre entier en base 3. Cette fonction prend comme paramètre d'entrée un entier et retourne une chaîne de caractères. On précise que l'opérateur % calcule le reste d'une division entière et que `11/3` → `3` car c'est une division entière dont le résultat est un entier égal au quotient de la division. La fonction `str` permet de convertir un nombre en une chaîne de caractères. Il ne reste plus qu'à compléter les deux lignes manquantes du programme suivant :

```
def base3 (n) :
    s = ""
    while n > 0 :
        r = n % 3
        # ....... à compléter
        # ....... à compléter
    return s
```

#### Correction

Il y a effectivement deux lignes à corriger. `r` désigne le reste de la division de `n` par 3. Il faut le convertir en chaîne de caractères et l'ajouter à gauche à la chaîne `s`. On passe au chiffre suivant en division `n` par 3 ce qui est rassurant puisque `n` va tendre vers zéro et le programme s'arrêter nécessairement au bout d'un moment. Pour vérifier que ce programme est correct, il suffit de l'appliquer à un nombre, voire appliquer le même algorithme mais en base 10 pour être vraiment sûr.

```
def base3 (n) :
    s = ""
    while n > 0 :
        r = n % 3
        s = str (r) + s
        n = n / 3           # équivalent à n = (n-r) / 3
                            # puisque / est une division entière
    return s
```

### 12.2.8 Comprendre une erreur de logique ***

Un professeur désireux de tester une répartition aléatoire des notes à un examen décide de tirer au hasard les notes de ses élèves selon une loi normale de moyenne 15 et d'écart-type 3. Il arrondit ses notes à l'entier le plus proche en n'omettant pas de vérifier que ses notes sont bien dans l'intervalle $[0, 20]$. On précise que l'instruction `float(i)` convertit un nombre `i` en nombre réel, cette conversion est utilisée pour être sûr que le résultat final sera bien réel et non le résultat d'opérations sur des entiers. Cette conversion intervient le plus souvent lors de divisions.

```
import copy
import math
import random

class Eleve :
    def __init__ (self, note) :
        self.note = note

e = Eleve (0)
l = []
for i in range (0,81) :
    e.note = int (random.gauss (15, 3) + 0.5)  # tirage aélatoire et arrondi
    if e.note >= 20 : e.note = 20              # pas de note au-dessus de 20
    if e.note < 0 : e.note = 0                 # pas de note négative
    l.append (e)

moy = 0
var = 0

for e in l :
    moy += e.note
moy = float (moy) / len (l)  # les notes sont entières,
                             # il faut convertir avant de diviser
                             # pour obtenir la moyenne
for e in l :
    var += (e.note - moy) ** 2
var = math.sqrt ( float (var) ) / len (l)

print "moyenne ", moy
print "écart-type ", var
```

Il songe à vérifier néanmoins que la moyenne de ses notes arrondies est bien conforme à ce qu'il a échafaudé.

```
moyenne  16.0
écart-type  0.0
```

La moyenne égale à 16 ne le perturbe guère, il se dit que l'arrondi a été plutôt généreux. Toutefois l'écart-type nul le laisse perplexe.

**1)** Que signifie un écart-type nul ? Quelle est l'erreur du professeur ? (Elle est située à l'intérieur de la boucle `for i in range(0,81)` :, ce n'est pas une erreur lors du calcul de l'écart-type ni une erreur de définition de la classe `Eleve`.)

**2)** Proposer deux solutions pour corriger ce problème, chacune d'elles revient à remplacer la ligne `l.append(e)`.

**3)** On sait que la variance d'une variable aléatoire $X$ vérifie : $\mathbb{V}(X) = \mathbb{E}\left(X^2\right) - [\mathbb{E}(X)]^2$. Cette astuce mathématique permet-elle de réduire le nombre de boucles du programme, si oui, comment ?

### Correction

**1)** Un écart-type nul signifie que toutes les notes sont identiques et égales à la moyenne. Selon le programme, tous les élèves ont donc 16. L'erreur provient du fait que l'instruction `l.append(e)` ajoute à chaque fois la même variable de type `Eleve`. A la fin de la boucle, la liste `l` contient 81 fois le même objet `Eleve` ou plus exactement 81 fois la même instance de la classe `Eleve`. On modifie la note de cet unique objet en écrivant : `e.note = int(random.gauss(15, 3) + 0.5)`. 16 est donc la note attribuée au dernier élève, la dernière note tirée aléatoirement.

**2)** Les deux corrections possibles consistent à créer à chaque itération une nouvelle instance de la classe `Eleve`.

1. `l.append(e)` devient `l.append(Eleve(e.note))`
2. `l.append(e)` devient `l.append(copy.copy(e))`

La variable `e` dans la boucle est un `Eleve` temporaire, il est ensuite recréé ou recopié avant l'ajout dans la liste. Ceci veut dire que l'instance ajoutée dans la liste n'est pas la même que celle utilisée dans la boucle.

Une troisième solution est envisageable même si elle introduit des modifications dans la suite du programme, elle est logiquement correcte : `l.append(e)` devient `l.append(e.note)`. La liste `l` n'est plus une liste de `Eleve` mais une liste d'entiers. Le programme devient :

```
#...
e = Eleve (0)
l = []
for i in range (0,81) :
    e.note = int (random.gauss (15, 3) + 0.5)
    if e.note >= 20 : e.note = 20
    if e.note < 0 : e.note = 0
    l.append (e.note)                           # ligne modifiée

moy = 0
var = 0

for note in l :                                 # ligne modifiée
    moy += note                                 # ligne modifiée
moy = float (moy) / len (l)
for note in l :                                 # ligne modifiée
    var += (note - moy) ** 2                    # ligne modifiée
var = math.sqrt ( float (var) ) / len (l)

print "moyenne ", moy
print "écart-type ", var
```

**3)** La formule mathématique permet de réduire le nombre de boucles à deux. Lors de la version initiale du programme, la première sert à créer la liste `l`, la seconde

à calculer la moyenne, la troisième à calculer la variance. On regroupe les deux dernières boucles en une seule.

```
moy = 0
var = 0

for note in l :
    moy += note
    var += note * note
moy = float (moy) / len (l)
var = float (var) / len (l)
var = var - moy * moy
var = math.sqrt ( float (var) )

print "moyenne ", moy
print "écart-type ", var
```

## 12.3 Troisème énoncé

### 12.3.1 Calcul d'un arrondi *

Ecrire une fonction qui arrondit un nombre réel à 0.5 près ? Ecrire une autre fonction qui arrondit à 0.125 près ? Ecrire une fonction qui arrondit à $r$ près où $r$ est un réel ? Répondre uniquement à cette dernière question suffit pour répondre aux trois questions.

#### Correction

Pour arrondi à 1 près, il suffit de prendre la partie entière de $x + 0.5$, arrondir $x$ à 0.5 près revient à arrondi $2x$ à 1 près.

```
def arrondi_05 (x) :
    return float (int (x * 2 + 0.5)) / 2

def arrondi_0125 (x) :
    return float (int (x * 8 + 0.5)) / 8

def arrondi (x, p) :
    return float (int (x / p + 0.5)) * p
```

### 12.3.2 Même programme avec 1,2,3 boucles **

Le programme suivant affiche toutes les listes de trois entiers, chaque entier étant compris entre 0 et 9.

```
for a in range (0, 10) :
    for b in range (0, 10) :
        for c in range (0, 10) :
             print [a,b,c]
```

**1)** Proposez une solution avec une boucle `while` et deux tests `if`.

**2)** Proposez une solution avec deux boucles `while`.

**Correction**

**1)** Une seule boucle contrôle les indices $a$, $b$, $c$. Quand un indice atteint sa limite, on incrémente le suivant et on remet l'indice à 0.

```
a,b,c = 0,0,0
while c < 10 :
    print [a,b,c]
    a += 1
    if a == 10 :
        b += 1
        a = 0
        if b == 10 :
            c += 1
            b = 1
```

**2)** L'avantage de cette dernière solution est qu'elle ne dépend pas du nombre d'indices. C'est cette solution qu'il faut préconiser pour écrire une fonction dont le code est adapté quelque soit la valeur de $n$.

```
l = [0,0,0]
while l [-1] < 10 :
    print l
    l [0] += 1
    i = 0
    while i < len (l)-1 and l [i] == 10 :
        l [i] = 0
        l [i+1] += 1
        i += 1
```

Ce problème était mal posé : il n'est pas difficile d'introduire des tests ou des boucles redondantes ou d'enlever une des conditions d'une boucle `while` pour la remplacer un test relié à une sortie de la boucle.

### 12.3.3 Suite récurrente (Fibonacci) **

La fonction `fibo` retourne la valeur de la suite de Fibonacci pour tout entier `n`. Quel est son coût en fonction de $O(n)$ ?

```
def fibo (n) :
    if n <= 2 : return 2
    else : return fibo (n-1) + fibo (n-2)
```

**Correction**

Lorsqu'on cherche à calculer `fibo(n)`, on calcule `fibo(n - 1)` et `fibo(n - 2)` : le coût du calcul `fibo(n)` est égal à la somme des coûts des calculs de `fibo(n - 1)` et `fibo(n - 2)` plus une addition et un test. Le coût de la fonction `fibo(n)` est plus facile à définir par récurrence. Le coût $c_n$ du calcul de `fibo(n)` vérifie donc :

$$c_0 = c_1 = c_2 = 1 \tag{12.1}$$

$$c_n = c_{n-1} + c_{n-2} + 2 \tag{12.2}$$

Le terme 1 dans (12.1) correspond au premier test. Le terme 2 dans (12.2) correspond au test et à l'addition. Le coût du calcul de `fibo(n)` est égal à une constante près à une suite de Fibonacci. Le code suivant permet de le vérifier en introduisant une variable globale. On modifie également la fonction `fibo` de façon à pouvoir changer $u_0$ et $u_1$.

```
nb = 0  # variable globale
def fibo (n,p) :
    global nb
    if n <= 2 :
        nb += 1
        return p # plus de récurrence
    else :
        nb += 2
        return fibo (n-1,p) + fibo (n-2,p)

for n in range (1, 20) :
    nb = 0    # remis à zéro, à chaque fois
              # nb est la mesure du coût
    print fibo(n,3)-2, nb  # nombres identiques
    # nb vérifie la récurrence de la suite c(n)
    #          c(n) = c(n-1) + c(n-2) + 2
```

suites de la forme (12.3) :

$$
\begin{aligned}
u_0 &= p \\
u_1 &= p \\
u_2 &= u_0 + u_1 + d = 2p + d \\
u_3 &= u_1 + u_2 + d = 3p + 2d \\
u_4 &= u_2 + u_3 + d = 5p + 4d \\
u_5 &= u_3 + u_4 + d = 8p + 7d \\
u_6 &= u_4 + u_5 + d = 13p + 12d \\
u_7 &= u_5 + u_6 + d = 21p + 20d \\
u_8 &= u_6 + u_7 + d = 34p + 33d \\
&\dots
\end{aligned}
$$

Dans cet exemple, on s'aperçoit que la suite est égale à son coût. Pour le démontrer d'une façon plus théorique, on s'intéresse aux suites de la forme suivante dont la récurrence est développée ci-dessus à droite :

$$
\begin{aligned}
u_0 &= u_1 = u_2 = p \\
u_n &= u_{n-1} + u_{n-2} + d
\end{aligned}
\tag{12.3}
$$

Par une astuce de calcul, on peut réduire l'écriture de la suite $(u_n)$ à une somme linéaire de deux suites $(U_n)$ et $(V_n)$. La suite $(U_n)$ est définie par $U_0 = 1$, $U_1 = 1$, $U_n = U_{n-1} + U_{n-2}$ et la suite $(V_n)$ définie par $V_0 = 0$, $V_1 = 0$, $V_n = V_{n-1} + V_{n-2} + 1$. On en déduit que :

$$u_n = U_n p + V_n d$$

Si on arrive à montrer que $U_n = V_n + 1$, cela montrera que la fonction `fibo` citée dans la correction est bien égale à son coût. Or :

$$V_n + 1 = V_{n-1} + V_{n-2} + 1 + 1 = (V_{n-1} + 1) + (V_{n-2} + 1)$$

La suite $V_n^* = V_n + 1$ suit bien la récurrence de la suite $U_n$. On vérifie que les premières valeurs sont identiques et cela montre que $U_n = V_n + 1$. On en déduit que :

$$u_n = U_n p + (U_n - 1)d = U_n(p + d) - d$$

Pour conclure, une suite de Fibonacci vérifie la récurrence $u_n = u_{n-1} + u_{n-2}$. Si on pose $\lambda_1 = \frac{1+\sqrt{5}}{2}$ et $\lambda_2 = \frac{1-\sqrt{5}}{2}$. Cette suite peut s'écrire sous la forme $u_n = A\lambda_1^n + B\lambda_2^n$. Le second terme étant négligeable par rapport au premier, le coût de la fonction `fibo` est donc en $O(\lambda_1^n)$. En d'autres termes, si on calcule le coût du calcul récursif d'une suite de Fibonacci, ce dernier est équivalent à la suite elle-même.

### 12.3.4 Calculer le résultat d'un programme *

**1)** Qu'affiche le code suivant :

```
l = [0,1,2,3,4,5]
g = l
for i in range (0, len (l)-1) :
    g [i] = g [i+1]
print l
print g
```

**2)** Et celui-ci :

```
l = [0,1,2,3,4,5]
g = [0,1,2,3,4,5]
for i in range (0, len (l)-1) :
    g [i] = g [i+1]
print l
print g
```

**3)** Et encore celui-là :

```
l = [0,1,2,3,4,5]
g = [0,1,2,3,4,5]
for i in range (0, len (l)) :
    g [i] = g [(i+1)%len (l)]
print l
print g
```

**4)** A votre avis, quel est l'objectif du programme et que suggérez-vous d'écrire ?

**5)** Voyez-vous un moyen d'écrire plus simplement la seconde ligne `g` $= [0, 1, 2, 3, 4, 5]$ tout en laissant inchangé le résultat ?

#### Correction

**1)** L'instruction `g = l` implique que ces deux variables désignent la même liste. La boucle décale les nombres vers la gauche.

```
[1, 2, 3, 4, 5, 5]
[1, 2, 3, 4, 5, 5]
```

**2)** L'instruction `g` $= [0, 1, 2, 3, 4, 5]$ implique que ces deux variables ne désignent plus la même liste. L'instruction `print l` affiche le contenu du début.

```
[0, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 5]
```

**3)** La boucle s'intéresse cette fois-ci au déplacement du premier élément en première position. Le programmeur a pris soin d'utiliser le modulo, $n\%n = 0$ mais le premier élément de la liste $g$ est devenu le second. Le résultat est donc :

```
[0, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1]
```

**4)** Le programme souhaite décaler les éléments d'une liste vers la gauche, le premier élément devenant le dernier.

```
l = [0,1,2,3,4,5]
g = [0,1,2,3,4,5]
for i in range (0, len (l)) :
    g [i] = l [(i+1)%len (l)] # ligne modifiée, g devient l
print l
print g
```

**5)** La seconde ligne impose de répéter le contenu de la liste. Il existe une fonction qui permet de le faire :

```
import copy
l = [0,1,2,3,4,5]
g = copy.copy (l) # on pourrait aussi écrire g = list (l)
                  # ou encore g = [ i for i in l ]
for i in range (0, len (l)) :
    g [i] = l [(i+1)%len (l)]
print l
print g
```

### 12.3.5 Comprendre une erreur de logique **

**1)** Un individu a écrit la fonction suivante, il inclut un seul commentaire et il faut deviner ce qu'elle fait. Après exécution, le programme affiche 4.

```
def mystere (l) :
    """cette fonction s'applique à des listes de nombres"""
    l.sort ()
    nb = 0
    for i in range (1,len (l)) :
        if l [i-1] != l [i] : nb += 1
    return nb+1

l = [4,3,1,2,3,4]
print mystere (l)  # affiche 4
```

**2)** Après avoir écrit l'instruction `print l`, on s'aperçoit que la liste `l` a été modifiée par la fonction `mystere`. Qu'affiche cette instruction ?

**3)** Comment corriger le programme pour que la liste ne soit pas modifiée ?

#### Correction

**1)** La fonction travaille sur une liste de nombres triés. La fonction parcourt cette liste et compte le nombre de fois que la liste triée contient deux éléments successifs différents. La fonction retourne donc le nombre d'éléments distincts d'une liste.

**2)** La fonction trie la liste, elle ressort donc triée de la fonction car le passage des listes à une fonction s'effectue par adresse. Le résultat de l'instruction `print l` est donc :

```
[1, 2, 3, 3, 4, 4]
```

**3)** Il suffit d'utiliser le module `copy`.

```
import copy
def mystere (l) :
    """cette fonction s'applique à des listes de nombres"""
    l = copy.copy (l) # ligne insérée
                      # on peut écrire aussi l = list (l)
    l.sort ()
    nb = 0
    for i in range (1,len (l)) :
        if l [i-1] != l [i] : nb += 1
    return nb+1
```

### 12.3.6 Héritage **

```
class Personne :
    def __init__ (self, nom) :
        self.nom = nom
    def entete (self) :
        return ""
    def __str__ (self) :
        s = self.entete () + self.nom
        return s

class Homme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "M. "

class Femme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "Melle "

h = Homme ("Hector")
f = Femme ("Gertrude")
print h
print f
```

*Programme A*

```
class Personne :
    def __init__ (self, nom, entete) :
        self.nom = nom
        self.entete = entete
    def __str__ (self) :
        s = self.entete + self.nom
        return s

h = Personne ("Hector", "M. ")
f = Personne ("Gertrude", "Melle ")
print h
print f
```

*Programme B*

**1)** Les deux programmes précédents affichent-ils les mêmes choses ? Qu'affichent-ils ?

**2)** On souhaite ajouter une personne hermaphrodite, comment modifier chacun des deux programmes pour prendre en compte ce cas ?

**3)** Quel programme conseilleriez-vous à quelqu'un qui doit manipuler cent millions de personnes et qui a peur de manquer de mémoire ? Justifiez.

#### Correction

**1)** Les programmes affichent la même chose et ils affichent :

```
M. Hector
Melle Gertrude
```

**2)**

```
class Personne :
    def __init__ (self, nom) :
        self.nom = nom
    def entete (self) :
        return ""
    def __str__ (self) :
        s = self.entete () + self.nom
        return s

class Homme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "M. "

class Femme (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "Melle "

class Hermaphrodite (Personne) :
    def __init__ (self, nom) :
        Personne.__init__ (self, nom)
    def entete (self) :
        return "Melle et M. "

h = Homme ("Hector")
f = Femme ("Gertrude")
g = Hermaphrodite ("Marie-Jean")
print h
print f
print g
```

*Programme A*

```
class Personne :
    def __init__ (self, nom, entete) :
        self.nom = nom
        self.entete = entete
    def __str__ (self) :
        s = self.entete + self.nom
        return s

h = Personne ("Hector", "M. ")
f = Personne ("Gertrude", "Melle ")
g = Personne ("Marie-Jean", \
                    "Melle et M. ")
print h
print f
print g
```

*Programme B*

**3)** Les deux programmes définissent une personne avec un prénom et un en-tête. Dans le second programme, l'en-tête est un attribut de la classe et cette classe permet de modéliser les hommes et les femmes. Dans le premier programme, l'en-tête est défini par le type de la classe utilisée. Il y a moins d'information mémorisée, en contre partie, il n'est pas possible d'avoir d'autres en-têtes que `M.` et `Melle`. Pour manipuler 100 millions de personnes, il vaut mieux utiliser le premier programme avec deux classes, il sera moins gourmant en mémoire.

### 12.3.7 Analyser un programme ***

On considère la fonction suivante qui prend comme entrée une liste d'entiers. Elle ne retourne pas de résultat car elle modifie la liste. Appeler cette fonction modifie la liste donnée comme paramètre. Elle est composée de quatre groupes d'instructions.

```
def tri_entiers(l):
    """cette fonction s'applique à une liste d'entiers"""

    # groupe 1
```

```
m = l [0]
M = l [0]
for k in range(1,len(l)):
    if l [k] < m : m = l [k]
    if l [k] > M : M = l [k]

# groupe 2
p = [0 for i in range (m,M+1) ]
for i in range (0, len (l)) :
    p [ l [i] - m ] += 1

# groupe 3
R     = [0 for i in range (m,M+1) ]
R [0] = p [0]
for k in range (1, len (p)) :
    R [k] = R [k-1] + p [k]

# groupe 4
pos = 0
for i in range (1, len (l)) :
    while R [pos] < i : pos += 1
    l [i-1] = pos + m
l [len (l)-1] = M
```

**1)** Que fait le groupe 1 ?

**2)** Que fait le groupe 2 ?

**3)** Que fait le groupe 3 ?

**4)** Que fait le groupe 4 ?

**5)** Quel est la boucle contenant le plus grand nombre d'itérations ?

### Correction

**1)** Le groupe 1 détermine les minimum et maximum de la liste `l`.

**2)** Dans le groupe 2, la liste `p` compte le nombre d'occurrences d'un élément dans la liste `l`. `p[0]` correspond au nombre de fois que le minimum est présent dans la liste, `p[len(l) - 1]` correspond au nombre de fois que le maximum est présent.

**3)** Le groupe 3 construit la fonction de répartition de la liste `l`. Si `m` est le minimum de la liste `l`, `R[i]` est le nombre d'éléments inférieurs ou égaux à `i + m` que la liste contient.

**4)** Le dernier groupe trie la liste `l`. Si $X$ est une variable aléatoire de fonction de répartition $F$ alors $F(X)$ suit une loi uniforme sur $[0, 1]$. C'est le même principe ici. La figure 12.1 illustre la fonction de répartition. A chaque marche correspond un élément de la liste et à chaque marche correspond une case du tableau `R`. En parcourant les marches du tableau `R`, on retrouve les éléments de la liste `l` triés. La marche plus haute traite le cas de plusieurs éléments égaux.

**5)** Le groupe 1 inclut une boucle de $n$ itérations où $n$ est le nombre d'éléments de la liste `l`. Les groupes 2 et 3 incluent une boucle de $M - n$ itérations. Le groupe 4 inclut deux boucles, la première inclut $n$ itérations, la seconde implique une variable `pos` qui passe de $m$ à $M$. Le coût de ce bloc est en $O(n + M - n)$. C'est ce dernier bloc le plus long.

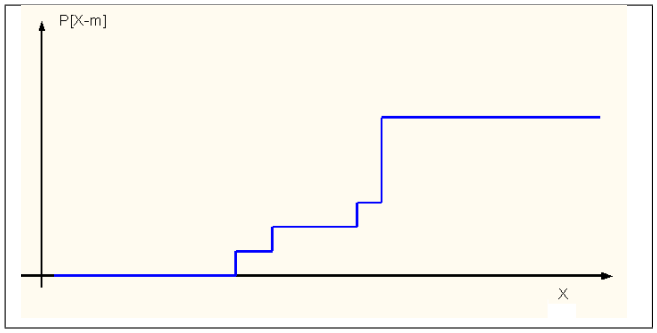



**Figure 12.1 :** *Construction de la fonction de répartition.*

### 12.3.8 Comprendre une erreur de logique ***

Chaque année, une société reçoit des cadeaux de ses clients et son patron souhaite organiser une tombola pour attribuer ces cadeaux de sorte que les plus bas salaires aient plus de chance de recevoir un cadeau. Il a donc choisi la méthode suivante :

- Chaque employé considère son salaire net par mois et le divise par 1000, il obtient un nombre entier $n_i$.
- Dans une urne, on place des boules numérotées que l'on tire avec remise.
- Le premier employé dont la boule est sortie autant de fois que $n_i$ gagne le lot.

La société a distribué plus de 100 cadeaux en quelques années mais jamais le patron n'en a gagné un seul, son salaire n'est pourtant que cinq fois celui de sa sécrétaire. Son système est-il équitable ? Il fait donc une simulation sur 1000 cadeaux en *Python* que voici. Sa société emploie quatre personnes dont les salaires sont $[2000, 3000, 5000, 10000]$.

```
import random

def tirage (poids) :
    nb = [ 0 for p in poids ]
    while True :
        i = random.randint (0, len (poids)-1)
        nb [i] += 1
        if nb [i] == poids [i] :
            return i

salaire = [ 10000, 5000, 3000, 2000 ]
poids   = [ int (s / 1000) for s in salaire ]
nombre  = [ 0 for s in salaire ]

for n in range (0,1000) :
    p = tirage (poids)
    nombre [p] += 1

for i in range (0, len (poids)) :
    print "salaire ", salaire [i], " : nb : ", nombre [i]
```

Les résultats sont plus que probants :

```
salaire  10000  : nb :  0
salaire  5000  : nb :  49
```

```
salaire  3000  : nb :  301
salaire  2000  : nb :  650
```

Il n'a aucune chance de gagner à moins que son programme soit faux. Ne s'en sortant plus, il décide d'engager un expert car il trouve particulièrement injuste que sa secrétaire ait encore gagné cette dernière bouteille de vin.

**1)** Avez-vous suffisamment confiance en vos connaissances en *Python* et en probabilités pour déclarer le programme correct ou incorrect ? Deux ou trois mots d'explications seraient les bienvenus.

**2)** Le patron, après une folle nuit de réflexion, décida de se débarrasser de son expert perdu dans des probabilités compliquées et modifia son programme pour obtenir le suivant :

```
import random

def tirage (poids, nb) :
    while True :
        i = random.randint (0, len (poids)-1)
        nb [i] += 1
        if nb [i] % poids [i]  == 0 :
            return i

salaire = [ 10000, 5000, 3000, 2000 ]
poids   = [ int (s / 1000) for s in salaire ]
nombre  = [ 0 for s in salaire ]
temp    = [ 0 for s in salaire ]

for n in range (0,1000) :
    p = tirage (poids, temp)
    nombre [p] += 1

for i in range (0, len (poids)) :
    print "salaire ", salaire [i], " : nb : ", nombre [i]
```

Et le résultat est beaucoup plus conforme à ses attentes :

```
salaire  10000  : nb :  90
salaire  5000  : nb :  178
salaire  3000  : nb :  303
salaire  2000  : nb :  429
```

Quelle est la modification dans les règles qu'il a apportée ? Vous paraissent-elles justes ?

### Correction

**1)** Le programme est correct, ses résultats aussi. Prenons le cas simple où il n'y a que le patron et sa sécrétaire. Le patron gagne 10000 euros et sa secrétaire seulement 1000 euros. On reformule le problème. On construit ensuite une suite de 10 nombres choisis uniformément au hasard dans l'ensemble $\{0, 1\}$. Le patron gagne si les 10 nombres sont les siens. Ce jeu est équivalent à celui décrit dans l'énoncé à ceci près qu'on continue le tirage même si quelqu'un gagne. Et avec cette nouvelle présentation, on peut conclure qu'il y a exactement $2^{10}$ tirages possibles et

qu'il n'y a qu'un seul tirage gagnant pour le patron. La probabilité de gagner est seulement de $2^{-10}$ et non $\frac{1}{10}$ comme le patron le souhaitait.

Si on revient au jeu à quatre personnes et les salaires de l'énoncé, la probabilité de gagner du patron est cette fois de $4^{-10}$, celle de sa sécrétaire est au moins supérieure à $\frac{1}{16}$ qui correspond à sa probabilité de gagner en deux tirages. Si le patron ne gagne pas, c'est que sa probabilité de gagner est beaucoup trop petite par rapport au nombre de tirages. Il en faudrait au moins $4^{-10}$ pour avoir une probabilité non négligeable que le patron gagne au moins une fois.

**2)** $90 * 5 = 450$ qui n'est pas très loin de 429 ou encore $178/2 * 5 = 445$. La secrétaire paraît avoir 5 fois plus de chances de gagner que son patron et 5/2 fois plus que la personne payée 5000 euros.

La fonction `tirage` reçoit un nouvel élément `nb` qui est un compteur. Le programme ne remet plus à zéro ce compteur entre deux tirages. Cela signifie que si la boule du patron est sortie trois fois alors que la secrétaire a gagné. Lors du prochain tirage, le compteur du patron partira de 3 et celui de sa secrétaire de 0.

Si on ne remet jamais les compteurs à zéro, au bout de 1000 tirages (et non 1000 cadeaux), la boule du patron est sortie environ autant de fois que celle de la secrétaire, soit 250 fois puisqu'il y a quatre employés. La secrétaire aura gagné $\frac{250}{2} = 125$ fois, le patron aura gagné $\frac{250}{10} = 25$ soit 5 fois moins [2].

## 12.4 Quatrième énoncé

### 12.4.1 Suite récurrente (Fibonacci) *

Réécrire la fonction `u` de façon à ce qu'elle ne soit plus récurrente.

```
def u (n) :
    if n <= 2 : return 1
    else : return u (n-1) + u (n-2) + u (n-3)
```

**Correction**

```
def u_non_recursif (n) :
    if n <= 2 : return 1
    u0 = 1
    u1 = 1
    u2 = 1
    i = 3
    while i <= n :
        u  = u0 + u1 + u2
        u0 = u1
        u1 = u2
        u2 = u
        i += 1
    return u
```

2. Je cite ici la plus belle des réponses retournée par un élève : *Les patrons qui organisent des tombolas sont toujours mariés à leur secrétaire donc la question ne se pose pas.*

### 12.4.2 Calculer le résultat d'un programme *

**1)** Qu'affiche le programme suivant :

```
def fonction (n) :
    return n + (n % 2)

print fonction (10)
print fonction (11)
```

**2)** Que fait la fonction `fonction` ?

**3)** Ecrire une fonction qui retourne le premier multiple de 3 supérieur à `n` ?

**Correction**

**1)** La fonction `fonction` ajoute 1 si $n$ est impair, 0 sinon. Le programme affiche :

```
10
12
```

**2)** La fonction retourne le plus petit entier pair supérieur ou égal à $n$.

**3)** On cherche à retourner le plus petit multiple de 3 supérieur ou égal à $n$. Tout d'abord, on remarque que $n + (n\%3)$ n'est pas la réponse cherchée car $4 + 4\%3 = 5$ qui n'est pas divisible par 3. Les fonctions les plus simples sont les suivantes :

```
def fonction3 (n) :
    k = 0
    while k < n : k += 3
    return k
```

Ou encore :

```
def fonction3 (n) :
    if   n % 3 == 0 : return n
    elif n % 3 == 1 : return n + 2
    else : return n + 1
```

Ou encore :

```
def fonction3 (n) :
    if n % 3 == 0 : return n
    else : return n + 3 - (n % 3)
```

### 12.4.3 Calculer le résultat d'un programme *

**1)** Qu'affiche le programme suivant :

```
def division (n) :
    return n / 2

print division (1)
print division (0.9)
```

**2)** Proposez une solution pour que le résultat de la fonction soit 0.5 lors d'une instruction `print division`... ?

**Correction**

**1)** 1/2 est égal à zéro en *Python* car c'est une division de deux entiers et le résultat est égal à la partie entière. La seconde division est une division entre un réel et un entier, le résultat est réel. Le programme affiche :

```
0
0.45
```

**2)** Voici deux solutions `print division(1.0)` ou `print division(float(1))`. Il est également possible de convertir `n` à l'intérieur de la fonction `division`.

### 12.4.4 Ecrire un programme à partir d'un algorithme **

On considère deux listes d'entiers. La première est inférieure à la seconde si l'une des deux conditions suivantes est vérifiée :

- les $n$ premiers nombres sont égaux mais la première liste ne contient que $n$ éléments tandis que la seconde est plus longue
- les $n$ premiers nombres sont égaux mais que le $n+1^{\text{ème}}$ de la première liste est inférieur au $n+1^{\text{ème}}$ de la seconde liste

Par conséquent, si $l$ est la longueur de la liste la plus courte, comparer ces deux listes d'entiers revient à parcourir tous les indices depuis 0 jusqu'à $l$ exclu et à s'arrêter sur la première différence qui détermine le résultat. S'il n'y pas de différence, alors la liste la plus courte est la première. Il faut écrire une fonction `compare_liste(p, q)` qui implémente cet algorithme.

**Correction**

Cet algorithme de comparaison est en fait celui utilisé pour comparer deux chaînes de caractères.

```
def compare_liste (p,q) :
    i = 0
    while i < len (p) and i < len (q) :
        if   p [i] < q [i] : return -1  # on peut décider
        elif p [i] > q [i] : return  1  # on peut décider
        i += 1                          # on ne peut pas décider
    # fin de la boucle, il faut décider à partir des longueurs des listes
    if   len (p) < len (q) : return -1
    elif len (p) > len (q) : return  1
    else : return 0
```

On pourrait également écrire cette fonction avec la fonction `cmp` qui permet de comparer deux éléments quels qu'ils soient.

```
def compare_liste (p,q) :
    i = 0
    while i < len (p) and i < len (q) :
        c = cmp (p [i], q [i])
        if c != 0 : return c            # on peut décider
        i += 1                          # on ne peut pas décider
    # fin de la boucle, il faut décider à partir des longueurs des listes
    return cmp (len (p), len (q))
```

### 12.4.5 Comprendre une erreur d'exécution *

Le programme suivant est erroné.

```
l = [0,1,2,3,4,5,6,7,8,9]
i = 1
while i < len (l) :
    print l [i], l [i+1]
    i += 2
```

Il affiche des résultats puis une erreur.

```
1 2
3 4
5 6
7 8
9
Traceback (most recent call last):
  File "examen2008_rattrapage.py", line 43, in <module>
    print l [i], l [i+1]
IndexError: list index out of range
```

Pourquoi ?

#### Correction

Le programme affiche les nombres par groupes de deux nombres consécutifs. Une itération de la boucle commence si la liste contient un élément suivant et non deux. Le programme est donc contraint à l'erreur car lors de la dernière itération, la liste contient un dixième nombre mais non un onzième. Le programme affiche le dixième élément (9) puis provoque une erreur `list index out of range`.

### 12.4.6 Précision des calculs **

On cherche à calculer la somme des termes d'une suite géométriques de raison $\frac{1}{2}$. On définit $r = \frac{1}{2}$, on cherche donc à calculer $\sum_{i=0}^{\infty} r^i$ qui une somme convergente mais infinie. Le programme suivant permet d'en calculer une valeur approchée. Il retourne, outre le résultat, le nombre d'itérations qui ont permis d'estimer le résultat.

```
def suite_geometrique_1 (r) :
    x = 1.0
    y = 0.0
    n = 0
    while x > 0 :
        y += x
        x *= r
        n += 1
    return y,n

print suite_geometrique_1 (0.5)  #affiche (2.0, 1075)
```

Un informaticien plus expérimenté a écrit le programme suivant qui retourne le même résultat mais avec un nombre d'itérations beaucoup plus petit.

```
def suite_geometrique_2 (r) :
    x = 1.0
    y = 0.0
    n = 0
    yold = y + 1
    while abs (yold - y) > 0 :
        yold = y
        y += x
        x *= r
        n += 1
    return y,n

print suite_geometrique_2 (0.5)  #affiche (2.0, 55)
```

Expliquez pourquoi le second programme est plus rapide tout en retournant le même résultat. Repère numérique : $2^{-55} \sim 2,8.10^{-17}$.

**Correction**

Tout d'abord le second programme est plus rapide car il effectue moins d'itérations, 55 au lieu de 1075. Maintenant, il s'agit de savoir pourquoi le second programme retourne le même résultat que le premier mais plus rapidement. L'ordinateur ne peut pas calculer numériquement une somme infinie, il s'agit toujours d'une valeur approchée. L'approximation dépend de la précision des calculs, environ 14 chiffres pour *Python*. Dans le premier programme, on s'arrête lorsque $r^n$ devient nul, autrement dit, on s'arrête lorsque $x$ est si petit que *Python* ne peut plus le représenter autrement que par 0, c'est-à-dire qu'il n'est pas possible de représenter un nombre dans l'intervalle $\left[0, 2^{-1055}\right]$.

Toutefois, il n'est pas indispensable d'aller aussi loin car l'ordinateur n'est de toute façon pas capable d'ajouter un nombre aussi petit à un nombre plus grand que 1. Par exemple, $1 + 10^{17} = 1,000\,000\,000\,000\,000\,01$. Comme la précision des calculs n'est que de 15 chiffres, pour *Python*, $1 + 10^{17} = 1$. Le second programme s'inspire de cette remarque : le calcul s'arrête lorsque le résultat de la somme n'évolue plus car il additionne des nombres trop petits à un nombre trop grand. L'idée est donc de comparer la somme d'une itération à l'autre et de s'arrêter lorsqu'elle n'évolue plus.

Ce raisonnement n'est pas toujours applicable. Il est valide dans ce cas car la série $s_n = \sum_{i=0}^{n} r^i$ est croissante et positive. Il est valide pour une série convergente de la forme $s_n = \sum_{i=0}^{n} u_i$ et une suite $u_n$ de module décroissant.

### 12.4.7 Analyser un programme **

Un chercheur cherche à vérifier qu'une suite de 0 et de 1 est aléatoire. Pour cela, il souhaite compter le nombre de séquences de $n$ nombres successifs. Par exemple, pour la suite 01100111 et $n = 3$, les triplets sont 011, 110, 100, 001, 011, 111. Le triplet 011 apparaît deux fois, les autres une fois. Si la suite est aléatoire, les occurrences de chaque triplet sont en nombres équivalents.

Le chercheur souhaite également faire varier $n$ et calculer les fréquences des triplets, quadruplets, quintuplets, ... Pour compter ses occurrences, il hésite entre deux structures, la première à base de listes (à déconseiller) :

```
def hyper_cube_liste (n, m = [0,0]) :
    if n > 1 :
        m [0] = [0,0]
        m [1] = [0,0]
        m [0] = hyper_cube_liste (n-1, m [0])
        m [1] = hyper_cube_liste (n-1, m [1])
    return m
h = hyper_cube_liste (3)
print h                     # affiche [[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
```

La seconde à base de dictionnaire (plus facile à manipuler) :

```
def hyper_cube_dico (n) :
    r = { }
    ind = [ 0 for i in range (0,n) ]
    while ind [0] <= 1 :
        cle                 = tuple (  ind ) # conversion d'une liste en tuple
        r [cle]             = 0
        ind [ len (ind)-1] += 1
        k                   = len (ind)-1
        while ind [ k ] == 2 and k > 0 :
            ind [k]    = 0
            ind [k-1] += 1
            k         -= 1
    return r
h = hyper_cube_dico (3)
print h        # affiche {(0, 1, 1): 0, (1, 1, 0): 0, (1, 0, 0): 0, (0, 0, 1): 0,
               #          (1, 0, 1): 0, (0, 0, 0): 0, (0, 1, 0): 0, (1, 1, 1): 0}
```

Le chercheur a commencé à écrire son programme :

```
def occurrence (l,n) :
    d = ....... # choix d'un hyper_cube (n)
    .....
    return d
suite = [ 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1 ]
h = occurrence (suite, 3)
print h
```

Sur quelle structure se porte votre choix (a priori celle avec dictionnaire), compléter la fonction `occurrence`.

### Correction

Tout d'abord, la structure matricielle est à déconseiller fortement même si un exemple d'utilisation en sera donné. La solution avec dictionnaire est assez simple :

```
def occurrence (l,n) :
    d = hyper_cube_dico (n)
    for i in range (0, len (l)-n) :
        cle = tuple (l [i:i+n])
        d [cle] += 1
    return d
```

Il est même possible de se passer de la fonction `hyper_cube_dico` :

```
def occurrence (l,n) :
    d = { }
    for i in range (0, len (l)-n) :
        cle = tuple (l [i:i+n])
        if cle not in d : d [cle] = 0
        d [cle] += 1
    return d
```

La seule différence apparaît lorsqu'un n-uplet n'apparaît pas dans la liste. Avec la fonction `hyper_cube_dico`, ce n-uplet recevra la fréquence 0, sans cette fonction, ce n-uplet ne sera pas présent dans le dictionnaire `d`. Le même programme avec la structure matricielle est plus une curiosité qu'un cas utile.

```
def occurrence (l,n) :
    d = hyper_cube_liste (n, [0,0])         # * remarque voir plus bas
    for i in range (0, len (l)-n) :
        cle = l [i:i+n]
        t   = d                         #
        for k in range (0,n-1) :        # point clé de la fonction :
            t = t [ cle [k] ]           #                 accès à un élément
        t [cle [ n-1] ] += 1
    return d
```

Si on remplace la ligne marquée d'une étoile par `d = hyper_cube_list(n)`, le programme retourne une erreur :

```
Traceback (most recent call last):
  File "examen2008_rattrapage.py", line 166, in <module>
    h = occurrence (suite, n)
  File "examen2008_rattrapage.py", line 160, in occurrence
    t [cle [ n-1] ] += 1
TypeError: 'int' object is not iterable
```

Cette erreur est assez incompréhensible puisque la modification a consisté à appeler une fonction avec un paramètre de moins qui était de toutes façons égal à la valeur par défaut associée au paramètre. Pour comprendre cette erreur, il faut exécuter le programme suivant :

```
def fonction (l = [0,0]) :
    l [0] += 1
    return l

print fonction ()         # affiche [1,0] : résultat attendu
print fonction ()         # affiche [2,0] : résultat surprenant
print fonction ( [0,0])   # affiche [1,0] : résultat attendu
```

L'explication provient du fait que la valeur par défaut est une liste qui n'est pas recréée à chaque appel mais c'est la même liste à chaque fois que la fonction est appelée sans paramètre. Pour remédier à cela, il faudrait écrire :

```
import copy
def fonction (l = [0,0]) :
```

```
    l = copy.copy (l)
    l [0] += 1
    return l
```

Dans le cas de l'hypercube, il faudrait écrire :

```
def hyper_cube_liste (n, m = [0,0]) :
    m = copy.copy (m)
    if n > 1 :
        m [0] = [0,0]
        m [1] = [0,0]
        m [0] = hyper_cube_liste (n-1, m [0])
        m [1] = hyper_cube_liste (n-1, m [1])
    return m
```

## 12.5 Exercices supplémentaires

### 12.5.1 Position initiale d'un élément après un tri **

On suppose qu'on peut trier la liste `l` = [...] avec la méthode `sort`. Une fois le tableau trié, comment obtenir la position du plus petit élément dans le tableau initial ?

#### Correction

L'idée est de ne pas trier la liste mais une liste de couples incluant chaque élément avec sa position. On suppose que la liste `l` existe. La méthode `sort` utilisera le premier élément de chaque couple pour tier la liste de couples.

```
l2 = [ ( l [i], i ) for i in range (0, len (l)) ]
l2.sort ()
print l2 [0][1]  # affiche la position du plus petit élément
                 # dans le tableau initial
```

### 12.5.2 Comprendre une erreur de logique *

**1)** Quel est le résultat affiché par le programme suivant :

```
def ensemble_lettre (s) :
          ens = []
          for i in range (0, len (s)) :
              c = s [i]
              if c in ens :
                  ens.append (c)
          return ens

print lettre ("baaa")
```

Est-ce que ce résultat change si on appelle la fonction `ensemble_lettre` avec un autre mot ?

**2)** Le programme précédent n'est vraisemblablement pas fidèle aux intentions de son auteur. Celui-ci avait pour objectif de déterminer l'ensemble des lettres différentes

de la chaîne de caractères passée en entrée. Que faut-il faire pour le corriger ? Que sera le résultat de l'instruction `ensemble_lettre("baaa")` en tenant compte de la modification suggérée ?

**Correction**

**1)** L'instruction `if c in ens :` signifie que le caractère `c` est ajouté seulement s'il est déjà présent dans la liste `s` qui est vide au début de la fonction. Elle sera donc toujours vide à la fin de l'exécution de la fonction et sera vide quelque soit le mot `s` fourni en entrée comme paramètre. Le résultat ne dépend donc pas du mot.

**2)** Il suffit de changer `if c in ens :` en `if c not in ens :` pour donner :

```
def ensemble_lettre (s) :
    ens = []
    for i in range (0, len (s)) :
        c = s [i]
        if c not in ens :
            ens.append (c)
    return ens
```

Le résultat pour le mot "baaa" est ["b","a"].

### 12.5.3 Comprendre une erreur d'exécution *

```
k = [10,14,15,-1,6]
l = []
for i in range (0,len (k)) :
    l.append ( k [ len (k) - i ] )
```

Le programme génère une exception de type `IndexError`, pourquoi ?

**Correction**

Lors du premier passage dans la boucle `for`, le premier ajouté dans la liste `l` est `k[len(k)]` qui n'existent pas puisque les indices vont de 0 à `len(k)`. Voici la correction :

```
k = [10,14,15,-1,6]
l = []
for i in range (0,len (k)) :
    l.append ( k [ len (k) - i-1 ] )  # -1 a été ajouté
```

### 12.5.4 Précision des calculs ***

On cherche à calculer $\ln 2$ grâce à la formule $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + ....$ Pour $x = 1$, cette suite est convergente et permet de calculer $\ln 2$.

**1)** On propose deux programmes. Le premier utilise une fonction `puiss`. Le second ne l'utilise pas :

```
def puiss (x, n) :
    s = 1.0
    for i in range (0,n) :
        s *= x
    return s

def log_suite (x) :
    x    = float (x-1)
    # ......
    s    = 0
    old  = -1
    n    = 1
    while abs (old - s) > 1e-10 :
        old = s
        po  = puiss (x,n) / n
        if n % 2 == 0 : po = -po
        s  += po
        n  += 1
        # ......
    return s

print log_suite (2)
```

```
# ......
# ......
# ......
# ......
# ......
# ......
def log_suite (x) :
    x    = float (x-1)
    x0   = x
    s    = 0
    old  = -1
    n    = 1
    while abs (old - s) > 1e-10 :
        old = s
        po  = x / n
        if n % 2 == 0 : po = -po
        s  += po
        n  += 1
        x  *= x0
    return s

print log_suite (2)
```

Quel est le programme le plus rapide et pourquoi ? Quels sont les coûts des deux algorithmes ?

**2)** Combien faut-il d'itérations pour que la fonction retourne un résultat ?

**3)** On introduit la fonction `racine_carree` qui calcule $\sqrt{k}$. Celle-ci est issue de la résolution de l'équation $f(x) = 0$ avec $f(x) = x^2 - k$ à l'aide de la méthode de *Newton*[3].

```
def racine_carree (k) :
    x0 = float (k)+1
    x  = float (k)
    while abs (x-x0) > 1e-10 :
        x0 = x
        x  = (k-x*x) / (2 * x) + x
    return x
```

Cette fonction retourne un résultat pour $\sqrt{2}$ en 6 itérations. On décompose ensuite $\ln 2 = \ln\left((\sqrt{2})^2\right) = 2\ln\sqrt{2} = 2\ln\left(1+\left(\sqrt{2}-1\right)\right)$. En utilisant cette astuce et la fonction `racine2`, à combien estimez-vous grossièrement le nombre d'itérations nécessaires pour calculer $\ln 2$ : 10, 40 ou 100 ? Justifiez. On rappelle que $2^{10n} \sim 10^{3n}$.

**4)** Que proposez-vous pour calculer $\ln 100$ ?

### Correction

**1)** Le second programme est le plus rapide. A chaque boucle du premier programme, la fonction `puiss` calcule $x^n$ avec $n$ multiplications, ce calcul est remplacé par une seule multiplication dans le second programme.

3. Cette méthode est utilisée pour résoudre l'équation $f(x) = 0$. On construit une suite convergeant vers la solution définie par $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Sous certaines conditions, la suite $(x_n)$ converge vers la solution. Il suffit d'appliquer cela à la fonction $f(x) = x^2 - k$ où $k$ est le nombre dont on veut trouver la racine.

Les deux programmes exécuteront exactement le même nombre de fois la boucle `while`. Soit $n$ ce nombre d'itérations, le coût du premier programme est $1+2+3+4+...+n \sim O(n^2)$ car la fonction `puiss` fait $i$ passages dans sa boucle `for` pour l'itération $i$. Le coût du second programme est en $O(n)$.

**2)** Si on pose $s_n = \sum_{k=1}^{n} (-1)^{k+1} \frac{x^k}{k}$, la condition d'arrêt de la fonction `log_suite` correspond à :

$$|s_n - s_{n-1}| > 10^{-10} \iff \left|\frac{x^k}{k}\right| > 10^{-10}$$

Comme on calcule $\ln(1+1)$, $x=1$, la condition est donc équivalente à :

$$\frac{1}{k} > 10^{-10} \iff k < 10^{10}$$

La fonction `log_suite` fait dix milliards de passages dans la boucle `while`, dix milliards d'itérations.

**3)** Cette fois-ci, on cherche à calculer $\ln\sqrt{2} = \ln\left(1+\sqrt{2}-1\right)$, d'où $x = \sqrt{2}-1$. La condition d'arrêt est toujours $\left|\frac{x^k}{k}\right| > 10^{-10}$ et on cherche à majorer le nombre d'itérations nécessaires pour que la fonction converge vers un résultat. $\sqrt{2} \sim 1.414 < \frac{1}{2}$. On majore donc $x$ par $\frac{1}{2}$ :

$$\begin{aligned}
&\implies \left|\frac{\left(\sqrt{2}-1\right)^k}{k}\right| > 10^{-10} \\
&\implies \left|\frac{1}{2^k k}\right| > 10^{-10} \\
&\implies \frac{1}{2^k} > 10^{-10} \implies 2^k < 10^{10}
\end{aligned}$$

On utilise l'approximation donnée par l'énoncé à savoir $2^{10n} \sim 10^{3n}$.

$$2^k < 10^{10} \iff 2^{10k} < 10^{100} \sim 10^{3k} < 10^{100} \iff 3k < 100 \iff k < \frac{100}{3} \iff k < 34$$

Si on ajoute 34 aux 6 itérations nécessaires pour calculer $\sqrt{2}$, on trouve 40 itérations.

**4)** $\ln(1+x)$ n'est calculable que pour $|x| < 1$. Il faut fonc transformer $\ln 100$ pour le calculer à partir d'une racine carrée et de $\ln(1+x)$ pour un $x$ tel que $x < 1$ (dans le cas contraire, la suite ne converge pas) :

$$\begin{aligned}
\ln 100 &= 2\ln 10 = 2\ln 2 + 2\ln 5 = 2\ln 2 + 2\ln\left(4\frac{5}{4}\right) \\
&= 2\ln 2 + 2\ln 4 + 2\ln\frac{5}{4} = 2\ln 2 + 4\ln 2 + 2\ln\left(1+\frac{5}{4}-1\right) \\
&= 6\ln 2 + 2\ln\left(1+\frac{1}{4}\right)
\end{aligned}$$

C'est une décomposition, ce n'est pas la seule possible.

### 12.5.5 Permutation aléatoire **

Une liste `l` contient des nombres et on souhaite les permuter dans un ordre aléatoire. On dispose pour cela de deux fonctions :

```
i = random.randint (0,n) # tire aléatoirement un nombre entier entre 0 et n inclus

l.sort ()                # trie la liste l quel que soit son contenu
```

Dans le cas d'une liste de couples (ou 2-uple), chaque élément de la liste est trié d'abord selon la première coordonnée puis, en cas d'égalité, selon la seconde coordonnée. Les quelques lignes qui suivent illustrent le résultat souhaité. Il reste à imaginer la fonction qui permute de façon aléatoire la liste `l`. (3 points)

```
def permutation_aleatoire (l) :
    ....
    return ...

print permutation_aleatoire ([1,2,3,4])   # affiche [3,1,4,2]
```

#### Correction

Pour répondre au problème, on s'inspire d'une méthode qui trie une liste tout en récupérant la position de chaque élément [4]. Pour ce faire, on construit une liste de couples (élément, position).

```
tab = ["zéro", "un", "deux"] # tableau à trier
pos = [ (tab [i],i) for i in range (0, len (tab)) ] # tableau de couples
pos.sort () # tri
print pos # affiche [('deux', 2), ('un', 1), ('zéro', 0)]
```

Pour effectuer une permutation, on construit une liste de couples (position aléatoire, élement). En triant sur une position aléatoire, on dérange la liste des éléments de façon aléatoire. Il suffit alors de récupérer la seconde colonne de cette liste de couples pour obtenir une permutation aléatoire.

```
from random import randint

def permutation (liste) :
    alea        = [ randint (0,len (liste)) for i in liste ]
    couple      = [ (r,l) for r,l in zip (alea,liste) ]
    couple.sort ()
    permutation = [ l[1] for l in couple ]
    return permutation

liste = [ i*2 for i in range (0,10) ]
permu = permutation (liste)
print liste  # affiche [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
print permu  # affiche [2, 6, 16, 10, 8, 14, 0, 12, 18, 4]
```

4. paragraphe 3.7.7, page 91

# Index

## P

**U**



**V**



**W**