Langage Python

http ://www.xavierdupre.fr/

Xavier Dupré

Repères

La table des matières est détaillée à la fin du document.

Chapitre 1

Introduction

1.1 Ordinateur et langages

L'informatique présentée dans ce cours est abordée d'un point de vue utilitaire. C'et outil permet d'automatiser des tâches répétitives, d'accélérer des calculs complexes comme l'approximation d'un calcul intégral (voir figure 1.1). Le langage *Python*, sans être le plus rapide des langages de programmation, permet dans la plupart des cas de concevoir plus rapidement un programme informatique répondant à un objectif donné. Mais avant de rentrer dans le vif du sujet, voici quelques détails techniques et quelques termes fréquemment employés en informatique.

1.1.1 L'ordinateur

On peut considérer simplement qu'un ordinateur est composé de trois ensembles :

- le microprocesseur
- la mémoire
- les périphériques (écran, imprimantes, disque dur, ...)

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.

C'est pourquoi 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). Ces échanges

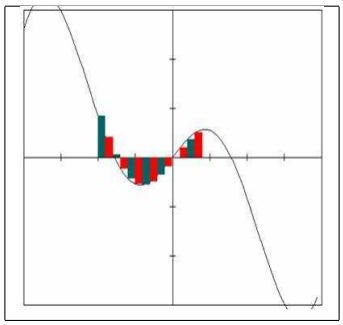

Fig. 1.1: Illustration du calcul approché d'une intégrale à l'aide de la méthode des rectangles.

entre processeur et mémoire sont rapides.

Les périphériques regroupent tout le reste (écran, clavier, souris, disque dur, imprimante...), tout ce qui nous permet de dialoguer avec l'ordinateur et tout ce qui nous permet de conserver des informations une fois que celui-ci est éteint.

1.1.2 Termes informatiques

Définition 1.1.1 : 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 *Python*, en français, un algorithme reste le même, ce qui favorise l'apprentissage d'un nouveau langage informatique lorsqu'on en connaît déjà un.

Définition 1.1.2 : programme
Un programme informatique est une suite d'instructions ou séquence d'instructions. C'est la réalisation informatique d'un algorithme. Il dépend du langage.

Définition 1.1.3 : compilateur et compilation
Le compilateur est un programme qui traduit un code écrit dans un langage de programmation (ici le C) en langage dit "machine", compréhensible par l'ordinateur. La compilation est le fait de traduire un programme afin que l'ordinateur le comprenne.

Par la suite, certaines erreurs de compilations courantes seront citées, celles-ci dépendant du compilateur choisi qui est pour ce cours celui du langage *Python*.

Définition 1.1.4 : mot clé
Un mot clé est une composante du langage et fait partie de sa grammaire.

La table 3.1 (page 41) regroupe les mots-clé du langage *Python*.

1.2 Présentation du langage *Python*

1.2.1 Histoire

Python est un langage objet interprété de haut niveau, créé au début des années quatre-vingt-dix par Guido Van Rossum au Centrum voor Wiskunde à *Informatica*, Amsterdam. . En 1995, Rossum poursuivit le développement de *Python* à la *Corporation for National Research Initiatives* de Reston (Virginie). Et en 2000, Rossum créa l'équipe BeOpen PythonLabs qui, en octobre de la même année est incorporée à Zope Corporation puis à la société Digital Creations. 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. Elle est sponsorisée en particulier par Zope Corporation. *Python* est distribué sous forme de logiciel libre. *Python* est couvert par sa propre licence (voir le site [www-PyLicence] ou le chapitre 9.2).Toutes les versions

depuis la 2.0.1 sont compatibles avec la licence GPL (GNU Public Licence[1])..

1.2.2 Description sommaire

On distingue plusieurs classes parmi les langages informatiques selon la syntaxe qu'ils proposent ou les possibilités qu'ils offrent. *Python* est un langage :

1. interprété
2. orienté objet
3. de haut niveau
4. modulaire
5. à syntaxe positionnelle

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 néanmoins possible de traduire un programme dans un langage (bytecode) qui est ensuite interprété par une machine virtuelle *Python*. Ce mécanisme est semblable à celui propre au langage Java.

On considère que le langage *Python* est de haut niveau car il propose des fonctionnalités avancées et automatiques telle le *garbage collecting*. Cette tâche correspond à la destruction automatique des objets créés lorsqu'ils ne sont plus utilisés. Il propose également des structures de données complexes telles que des dictionnaires, éloignées des types numériques standards.

Le langage *Python* est modulaire. La définition du langage est très succinte et autour de ce noyau concis, de nombreuses librairies ou modules ont été développées. *Python* est assez intuitif, être à l'aise avec ce langage revient à connaître tout autant sa syntaxe que les nombreux modules disponibles, eux-mêmes écrits en *Python*.

Le langage *Python* est à syntaxe positionnelle en ce sens que l'indentation fait partie du langage. Le point virgule permet de séparer les instruction en langage C, l'accolade permet de commencer un bloc d'instruction. 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 montrer qu'elles appartiennent au même bloc d'instructions.

1.2.3 Avantages et inconvénients du langage *Python*

Alors qu'il y a quelques années, le langage C puis le langage C++ s'imposaient souvent comme langage de programmation, il existe dorénavant une profusion de langages (Java, PHP, Visual Basic, Perl, ...). Il est souvent possible de transposer les mêmes algorithmes d'un langage à un autre. Le choix approprié est alors celui qui offre la plus grande simplicité lors de la mise en œuvre d'un programme et aussi lors de son utilisation (vitesse d'exécution notamment).

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, Windows, Mac Os... *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. Les paragraphes qui suivent présentent les avantages et les inconvénients de *Python* face à d'autres langages.

[1] voir le site [www-GPL].

1.2.3.1 *Python* et Java

La syntaxe de *Python* est beaucoup plus simple que celle de Java (proche du C), ce qui améliore de façon très significative les temps de développement. Le programmeur ne perd pas de temps en déclaration de types, de variables, ... *Python* intègre des types de données très puissants, comme les listes et dictionnaires polymorphiques qui simplifient considérablement le travail de programmation. Enfin, *Python* est un langage totalement ouvert et libre, qui ne dépend d'aucune entreprise particulière.

1.2.3.2 *Python* et Perl

Python et Perl partagent un certain nombre de concepts mais leurs philosophies sont totalement différentes. Perl est plutôt destiné à programmer des tâches de bas niveau, avec son système d'expressions régulières, d'analyse de fichier et de génération de rapport. *Python* est plus orienté vers le développement d'applications, nécessitant des structures de données plus complexes et encourage le programmeur à produire du code facile à maintenir.

1.2.3.3 *Python* et C++

La plupart des remarques concernant Java s'appliquent à C++. Ajoutons encore que si le code *Python* est typiquement trois à cinq fois plus court que le code Java équivalent, il est de cinq à dix fois plus court que le code C++ correspondant. C'est un gain de temps notable lors des phases de développement et de maintenance des programmes. Un programme C++ nécessite une recompilation chaque fois que l'on change d'environnement, un programme compilé sur une plate-forme ne pouvant en aucun cas être exécuté sur une autre. A l'inverse, un programme *Python* s'exécutera sur toute plate-forme disposant de la machine virtuelle Python. Son principal inconvénient face au langage C++ est sa vitesse d'exécution, plus lente.

1.2.3.4 Conclusion

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 ou Java, et ce, d'une manière assez simple.

1.3 Installation sur Windows

1.3.1 Installation du langage *Python*

Python a l'avantage d'être disponible sur de nombreuses plate-formes comme *Windows*, *Linux* ou encore *Macintosh*. L'installation présentée ici concerne le système d'exploitation Windows uniquement. Toutefois, excepté ce paragraphe, les exemples décrits par la suite ne dépendront pas d'un quelconque système.

Sous *Windows*, il suffit de décompresser le fichier *Python-2.3.4.exe* disponible à l'adresse [www-PyDownload] et dont la documentation associée est fournie à l'adresse [www-PyDoc]. Il est conseillé de télécharger la dernière version stable du langage. Les options d'installation choisies sont celles par défaut, le répertoire d'installation est *C :/Python23*. A la fin de cette installation apparaît un menu supplémentaire dans le menu *Démarrer* (ou *Start*) de Windows comme le montre la figure 1.2.

Ce menu contient les intitulés suivant :

Fig. 1.2: Menu ajouté à Windows

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 décrit en détail le langage *Python*, elle inclut également un tutoriel qui permet de le découvrir. La ligne de commande (voir figure 1.4) permet d'exécuter des instructions en langage *Python*. Elle est pratique pour effectuer des calculs mais reste contre-indiqué lorsqu'il s'agit d'écrire un programme. C'est pourquoi il est nécessaire d'utiliser un éditeur de texte qui permet d'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.

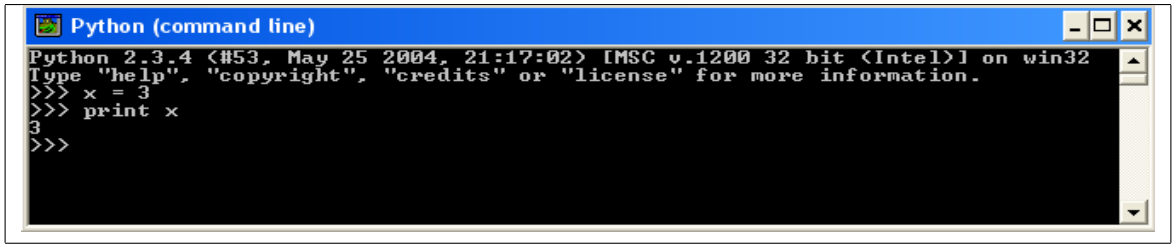

Fig. 1.3: Ligne de commande, la première ligne affecte la valeur 3 à la variable x, la seconde ligne l'affiche.

1.3.2 Utilisation de l'éditeur de texte

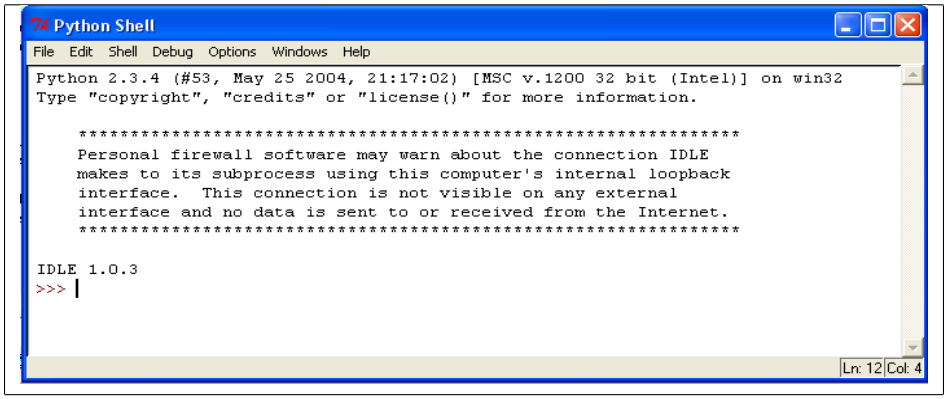

Fig. 1.4: Fenêtre de commande fournie avec le langage *Python*.

La figure 1.2 montre le menu installé par *Python*dans le menu "Démarrer" de Windows. En choisissant

l'intitulé "IDLE (Python GUI)", on active la fenêtre de commande de *Python*(voir figure 1.4).Les instructions sont interprétées 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 `x`, `y`, `z` existent toujours. Pour effacer toutes les variables créées, il suffit de redémarrer l'interpréteur par l'intermédiaire du menu `Shell` $-->$ `RestartShell`. Les trois variables précédentes auront disparu.

Il n'est pas possible de conserver le texte qui a été saisi au clavier, ce programme conserve seulement les résultats de son interprétation. Il est possible toutefois 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 suffit d'actionner le menu `File` $-->$ `NewWindow` qui ouvre une seconde fenêtre qui fonctionne comme un éditeur de texte (voir figure 1.5).

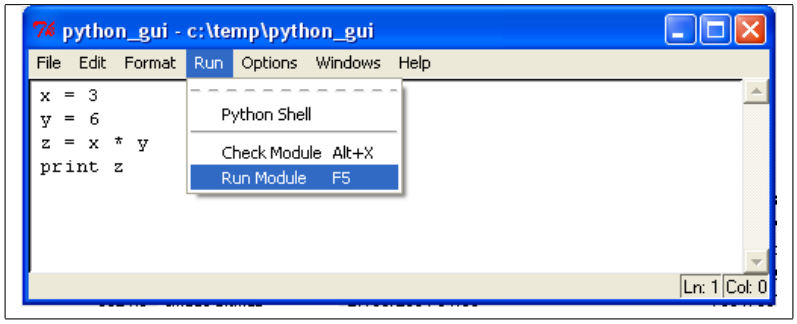

Fig. 1.5: Fenêtre de programme, le menu `Run` $-->$ `RunModule` permet de lancer l'exécution du programme. L'interpréteur *Python* est réinitialisé au début de l'exécution et ne conserve aucune trace des travaux précédents.

Après que le programme a été entré, le menu `Run` $-->$ `RunModule` exécute le programme. Il demande au préalable s'il faut enregistrer puis réinitialise l'interpréteur *Python* pour effacer les traces des exécutions précédentes. Le résultat apparaît dans la première fenêtre (celle de la figure 1.4). La pression des touches "Ctrl + C" permet d'arrêter le programme avant qu'il n'arrive à sa fin.

Remarque 1.3.1: fenêtre intempestive
Si on veut de débarasser de cette fenêtre intempestive qui demande confirmation pour enregistrer les dernières modifications, il suffit d'aller dans le menu `Options` $-->$ `ConfigureIDLE` et de choisir `No prompt` sur la quatrième ligne dans la rubrique `General` (voir figure 1.6).

Cette description succinte permet néanmoins de réaliser puis d'exécuter des programmes simples. 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 existe d'autres éditeurs plus riches comme celui proposé au paragraphe 1.3.3.1. Ils sont souvent conçus de manière à s'adapter à plusieurs autres langages tels que C, Perl, PHP, ou HTML. Il suffit pour cela de leur indiquer l'emplacement du compilateur associé au langage choisi. Cette configuration est souvent manuelle mais semblable d'un éditeur à l'autre.

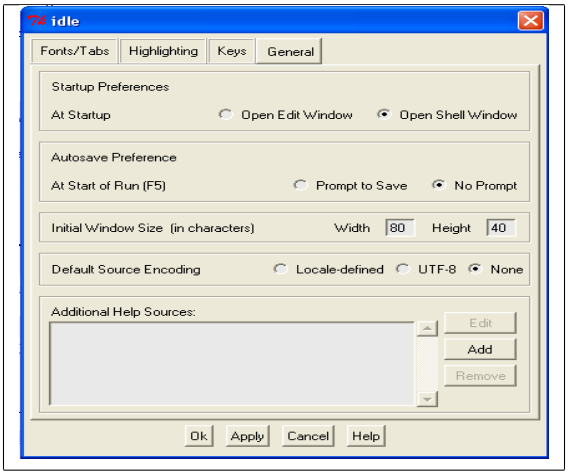

Fig. 1.6: Cliquer sur `General` puis sur `Noprompt` pour se débarrasser de la fenêtre intempestive qui demande à l'utilisateur de confirmer la sauvegarde des dernières modifications.

1.3.3 Installation d'un autre éditeur de texte

1.3.3.1 Editeur Syn Editor

Il existe de nombreux éditeurs de texte, payant ou gratuit. Le premier proposé est *Syn Text Editor*, il est gratuit et fonctionne avec de nombreux compilateurs de langages différents (voir figure 1.8). Il existe d'autres éditeurs, dont la plupart sont référencés par le site officiel de *Python*à l'adresse [www-PyEdit]. Pour installer cet éditeur, il suffit d'exécuter le fichier *synsetup-2.1.0.43.exe* téléchargeable depuis l'adresse [www-Syn] dans la rubrique *Download*.

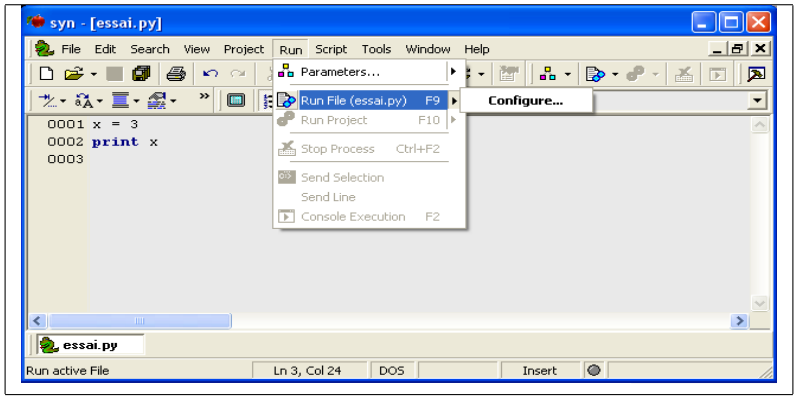

Fig. 1.7: Même programme que celui de la figure 1.4, ce programme est enregistré sous le nom "essai.py", et on tente de l'exécuter par l'intermédiaire du menu "Run -> Run File".

La fenêtre de la figure 1.9a apparaît où aucun champ n'est renseigné. Il faut alors ajouter un profile qu'on appellera *python*. La fenêtre 1.9b apparaît. Une fois cette étape terminée, il faut cliquer sur l'icône cerclé de rouge de l'image 1.9a pour insérer un nom de programme, ici "essai". Une fois ce nom ajouté, on clique dessus pour enfin renseigner les derniers champs comme indiqué dans la figure 1.9a afin de préciser l'emplacement du compilateur *Python*.

Après cette étape, il ne reste plus qu'à exécuter le programme en appuyant sur la touche F9. Le résultat s'affiche dans la fenêtre "Output" en bas de l'écran comme le montre la figure 1.10. Une erreur de syntaxe comme dans le programme suivant :

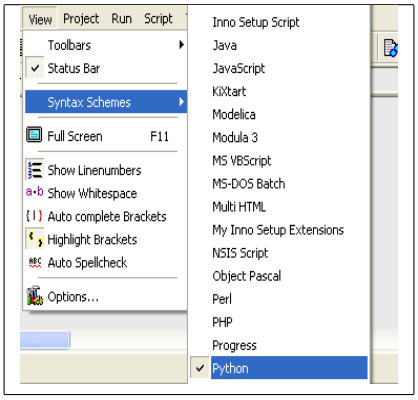

Fig. 1.8: Configuration de l'éditeur, il est possible de configurer l'éditeur pour de nombreux langages.

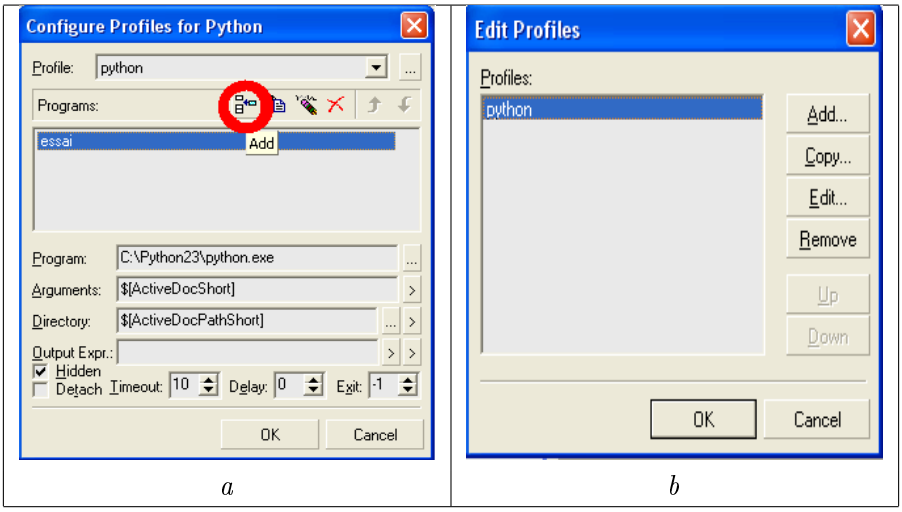

Fig. 1.9: Configuration de l'éditeur, ajout d'un profil qui permet d'exécuter le programme avec le compilateur *Python*.

```
x = 3
print xy
```

aboutira à une erreur de compilation à la seconde ligne indiquant que la variable `xy` n'existe pas :

```
=============== Running Profile: python ===============

------------------------------------------------------------
Commandline: C:/Python23/python.exe c:/temp/essai.py
Workingdirectory: c:/temp
Timeout: 10 sec, Delay: 0 sec, Started: 08/14/04 13:45:13
------------------------------------------------------------

Traceback (most recent call last):
    File "c:/temp/essai.py", line 2, in ?
    print xy
```

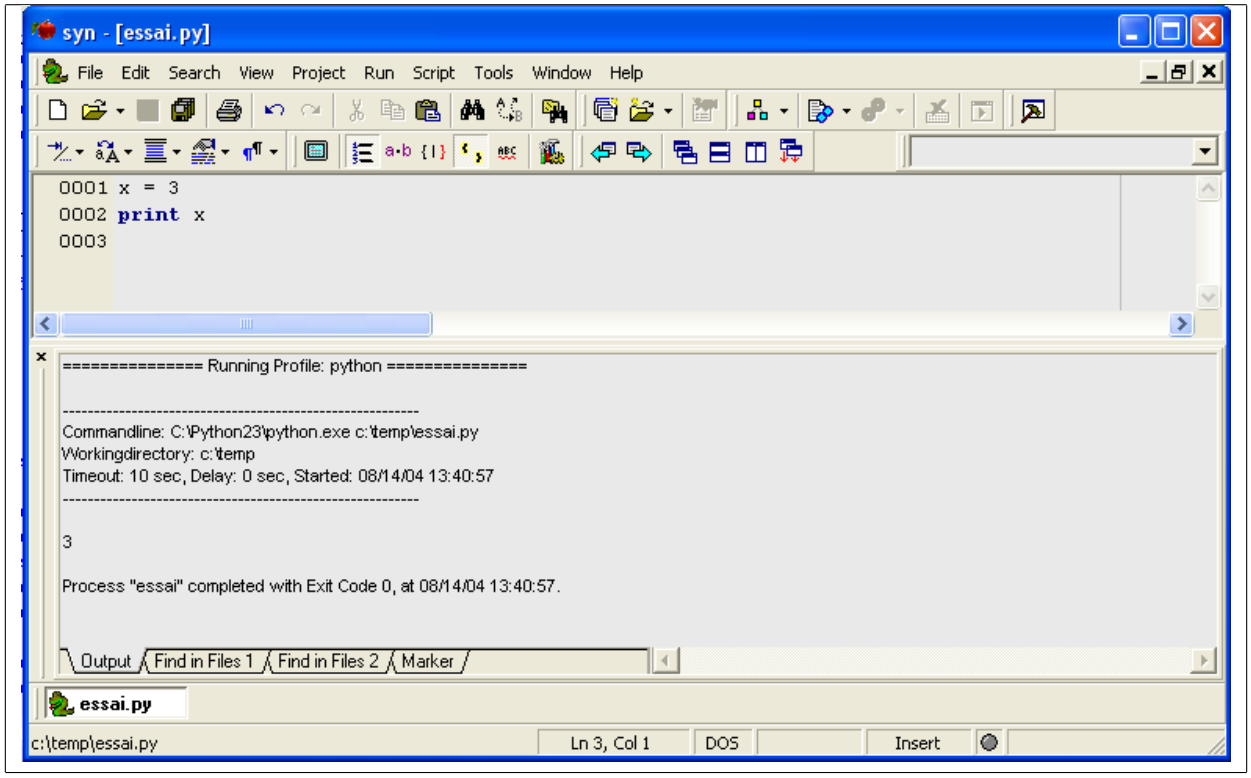

Fig. 1.10: Exécution du programme "essai.py" dans la fenêtre "Output".

```
NameError: name 'xy' is not defined

Process "essai" completed with Exit Code 1, at 08/14/04 13:45:13.
```

Cet éditeur possède néanmoins un inconvénient qui apparaît lors de l'utilisation de la fonction `raw_input` qui permet de saisir une réponse au clavier. Cette fonction bloque le programme. Dans ce cas, il est préférable d'utiliser l'éditeur de texte fourni avec le langage *Python*.

1.3.3.2 Editeur SciTe

L'éditeur SciTe est une autre possibilité. Contrairement au précédent, la fonction `raw_input` fonctionne parfaitement. La gestion des erreurs est également plus pratique puisqu'il est possible de cliquer sur une erreur pour que l'éditeur positionne automatique le curseur sur la ligne ayant généré cette erreur. Cet éditeur est accessible depuis l'adresse [www-Scite] et téléchargeable depuis l'adresse [www-SciteDownload].

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 1.11).

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)"
```

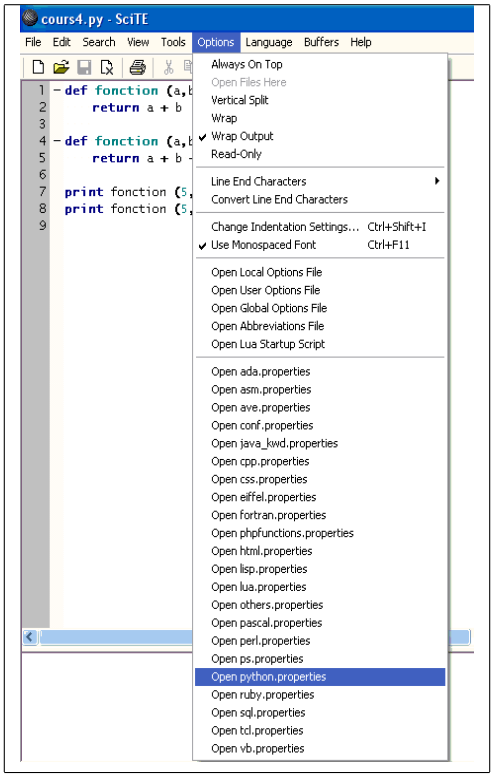

Fig. 1.11: Configuration de l'éditeur de texte Scite. Il faut cliquer sur le menu `Option`, puis sur `Open python.properties` pour ouvrir les paramètres de configuration associées au langage *Python*.

```
    command.go.subsystem.*.pyw=1
```

Il suffit de préciser le répertoire où se trouve le compilateur, par exemple `c : /python23`.

```
if PLAT_WIN
    command.go.*.py=c:\python23\python -u "$(FileNameExt)"
    command.go.subsystem.*.py=0
    command.go.*.pyw=c:\python23\pythonw -u "$(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 1.12 montre un cas où l'exécution a été interrompue par une erreur.

1.4 Installation sur Mac OSX

Le langage *Python* est déjà présent dans les ordinateurs d'Apple si ceux-ci sont équipés du système d'exploitation Mac OSX. Il reste néanmoins à installer les packages ou modules et cela peut parfois poser de nombreux problèmes. Tout d'abord, les dernières versions des packages sont plus rapidement disponibles sous Windows et sont souvent munis d'un programme d'installation automatique. En revanche, le noyau de Mac OSX est un noyau Linux. Certains modules disposent d'un installateur (fichier dont l'extension est *pkg* ou *mpkg*), d'autres doivent être installés en ligne de commande à partir d'une fenêtre Terminal car ils ne sont pas disponibles dans une version compilée (prête à l'emploi). De plus, certains packages

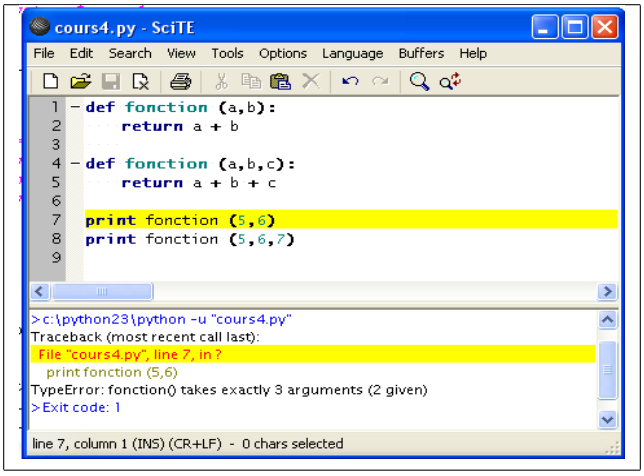

Fig. 1.12: Exécution d'un programme interrompue par une erreur. Pour cet éditeur, il suffit de cliquer sur le message d'erreur pour placer le curseur sur la ligne erronnée.

posent des problèmes comme le package SciPy. Toutefois, une fois que ces étapes parfois fastidieuses sont achevées, l'utilisation de *Python* est aussi facile sous Mac OSX que sous Windows.

A noter que toutes les manipulations qui suivent ont été faite avec une version Mac OSX 10.4 (ou Tiger). Le langage Python fait partie du système d'exploitation Mac OS X Tiger dans sa version 2.3. Cela signifie que tous les packages qui doivent être installés doivent l'être pour la version 2.3. Il est néanmoins possible de mettre à jour cette version (voir la page http ://www.python.org/download/download_mac.html).

1.4.1 Editeur

L'éditeur proposé est *TextWrangler*. Il est gratuit et accessible depuis l'adresse http ://www.barebones.com/products/textwrangler/index.shtml. Il permet d'exécuter un programme directement ou dans une fenêtre Terminal (utile si les programmes sont longs et ont beaucoup d'affichages). Il propose aussi d'associer à certains intitulés du menu un raccourci clavier, qui n'existent pas pour l'exécution d'un programme *Python*.

Il possède quelques défauts comme celui de ne pas remplacer automatiquement les tabulations par des espaces, il faut le spécifier pour chaque fichier. Lorsqu'un programme a été écrit sous un ordinateur équipé de Windows, celui-ci contient des codes de fins de lignes différents de ceux-ci utilisés sous Linux 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. Les accents sont aussi mal gérés car ils n'existent pas dans la langue anglaise.

1.4.2 Installation de packages

1.4.2.1 Installation de packages automatiquement

Un site recense de nombreux packages pour Mac, il s'agit de l'adresse http ://pythonmac.org/packages/. Il faut néanmoins prendre soin de choisir des packages correspondant à votre version de *Python* et à votre version de Mac OSX. Après avoir téléchargé, le fichier pkg ou mpkg, il suffit de l'exécuter.

En général, cette étape suffit pour que le module puisse être utilisé par la suite dans un programme *Python*. Toutefois, il arrive que cela ne fonctionne pas dans certains cas. Il est alors conseillé d'aller sur le site du

package concerné ou alors de reprendre l'intégralité ou une partie du message d'erreur et de regarder ce qu'un moteur de recherche comme Google trouve à son propos. Il est fort probable qu'une personne ait déjà été confronté à ce genre de problème et la recherche est rarement vaine. Elle aboutit souvent à plusieurs pages qu'il faut parcourir pour trouver une réponse (en anglais très souvent) qui soit compréhensible. Une erreur fréquente provient des dépendances entre packages comme pour le package PyGame.

1.4.2.2 module PyGame

Le package *PyGame* ne fonctionne que si le package *PyObjC* est aussi installé. Seul, l'utilisation de PyGame aboutit à une erreur difficilement compréhensible excepté par Google qui retourne quelques messages échangés par des programmateurs ayant déjà rencontré le problème :

```
2004-08-07 21:08:15.153 Python[695] *** _NSAutoreleaseNoPool(): Object
0x30ead0 of class NSCFArray autoreleased with no pool in place - just
leaking
```

Ces deux packages sont référencés à l'adresse http ://pythonmac.org/packages/ et s'installent automatiquement.

1.4.2.3 Installation de modules manuellement

L'installation automatique n'est pas toujours disponible, soit parce que le package est court et ne contient que des fichiers écrits en *Python* (package py2html qui transforme des programmes en pages HTML), soit ils incluent des fichiers écrits en langage C qui doivent être compilés avec le compilateur *gcc* installé sur votre ordinateur. Pour ces packages, il convient de :

1. télécharger le package et le décompresser
2. d'ouvrir une fenêtre Terminal
3. de se placer dans le répertoire où ont été décompressés les fichiers (utliser les commandes *ls* et *cd*)
4. de taper la commande *sudo su* qui donne tous les droits d'installation (cette commande demande le mot de passe de l'ordinateur et vous donne le droit de modifier le système)
5. de taper la commande *python setup.py build* s'il y a des fichiers C à compiler (module SciPy par exemple)
6. puis de taper la commande *python setup.py install*

Il est conseillé au préalable de modifier la version de votre compilateur. Sur la dernière version de Mac OSX, c'est le compilateur gcc 4.0 qui est utilisé or tous les packages que vous téléchargez sont bien souvent compilés avec la version 3.3. Il est préférable de la changer :

1. ouvrir une fenêtre Terminal
2. de taper la commande *sudo su* qui donne tous les droits d'installation
3. de taper *gcc_ select 3.3*

1.4.2.4 Numeric ou Numarray

De nombreux modules ont besoin de l'un ou de ses deux modules. Ils implémentent tous deux des vecteurs plus appropriés pour faire du calcul numérique. Il est conseillé d'installer les deux modules même si un autre module ne prétend n'avoir besoin que de l'un des deux (comme le module SciPy).

1.4.2.5 MatPlotLib

L'utilisation de MatPlotLib diffère de celle sous Windows. Alors qu'il est simplement nécessaire d'écrire la ligne :

```
import matplotlib.pylab as pylab
```

Avec la version Mac, il faut installer le package wxWindows et spécifier que MatPlotLib doit utiliser cette interface au début du programme :

```
#!/usr/bin/env pythonw2.3
import matplotlib
matplotlib.use("WXAgg")
import matplotlib.pylab as pylab
```

1.4.2.6 SciPy

Il est possible que les instructions d'installation du paragraphe 1.4.2.3 échouent avec le module SciPy. Voici quelques pistes pour terminer l'installation. A chaque modification, il faut relancer la compilation (*python setup.py build*) du module puis, si elle a marché, son installation (*python setup.py install*) jusqu'à ce qu'un message indiquant la réussite de l'opération apparaisse. Il faut néanmoins vérifier que tous les packages dont SciPy a besoin ont été préalablement installés (spécifiés à l'adresse http ://www.scipy.org/documentation/Members/fonnesbeck/osx_build.txt). Il est aussi conseillé d'installer les packages Numeric et Numarray (voir paragraphe 1.4.2.4). Il faut aussi appliquer la dernière remarque du paragraphe 1.4.2.3 concernant *gcc_select*.

1. installer le package g77 (chercher avec Google)
2. décompresser le fichier g77....
3. lancer un shell terminal
4. taper sudo su
5. déplacer chaque dans chaque répertoire vers le même répertoire dans /usr/local/... ex : mv * /usr/local/bin
6. recopier le fichier g77 dans /usr/bin
7. recopier le fichier gnufcompiler.py dans SciPy... complete/scipy_core/scipy/dist_utils
8. installer numarray
9. déplacer Numeric de son répertoire vers site-package s'il a été installé avec un .pkg, rien s'il a été compilé (avec 3.3)
10. taper export FC_VENDOR=Gnu (dans le fichier 3.3 py de ce répertoire, remplacer l'option lcc_dynamic par dynamic_lib (à essayer)
11. installer fftw avec cette configuration
12. taper python setup.py build
13. taper sudo python setup.py install

1.5 Notes à propos de ce document

Ce document décrit les points essentiels du langage *Python*. Une description plus complète est disponible dans le livre [Martelli2004] qui s'intéresse sommairement aux nombreuses extensions disponibles recouvrant des thèmes comme les calculs numériques (matriciels, ...), le traitement des images, la communication via un réseau ou encore l'utilisation du langage C via *Python* et l'utilisation réciproque, la lecture et l'écriture de données au format XML. Le site officiel du langage *Python* ([www-Python]) abrite la version la plus récente ainsi que de nombreux modules à l'adresse ([www-PyModule]). Les fonctionnalités offertes par ces modules sont très variées. Le module *pyXLWriter* permet par exemple d'écrire des données au format Excel. Le module FSA (Finite State Automate) implémente quelques algorithmes concernant les automates à états finis. Le module *MySQL-python* offre quant à lui un accès à *MySQL* via *Python*. Il serait inutile de dresser la liste des nombreux modules disponibles dans ce langage. Ils recouvrent de nombreux domaines. La simplicité du langage *Python* favorise son extension.

Etant donné que *Python* est récent, il existe encore peu de livres consacrés à ce langage. La plupart des documents sont disponibles via internet en anglais principalement comme sur le site officiel voire en français (voir le site [www-DocF]). Pour ceux qui débutent la programmation avec *Python*, le livre [Swinnen2003] propose une approche plus pédagogique que celle de [Martelli2004] qui s'adresse à des personnes déjà accoutumés à d'autres langages.

Vieux d'une dizaine d'années, *Python* est un langage en expansion dont la richesse réside surtout dans la liste des modules disponibles. Cette liste couvre un large panel de besoins susceptible de s'accroître avec le temps. Il peut être parfois utile de jeter un coup d'œil aux modules recensés sur internet ([www-Python]) avant de se lancer dans la programmation pour découvrir peut-être qu'un module recouvre une partie du programme sur le point d'être développé.

L'intérêt d'un tel langage est de parvenir à construire le programme répondant à votre problème le plus rapidement et le plus simplement possible. Il n'est pas aussi rapide qu'un langage comme le C mais il propose une approche simple qu'il est possible d'étoffer lorsque cela est nécessaire à l'aide d'autres langages plus rapides. Ce document n'a pas pour objectif d'étudier le langage *Python* de manière approfondie mais de proposer un outil informatique simple d'utilisation au service de travaux statistiques, économiques, financiers...

Chapitre 2

Type de variables du langage *Python*

2.1 Variables

2.1.1 Définition

Les variables permettent de manipuler des informations en les nommant. Elles jouent un rôle semblable aux inconnues dans une équation mathématique. Par exemple, on désire calculer la somme de n nombres $(u_1, \ldots, u_n)$. Or l'ordinateur ne sait bien souvent manipuler que deux nombres à la fois. Par conséquent, 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
for i in range(1,n) :       # pour tous les indices de 1 à n
    somme = somme + i       # on ajoute le ième élément à somme
```

> **Définition 2.1.1 : 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.
>
> *un type :* c'est une information sur le contenu de la variable qui indique au compilateur *Python* la manière de manipuler cette information.

Et avec les variables, leur contraire, les constantes :

> **Définition 2.1.2 : 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
str = "chaîne"    # création d'une variable chaîne de caractères appelée str
                  # initialisée à "chaîne"
```

Remarque 2.1.3: 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 3 à la variable x
```

Remarque 2.1.4: instruction sur plusieurs lignes
Le *Python* impose une instruction par ligne. Il est donc impossible de créer deux variables sur la même ligne, de réaliser deux affectations sur une même ligne. La réciproque est aussi vraie, il n'est pas non plus 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 lignes peuvent être écrites sur plusieurs de manière plus lisibles, de sorte qu'elles apparaissent en entier.

2.2 Types immuables (ou *immutables*)

> **Définition 2.2.1 : 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 `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 types immuables sont de taille fixe à l'exception des T-uples et des chaînes de caractères. Tous les autres types ne sont qu'un assemblage de types immuables.

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 utilisent cette convention lorsqu'il leur est impossible de retourner un résultat. Dans ce cas, il est possible de générer une erreur (ou exception, voir paragraphe 5), de retourner une valeur par défaut ou encore de retourner `None`. Il n'y a pas de choix meilleur que les autres, 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 = 3.5` crée une variable de type `float` initialisée à `3.5`. Le programme suivant permet de vérifier cela en affichant pour les variables `x` et `y`, leurs valeurs et leurs types respectifs.

```
x = 3
y = 3.5
print "x =", x, type(x)
print "y =", y, type(y)
```

Ce qui donne à l'écran :

```
x = 3 <type 'int'>
y = 3.5 <type 'float'>
```

Voici la liste des opérateurs qui s'appliquent aux nombres réels et entiers.

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	`x = y//3`
%	reste d'une division entière (modulo)	`x = y%3`
*= /=	multiplication ou division puis affectation	`x* = 3`
**	puissance	`x = y * *3`

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. De même, `7&2` s'écrit `1011&10 = 10`, qui vaut 2 en base décimale.

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)
print type(x), type(y)
```

Le programme précédent donne comme résultat :

```
<type 'int'> <type 'float'>
```

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

Remarque 2.2.2: division entière

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

L'opérateur // permet d'effectuer une division entière lorsque les deux nombres à diviser sont réels. Il existe néanmoins un autre cas pour lequel cet opérateur est implicite : lorsque la division opère sur deux nombres entiers ainsi que le montre l'exemple précédent. Pour obtenir le résultat juste incluant la partie décimale, il faut convertir les entiers en réels.

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

2.2.3 Booléen

Les booléens sont le résultat d'opérations logiques et ont deux valeurs possibles : `True` ou `False`.

```
x = 4 < 5
print x
print not x
```

Le programme précédent a pour résultat :

```
True
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`

Voici la liste des opérateurs de comparaisons qui retournent des booléens.

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-clé qui retournent un résultat de type booléens :

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

Ces deux opérateurs seront utilisés ultérieurement, `in` lors des boucles (paragraphe 3.3.2), `is` lors de l'étude des classes (chapitre 4). Bien souvent, les booléens sont utilisés de manière implicite lors de tests (paragraphe 3.2).

2.2.4 Chaîne de caractères

2.2.4.1 Création d'une chaîne de caractères

> **Définition 2.2.3 : chaîne de caractères**
> Le terme "chaîne de caractères" ou *string* en anglais signifie une suite finie de caractères, autrement dit, du texte. Une chaîne de caractères est une variable contenant 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.

```
# -*- coding: cp1252 -*-
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,
                        # il ne doit rien y avoir d'espace après le symbole \
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'affichages. *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, il faut pour cela ajouter le commentaire suivant $\#-*-$ `coding : cp1252` $-*-$ à la première ligne du programme. 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).

\"	guillemet
\'	apostrophe
\n	passage à la ligne
\\	insertion du symbole \
\%	pourcentage, ce symbole est aussi un caractère spécial
\t	tabulation

Tab. 2.1: Liste des des extra-caractères les plus couramment utilisés à l'intérieur d'une chaîne de caractères.

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, celles-ci sont regroupées 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
```

opérateur	**signification**	**exemple**
+	concaténation de chaînes de caractères	`t` = "`abc`" + "`def`"
[n]	obtention du $n^{ième}$ caractères, le premier caractère a pour indice 0	`t` = "`abc`" `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`" `print t[0 : 2] # donne ab`

Tab. 2.2: Opérations applicables aux chaînes de caractères.

Il existe d'autres fonctions qui permettent de manipuler les chaînes de caractères. Elles suivent la syntaxe des classes (voir chapitre 4) :

`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 présente une liste non exhaustive des fonctions disponibles dont un exemple d'utilisation suit.

```
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
```

`count(sub[, start[, end]])`	Retourne le nombre d'occurences de la chaîne de caractères `sub`, les paramètres par défaut `start` et `end` permet de réduit la recherche entre les caractères d'indice `start` et `end` exclus. Par défaut, `start` est nul tandis que `end` corresond à la fin de la chaîne de caractères.
`find(sub[, start[, end]])`	Recherche une chaîne de caractères `sub`, les paramètres par défaut `start` et `end` ont la même signication que ceux de la fonction `count`. Cette fonction retourne -1 si la recherche n'a pas abouti.
`isalpha()`	Recherche `True` si tous les caractères sont des lettres, `False` sinon.
`isdigit()`	Recherche `True` si tous les caractères sont des chiffres, `False` sinon.
`replace(old, new[, count])`	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne `sub` par `new`. Si le paramètre optionnel `count` est renseigné, alors seules les `count` premières occurrences seront remplacées.
`split([sep[, maxsplit]])`	Découpe la chaîne de caractères en se servant de la chaîne `split` comme délimiteur. Si le paramètre `maxsplit` est renseigné, au plus `maxsplit` coupures seront effectuées.
`upper()`	remplace les minuscules par des majuscules
`lower()`	remplace les majuscules par des minuscules

Tab. 2.3: Quelques fonctions s'appliquant aux chaînes de caractères, l'aide associée au langage *Python* fournira la liste complète. Certains des paramètres sont encadrés par des crochets, ceci signifie qu'ils sont facultatifs.

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. Le format est le suivant :

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

> `c1` est un code choisi parmi ceux de la table 2.4 (page 27). 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 :

```
# -*- coding: cp1252 -*-
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 :

$$\text{"\%n.df"\%x}$$

où **n** est le nombre de chiffres total et **d** est le nombre de décimales

Exemple :

```
x = 0.123456789
print x
print "%1.2f" % x
print "%6.2f" % x
```

Ce qui donne :

```
0.123456789
0.12
  0.12
```

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

2.2.5 T-uple

Définition 2.2.4 : T-uples
Les T-uples sont un tableau d'objets qui peuvent être de tout type. Ils ne sont pas modifiables.

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

Tab. 2.4: Liste non exhaustive des codes utilisés pour formater des informations dans une chaîne de caractères.

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

```
x = (4,5)                # création d'un T-uple composé de deux entiers
x = ("un",1,"deux",2)    # création d'un T-uple composé deux chaînes de caractères
                         # et de deux entiers, l'ordre d'écriture est important
x = (3,)                 # création d'un T-uple d'un élément, sans la virgule,
                         # le résultat est entier
```

Ces objets sont des vecteurs d'objets. Il est possible d'effectuer les opérations regroupées dans la table 2.5. Etant donnée que les chaînes de caractères sont également des tableaux, ces opérations reprennent en partie celles de la table 2.2. Quelques fonctionnalités sont néanmoins ajoutées par rapport à celles regroupées dans la table 2.5.

`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` , `n * s`	concatène `n` copies de `s` les unes à la suite des autres
`s[i]`	retourne une copie du $i^{\text{ème}}$ 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`, résultat difficile à prévoir lorsque les types des éléments sont différents
`sum(s)`	retourne la somme de tous les éléments

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

Remarque 2.2.5: T-uple, opérateur []
Les T-uples ne sont pas modifiables, cela signifie qu'il est possible 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 messsage 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
```

Remarque 2.2.6: not in
Pour vérifier la non appartenance d'un élément à l'intérieur d'un T-uple, on est parfois tenté d'écrire :

```
if not p in tuple :
   ...
```

La priorité de l'opérateur `not` est plus forte que celle de l'opérateur `in`, bien que syntaxiquement correct, il est préférable le test comme suit :

```
if p not in tuple :
   ...
```

Cette remarque s'applique également aux listes, aux dictionnaires et aux classes si elles ont définit des itérateurs.

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.

Il est impossible de dresser une liste exhaustive de ces types puisque le langage *Python* offre la possibilité de créer ses propres types (voir le chapitre 4, notamment le mot-clé `__slots__`). La plupart seront introduits par les différents modules qui seront intégrés au programme (voir chapitre 6).

2.3 Types modifiables (ou *mutable*)

2.3.1 Liste

2.3.1.1 Définition et fonctions

> **Définition 2.3.1 : 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 comprise entre crochets et séparées 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 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
```

Ces objets sont des listes chaînées d'objets. Il est possible d'effectuer les opérations regroupées dans la table 2.6. Ces opérations reprennent celle des T-uples (voir table 2.6) et incluent d'autres fonctionnalités puisque les listes sont modifiables (voir table 2.7). Il est donc possible d'insérer, de supprimer des éléments, de les trier. La syntaxe des opérations sur les listes que présentent la table 2.7 nécessitent la connaissance des classes présentées au chapitre 4.

`x in s`	vrai si `x` est un des éléments de `l`
`x not in s`	réciproque de la ligne précédente
`l + t`	concaténation de `l` et `t`
`l * n` , `n * s`	concatène `n` copies de `l` les unes à la suite des autres
`l[i]`	retourne l'élément i^{ème} élément de `l`, à la différence des T-uples, l'instruction `l[i]` = "`3`" est valide, elle remplace l'élément `i` par 3.
`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`, résultat difficile à prévoir lorsque les types des éléments sont différents
`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

Tab. 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`.

```
# -*- coding: cp1252 -*-
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
```

`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, une exception est déclenchée (voir le paragraphe 5).
`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 occurence 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 dont la valeur est d'abord retournée puis il est 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])`	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).

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

```
print x               # affiche la liste triée

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
```

Voici les trois listes affichées par cet exemple :

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

Comme cet élément n'existe pas, il 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).

```
# -*- coding: cp1252 -*-
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 "ça marche"
```

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 utilisés dans des boucles ou notamment par l'intermédiaire de la fonction **range**. Cette fonction retourne une liste d'entiers.

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

> Retourne une liste incluant tous les entiers compris entre **debut** et **fin** inclus. Si le paramètres facultatifs **marche** est renseigné, la liste contient tous les entiers **n** compris **debut** et **fin** exclu et tels que $\mathbf{n} - \mathbf{debut}$ soit un multiple de **marche**.

Exemple :

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

La fonction **xrange** est équivalente à **range** à ceci près qu'elle utilise moins de mémoire. En contrepartie, cette opacité est reflétée par l'impossibilité d'afficher le résultat de la fonction **xrange** sous forme de liste.

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

Cette fonction est souvent utilisée lors de boucles (voir paragraphe 3.3.2). 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 compris entre 1 et 20 exclu.

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

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])
```

Affiche :

```
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. Il s'agit d'effectuer un même traitement pour tous les éléments de la liste `x`.

```
x = ["un", 1, "deux", 2, "trois", 3]
for el in x :
    print "la liste inclut : ", el
```

Ce programme a pour résultat :

```
la liste inclut :  un
la liste inclut :  1
la liste inclut :  deux
la liste inclut :  2
la liste inclut :  trois
la liste inclut :  3
```

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`.

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

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

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

```
x = range(0,5)
y = [ i for i in x if i % 2 == 0]   # sélection les éléments pairs
print y                             # affiche [0,2,4]
z = [ i+j for i in x for j in x]    # 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, dictionnaire), toutes de longueur l. La fonction **zip** permet de construire une liste de T-uples qui est la juxtaposition de toutes ces séquences. La $i^{\text{ième}}$ 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 = 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 Copie

copie Les listes sont des objets et une affectation n'est pas équivalente à une copie comme le montre le programme suivant.

```
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]
```

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 4.4). Pour copier une liste, il faut utiliser la fonction **copy** du module **copy**.

```
l = [4,5,6]
import copy
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]
```

2.3.2 Dictionnaire

2.3.2.1 Définition et fonctions

> **Définition 2.3.2 : 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 séparées 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 pour cet exemple des chaînes de caractères. Pour associer la valeur associé à la clé "cle1", il suffit d'écrire :

```
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 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 associées aux dictionnaires.

`x in d`	vrai si `x` est une des clés de `d`
`x not in d`	réciproque de la ligne précédente
`l[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 l[i]`	supprime l'élément associée 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, particulier, `d` est égal à `dict(d.items())`

Tab. 2.8: Opérations disponibles sur les dictionnaires, `d` est un dictionnaire, `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

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 retourné à moins que le paramètre optionnel x soit renseigné, auquel cas, ce sera ce paramètres qui sera retourné.
d.clear()	Supprime tous les éléments du dictionnaire.
d.update(d2)	Pour chaque clé de d1, d[k] = d1[k]
d.setdefault(k[,x])	Retourne d[k] si la clé k existe, sinon, affecte x à d[k].
d.popitem()	Retourne un éléments et le supprime du dictionnaire.

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

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 à condition que celui-ci ne soit pas modifié lors du parcours. Un exemple de leur utilisation est présenté dans le paragraphe suivant.

2.3.2.2 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).

```
# -*- coding: cp1252 -*-
x = { "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.

```
# -*- coding: cp1252 -*-
d = { "un":1, "zéro":0, "deux":2, "trois":3, "quatre":4, "cinq":5, "six":6, "sept":1, \
       "huit":8, "neuf":9, "dix":10 }

dinv = dict ()
```

```
it = d.iteritems ()     # création d'un itérateur sur les couples (clé, valeur)
while True:             # ne change jamais lors d'un parcours avec itérateur
    try :               # ne change jamais lors d'un parcours avec itérateur,
                        # permet d'attraper une exception
        x = it.next ()  # affecte à x le couple (clé, valeur) désigné par l'itérateur
        dinv [ x [1] ] = x [0]
    except StopIteration :  # attrape l'exception qui signifie que tous les éléments
                            # du dictionnaire ont été visités
        break               # on sort donc de la boucle

print d                 # affiche le dictionnaire
print dinv              # affiche le dictionnaire inversé
```

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 est interprétable 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
```

Le module **pprint** inclut une fonction du même nom proposant des affichages plus lisibles des types comme les listes ou les dictionnaires. L'instruction **print** affiche les éléments d'une liste les uns à la suite des autres. Lorsque cette présentation devient illisible, la fonction **pprint** affiche un élément par ligne.

```
import pprint
x = ["un","deux","trois"]
x = x*4
print x
pprint.pprint (x)
```

Ce programme a pour résultat :

```
['un', 'deux', 'trois', 'un', 'deux', 'trois', 'un', 'deux
', 'trois', 'un', 'deux', 'trois']
['un',
 'deux',
 'trois',
```

```
 'un',
 'deux',
 'trois',
 'un',
 'deux',
 'trois',
 'un',
 'deux',
 'trois']
```

2.4.1.1 Copie

copie A l'instar des listes (voir paragraphe 2.3.1.6), 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 4.4). 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}
```

2.4.2 Informations fournies par *Python*

Bien que les fonctions ne soient définies que plus tard (paragraphe **??**), 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__', '', '', '', 'x']
```

Certaines variables, des chaînes des caractères existent déjà avant même la première instruction :

`__builtins__`	utilisée lors de la compilation d'un code défini lors de l'exécution du programme
`__doc__`	chaîne commentant le fichier, c'est une chaîne de caractères insérée aux premières du fichiers et entourés des symboles """.
`__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).

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

```
x = 3
print x, type(x)
x = 3.5
print x, type(x)
```

L'exemple précédent aboutit au résultat suivant :

```
3 <type 'int'>
3.5 <type 'float'>
```

Autre fonction intérassant, la fonction `eval` permet d'évaluer une chaîne de caractères. Le petit exemple suivant permet d'afficher le contenu de toutes les variables définies à cet instant du programme.

```
""" affiche toutes les variables """
x = 3
x = x ** eval ("x")
for s in dir ():
  print s, "  =  ", eval (s)
```

Ceci aboutit au résultat suivant :

```
__builtins__   =   <module '__builtin__' (built-in)>
__doc__   =    affiche toutes les variables
__file__   =   C:\temp\essai2.py
__name__   =   __main__
x   =   27
```

La première ligne du programme est une chaîne de caractères. Elle est facultative mais sa présence est considérée comme un commentaire associé au fichier contenant le programme. Cette convention est utile lors de la conception de modules (voir chapitre 6).

2.4.3 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 résultat comme la fonction `divmod` illustré 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 valeur du tuple `point`.

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

Chapitre 3

Syntaxe du langage *Python*

3.1 Les trois concepts des algorithmes

Les algorithmes sont composées 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
```

Si tout programme peut se résumer à ces trois concepts, l'aisance que l'on peut acquérir dans tel ou tel langage dépend de la syntaxe qui les met en place. Cette syntaxe permet de transcrire des algorithmes rédigés en français comme celui de l'exemple ci-dessus pour aboutir au programme suivant, écrit en *Python* et interprétable par l'ordinateur via le compilateur associé à ce langage *Python*.

```
moy = 0
for i in range(1,N):
  moy += n [i]
moy /= N
```

Le premier élément de cette syntaxe est constitué de ses mots-clé (voir table 3.1) et des symboles (voir table 3.2) dont certains interviennent lors de l'écriture des tests, des boucles et des fonctions. La fonction `iskeyword` du module `keyword` 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
```

3.2 Tests

> **Définition 3.2.1 : test**
> Les tests permettent d'exécuter des instructions différentes selon la valeur d'une condition logique.

3.2.1 Syntaxe

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

La clause `else` est facultative, lorsque la condition `condition1` est fausse est 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
   ...
```

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

Tab. 3.1: Mots-clé du langage *Python*.

S'il est nécessaire d'enchaîner plusieurs tests d'affilée, il est possible de condenser l'écriture :

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

Remarque 3.2.2: indentation
Le décalage des instructions par rapport aux lignes contenant les mots-clé `if`, `elif`, `else` est très important : il fait partie de la syntaxe du langage et s'appelle l'*indentation* (voir paragraphe 3.6). Celle-ci permet de grouper les instructions ensemble. Le programme suivant est syntaxiquement correct, toutefois, la dernière ligne n'est pas correctement indentée.

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

Le programme répond de manière erronnée :

```
le nombre est positif
```

=	+	-	*	/
:	+=	-=	*=	/=
#	**	"	'	//
%	~	"""	"'	\
[	]	(	)	,
<	>	<=	>=	.
^	!	==	!=	&
«	»	|	{	}

Tab. 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.

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

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 lesquels 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
`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

3.2.4 Ecriture condensée

Il existe deux écritures de tests condensées. 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 est équivalent à :

```
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
   print "pas de signe"
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
   print "pas de signe"
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 ("dit oui : ")
if s == "oui" or s [0:1] == "o" or s [0:1] == "O" or s == "1" :
    print "oui"
else :
    print "non"
```

3.2.6 Passer, instruction pass

Dans certains cas, aucune instruction ne doit être exécuté 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. Selon les éditeurs, une boîte de dialogue contenant un message d'erreur peut se déclencher ou simplement afficher un message d'erreur (voir figure 3.1).

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

3.3 Boucles

Définition 3.3.1 : boucle
Les boucles permettent de répéter une séquence d'instructions tant qu'une certaine condition est vérifiée.

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


```
File"cours.py", line6
  else :
  ^
IndentationError : expected an indented block
```

Fig. 3.1: Entre `elif` et `else`, le compilateur demande une ligne indentée. Soit, une boîte de dialogue apparaît lorsque l'instruction `pass` est manquante, soit le compilateur génère un message d'erreur.

Le langage *Python* propose deux types de boucles. La boucle `while` suit scrupuleusement la définition 3.3.1. 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 même 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 :

```
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 3.2.2 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.

```
# -*- coding: cp1252 -*-
n = 0
while n < 5:
   print "à l'intérieur ", n
   n += 1
print "à l'extérieur ", n
```

Ce petit programme affiche à l'écran :

```
à l'intérieur  0
à l'intérieur  1
à l'intérieur  2
à l'intérieur  3
à l'intérieur  4
à l'extérieur  5
```

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` 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 < 5` toujours vraie.

```
n = 0
while n < 5 :
   print n
   n + 1          # l'instruction correcte est n += 1, n n'est jamais modifié
```

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 :

```
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.

Tout comme les tests, la remarque 3.2.2 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
```

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 d'un dictionnaire dans un ordre croissant.

```
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 triés [8, 1, 3, 5, 7]
for x in k:                # affiche les éléments du dictionnaire
    print x,":",d [x]      # triés par clés croissantes
```

La boucle certainement 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.

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

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 :

[**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.

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

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

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

D'autres exemples de cette syntaxe réduite, ils impliquent des tests, des boucles imbriquées.

```
x = range(0,5)
y = [ i for i in x if i % 2 == 0]   # sélection les éléments pairs
print y                             # affiche [0,2,4]
z = [ i+j for i in x for j in x]    # 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]
```

Cette écriture condensée est bien souvent plus lisible que la première et c'est pourquoi elle lui est préférée. Tout dépend des préférences de celui qui programme.

3.3.2.3 Itérateurs

Toute boucle `for` peut s'appliquer sur un objet muni d'itérateurs tels que les 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
```

L'exemple précédent est convertible en une boucle `while` en faisant apparaître explicitement les itérateurs (voir paragraphe 4.2.3). 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 pargraphe 2.4.3). 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 suivant :

```
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.

```
# -*- coding: cp1252 -*-
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
```

Ce programme affiche :

```
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
```

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 pour dénicher tous les nombres premiers compris entre 1 et 99.

```
d = dict ()
for i in range(1,100):
    d [i] = True                 # d [i] est vrai si i est un nombre premier

for i in range(2,100):

   if not d [i]: continue        # si d [i] est faux,
                                 # les multiples de i ont déjà été cochés
```

```
   for j in range (2,100):
       if i*j >= 100: break       # d [i*j] est faux pour tous les multiples de i
       d [i*j] = False

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                  # d [i] est vrai si i est un nombre premier,
                                  # au départ, tous les nombres sont premiers

for i in range(2,100):
   if d [i]:                      # les lignes qui suivent sont décalées d'un cran
       for j in range (2,100):    # supplémentaire
           if i*j >= 100: break
           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 lesquels il est nécessaire d'arrêter l'exécution de cette boucle. De même, pour une boucle `for`, il n'est pas toujours utile de 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 plus 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 instructin n'est pas exécutée
print "l'élément ",c," est en position ",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 second instruction break est nécessaire pour sortir
                          # de la seconde boucle lorsque la solution a été trouvé
if c == n:
    print n, " est la somme des carrés de deux entiers :", \  # le symbol \ permet
          x, "*", x, "+", y, "*", y, "=", n                   # de passer à la ligne
else:                                                         # sans changer d'instruction
    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 ne sont exécutées que si aucune instruction **`break`** n'a été exécutée 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
```

En fait, 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`**, une instruction **`return`** (voir les fonctions, paragraphe 3.4), par la levée d'une exception (voir paragraphe 5) ou tout simplement lorsque l'exécution de la boucle s'est achevée.

3.3.4.4 Suppression 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 modifié 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]
```

3.4 Fonctions

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.

3.4.1 Définition

3.4.1.1 Syntaxe

> **Définition 3.4.1 : fonction**
> Une fonction est une partie d'un programme - ou sous-programme - qui fonctionne indépendamment du reste du programme, qui reçoit une liste de paramètres et qui retourne un résultat. Le corps de la fonction désigne toute instruction du programme qui ne peut être exécutée que si la fonction est appelée.

Les fonctions sont couramment utilisées afin de ne pas recopier le code associé à une tâche couramment répétée. Elles rendent également le programme plus lisible dans la mesure où il est un assemblage de fonctions les plus courtes possibles. D'une manière générale, on évite les grandes fonctions pour préférer la scinder en plusieurs petites fonctions, chacune affectée à une tâche précise.

```
def fonction_nom (param_1, param_2, ..., param_n) :
    instruction_1
    instruction_2
    ...
    instruction_n
    return resultat_1, ..., resultat_n
```

Où `fonction_nom` est le nom de la fonction. Celui-ci commence par une lettre, peut inclure des chiffres mais ne doit pas contenir de lettres accentuées. `param_1` à `param_n` sont les noms des paramètres, `resultat_1` à `resultat_n` sont les résultats retournés par la fonction. S'il n'y en a aucun, l'instruction `return` est facultative. Cette instruction peut apparaître plusieurs 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. Les instructions associées à une fonctions doivent être indentées par rapport au mot-clé `def`.

Pour exécuter une fonction ainsi définie, il suffit de suivre la syntaxe suivante :

```
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.

3.4.1.2 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.

```
# -*- coding: cp1252 -*-
import math
def coordonnees_polaires (x,y):
    rho     = math.sqrt(x*x+y*y)
    theta   = math.atan2 (y,x)
    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)
```

3.4.1.3 Paramètres avec des valeurs par défaut

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

Exemple :

```
# -*- coding: cp1252 -*-
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
    # ...
```

3.4.1.4 Ordre des paramètres

Le paragraphe 3.4.1.1 a présenté la syntaxe d'appel a une fonction. Lors de l'appel, le nom des paramètres n'intervient plus, supposant que chacun 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.

$$\texttt{x\_1}, ..., \texttt{x\_n} = \texttt{fonction\_nom}\ (\texttt{param\_1} = \texttt{valeur\_1}, ..., \texttt{param\_n} = \texttt{valeur\_n})$$

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 cet syntaxe, l'ordre d'écriture n'importe pas. La valeur `valeur_i` sera toujours attribuée à `param_i`. `x_1` à `x_n` reçoivent les résultats retournés par la fonction. Cette affectation est facultative.

Exemple :

```
# -*- coding: cp1252 -*-
def identite (nom, prenom):
    print "nom : ", nom, " prénom : ", prenom

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

```
# -*- coding: cp1252 -*-
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
```

3.4.1.5 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)
```

3.4.2 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` défini dans l'exemple du paragraphe précédent 3.4.1.2, le message affiche 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.

```
# -*- coding: cp1252 -*-
import math
def coordonnees_polaires (x,y):
    """convertit des coordonnées cartésiennes
    (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
    (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.

3.4.3 Paramètre modifiable, paramètre immuable

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 répercussion 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 n'appelle ceci un passage de paramètre par valeur. A l'opposé, toute modification d'une variable d'un type modifiables à l'intérieur d'une fonction est répercutée à la variable qui a été utilisée lors de l'appel de cette fonction.

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.

```
# -*- coding: cp1252 -*-
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 modifiable et 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 paramètree `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 3.4.2: passage par adresse
Dans l'exemple précédent, il faut faire attention 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 appelée `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 4.4.

```
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[0:len(liste)] = []

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

3.4.4 Fonction récursive

> **Définition 3.4.3 : 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. De plus, *Python* n'autorise pas plus de 1000 appels récursifs, autrement dit `factorielle(1001)` provoque nécessairement une erreur d'exécution même si la condition d'arrêt est bien spécifiée.

3.4.5 Portée

3.4.5.1 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.

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

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 y ait confusion.

```
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 3.4.4 : portée d'une variable**
> La portée d'une variable associé 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 3.4.5 : 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 3.4.6 : variable globale**
> Une variable gloable est une variable dont la portée est l'ensemble du programme.

L'exemple suivant mélange variable locale et variable globale. L'identificateur `n` est utilisée 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.

3.4.5.2 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 celle 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 imbriquées
        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.

3.4.6 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 présentés 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 :

`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, `dictionnaire` reçoit la liste des couples (identificateurs, valeurs).

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

```
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 associés aux paramètres `param_1` à `param_n`, `liste_valeur_1` à `liste_valeur_p` formera 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 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 paramètres commencent la liste, suivent les valeurs associées à un nom de paramètre.

3.4.7 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`.

> `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
```

3.4.8 Fonctions générateurs

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 utilisés 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"
        yield i         # arrête la fonction qui reprendra
        i = i+1         # à la ligne suivante lors du prochain appel
```

```
    print "yield 2"
    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(5):
    print a                 # affiche tous les éléments que retourne la
                            # fonction fonction_yield, elle simule la liste
                            # [0,1,2,3,4]
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 1
1
yield 1
2
yield 1
3
yield 2
4
--------------------------------------------
yield 1
0
yield 1
1
yield 2
2
```

3.4.9 Appelable

La fonction `callable` retourne un booléen permettant de savoir si un identificateur est une fonction ou une classe (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
```

3.4.10 Fonctions ajoutées lors de l'exécution du programme

3.4.10.1 fonction `eval`

Cette fonction a déjà été abordée lors des paramètres 2.4.1 et 2.4.2. 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 envoyé à la fonction **eval** inclus 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
```

3.4.10.2 fonctions `compile`, `exec`

La fonction **compile** permet d'ajouter une ou des 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**.

```
# -*- coding: cp1252 -*-
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 Commentaires

Lorsque les commentaires incluent des symboles exclusivement français tels que les accents, le compilateur génère l'erreur suivante :

```
sys:1: DeprecationWarning: Non-ASCII character '\xe9' in file C:\temp\essai2.py on line 1,
but no encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Il est néanmoins possible d'utiliser des accents dans les commentaires à condition d'insérer le commentaire suivant à la première ligne du programme. Il n'est pas nécessaire de retenir cette commande si le programme est écrit dans l'éditeur de texte fourni avec *Python* car ce dernier propose automatiquement d'insérer cette ligne.

```
# -*- coding: cp1252 -*-
```

3.6 Indentation

L'indentation est synonyme de décalage. Pour toute boucle, test, fonction, et plus tard, 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.

Contraitement 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.

3.7 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.

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

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

def calcul_n_valeur (l,f):
    res = []
    for i in l:
        t = f (i)
        res.append (t)
    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 détient un fichier contenant des informations sur ses clients. Il est 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 détenues par sa banque le concernant. 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, les informations dont elle dispose mais propose des services à ses utilisateurs.

Les classes sont la concrétisation informatique du concept d'*objet*. Ce terme désigne une entité possédant des données et des méthodes permettant de les manipuler. Plus concrètement, une classe est un assemblage de variables appelés *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

4.1.1 Définition, déclaration

> **Définition 4.1.1 : 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.

```
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.

Il est néanmoins possible de les définir au moyen des classes. Une matrice est par exemple un objet qui inclut comme attributs :

- 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
- 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.

En résumé, les classes permettent de définir de nouveaux types. Pour créer une variable d'une classe donnée, il suffit d'appliquer la syntaxe suivante.

> `cl` = `nom_classe`()
>
> La création d'une variable de type objet est identique à celle des types standard du langage *Python*. On dit aussi que `cl` est 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 si le constructeur de la classe est modifié (voir paragraphe 4.2). .

> **Définition 4.1.2 : instance**
> Une instance d'une classe `C` désigne simplement une variable de type `C`. Le terme instance ne s'applique qu'aux variables de type objet.

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 4.1.3: classe sans attribut, sans méthode, et non vide
Il est possible d'insérer du code dans la classe alors que celle-ci n'inclut aucune méthode ni aucun attribut.

Ces lignes écrites hors de toute méthode sont exécutées lors de la création d'une instance de la classe comme le montre l'exemple suivant.

```
class classe_vide:
    print "création d'une instance de la classe classe_vide"
cl = classe_vide () # affiche création d'une instance de la classe classe_vide
```

Cette écriture n'est pas conseillé, il est préférable d'insérer ces lignes hors méthode à l'intérieur d'un constructeur (voir paragraphe 4.2).

Remarque 4.1.4: 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(nb)   # affiche <type 'instance'>
```

Pour savoir si une variable est une instance d'une classe donnée, il faut utiliser la fonction `isinstance` :

```
isinstance(nb,exemple_classe)   # affiche True
```

4.1.2 Méthodes

> **Définition 4.1.5 : méthode**
> Les méthodes sont des fonctions qui sont associées de manière explicite à une classe. Les attributs de la classe se comportent comme des variables globales pour toutes ses méthodes.

Les méthodes sont en fait des fonctions pour lesquelles la liste des paramètres contient obligatoirement un paramètre implicite qui est l'instance de la classe à laquelle cette méthode est associée. Ce paramètre (`self`) n'apparaît pas lors de l'appel.

```
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 :

```
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 4.1.6: 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 du 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 4.1.7 : 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 d'une 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 toujours défini par ces deux extrémités.

```
class nom_classe :
    nom_attribut = valeur
    def nom_methode(self, param_1, ..., param_n) :
        self.nom_attribut = valeur
```

Le mot-clé **self** doit être impérativement utilisé pour désigner un attribut à l'intérieur d'une méthode. Dans le cas contraire, **nom_attribut** désigne une variable locale sans aucun rapport avec un attribut portant le même nom. Les attributs peuvent être déclarés n'importe où, à l'intérieur d'une méthode précédés du mot-clé **self**, à l'extérieur d'une méthode mais à l'intérieur de la classe, à l'extérieur de la classe précédés du nom de l'instance de cette classe.

L'exemple suivant reprend l'exemple du paragraphe précédent mais cette fois-ci en définissant la variable **rnd** non plus comme variable globale mais comme attribut de la classe **exemple_classe**. A chaque création d'une instance de cette classe, la première valeur de la variable **rnd** est 42. Par conséquent, les instances **l** et **l2** retournent maintenant les mêmes suites de nombres aléatoires.

```
class exemple_classe:
    rnd = 42               # rnd est maintenant un attribut de la 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   # 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]
```

L'endroit où est déclaré l'attribut **rnd** a peu d'importance pourvu qu'il le soit avant sa première utilisation. Par exemple, dans l'exemple suivant, 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**.

1. Le premier endroit correspond au premier programme de ce paragraphe, c'est-à-dire en tête de classe.
2. Mais il est tout-à-fait possible d'écrire la même ligne après la méthode **methode1** comme dans l'exemple suivant. Lorsque l'instance **nb** de la classe **methode_classe** est créée, elle exécute toutes les lignes en dehors de toute méthode.

   Il n'est pas conseillé d'écrire des lignes de codes à l'intérieur d'une classe et à l'extérieur d'une méthode (voir paragraphe 4.2).

   ```
   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)
       rnd = 42

   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]
   ```

3. Il est aussi 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]
   ```

4. 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]
```

4.1.4 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 inclut la classe (voir chapite 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 4.1.5).
`__doc__`	contient un commentaire associé à la classe (voir paragraphe 4.1.7).

L'attribut `__class__` contient lui même d'autres d'attributs :

`__doc__`	contient un commentaire associé à la classe (voir paragraphe 4.1.7).
`__dict__`	contient la liste des méthodes et des attributs définis hors d'une méthode (voir paragraphe 4.1.6).
`__name__`	contient le nom de l'instance.
`__bases__`	contient les classes dont la classe de l'instance hérite (voir paragraphe 4.6).

```
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
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 ()
```

4.1.5 Liste des attributs

Chaque attribut d'une instance de classe est inséré dans un dictionnaire associé `__dict__`, attribut implicitement présent dès la création d'une instance. Il est aussi possible pour créer un attribut de l'insérer à ce dictionnaire qui est un attribut implicitement créé pour toutes les instances de classe.

```
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
print nb.__dict__        # affiche {'rnd': 42}
```

Il existe donc une dernière manière d'ajouter un attribut à une classe, c'est de l'ajouter à l'attribut `__dict__` de l'instance `nb`.

```
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.__dict__ ["rnd"] = 42 # déclaration par insertion au dictionnaire
print nb.__dict__
l = [nb.methode1(100) for i in range(0,10) ]
print l  # affiche [19, 46, 26, 88, 44, 56, 56, 26, 0, 8]
```

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 suivant ne réinitialise sa valeur à 42.

```
class exemple_classe:
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire compris entre 0 et n-1 inclus"""
        if not self.__dict__.has_key ("rnd"):  # on teste si l'attribut rnd existe
            self.rnd = 42                      # dans le cas contraire, on l'ajoute,
                                               # cette ligne, n'est exécutée qu'une
                                               # seule fois pour chaque instance
        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]
```

Remarque 4.1.8: attributs inclus dans `__dict__`
Le langage *Python* propose parfois quelques bizarreries comme le programme qui suit. Selon qu'un attribut déclaré en dehors d'une méthode reçoit ou non une valeur à l'intérieur d'une méthode, il sera ou ne sera pas inclus dans le dictionnaire `__dict__` (voir également la remarque 4.1.9).

```
class exemple_classe:
    rnd = 42
    def incremente_rnd (self):
```

```
        self.rnd += 1
        return self.rnd

cl = exemple_classe()

print "cl.__dict__           = ", cl.__dict__  # affiche {}
cl.incremente_rnd()
print "cl.__dict__           = ", cl.__dict__  # affiche {'rnd': 43}
```

4.1.6 Liste des méthodes

Il existe aussi un dictionnaire des méthodes associées à une classe. Il contient la liste des méthodes ainsi que les attributs définis à l'intérieur de la classe mais à l'extérieur de toute méthode.

```
class exemple_classe:
    attr = 0
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire compris entre 0 et n-1 inclus"""
        if not self.__dict__.has_key ("rnd"):  # on teste si l'attribut rnd existe
            self.rnd = 42                      # dans le cas contraire, on l'ajoute,
                                               # cette ligne n'est exécutée qu'une
                                               # seule fois pour chaque classe
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)

nb = exemple_classe ()
print nb.__class__.__dict__
```

Ce petit programme affiche :

```
{'__module__': '__main__', '__doc__': None,
'attr': 0, 'methode1': <function methode1 at 0x00A6C130>}
```

Ce dictionnaire contient également deux autres clés. `__module__` contient le nom du module dans lequel la classe est incluse (voir chapitre 6). `__doc__` contient un commentaire associé à la classe (voir paragraphe 4.1.7). Il est possible d'ajouter des méthodes à une classe en modifiant ce dictionnaire (voir paragraphe 4.3).

Remarque 4.1.9: attributs déclarés hors d'une méthode
Le langage *Python* propose une autre bizarrerie lié à la liste des méthodes. Celle-ci inclut également les attributs déclarés hors d'une méthode (voir également la remarque 4.1.8).

```
class exemple_classe:
    rnd = 42
    def incremente_rnd (self):
        self.rnd += 1
        return self.rnd

cl = exemple_classe()
```

```
print "cl.__dict__           = ", cl.__dict__  # affiche {}
cl.incremente_rnd()
print "cl.__dict__           = ", cl.__dict__  # affiche {'rnd': 43}
```

4.1.7 Commentaires

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 associée à la classe, la liste des méthodes ainsi que chacun des commentaires qui lui leur sont associés. Ce commentaire est affecté à l'attribut implicite `__doc__`.

```
class exemple_classe:
    """simule une suite de nombres aléatoires"""
    def methode1(self,n):
        """simule la génération d'un nombre aléatoire compris entre 0 et n-1 inclus"""
        if not self.__dict__.has_key ("rnd"):
            self.rnd = 42
        self.rnd = 397204094 * self.rnd % 2147483647
        return int (self.rnd % n)
nb = exemple_classe ()
help (exemple_classe)
```

Le programme précédent décrit intégralement la classe en affichant les lignes suivantes :

```
Help on class exemple_classe in module __main__:

class exemple_classe
 |  simule une suite de nombres aléatoires
 |
 |  Methods defined here:
 |
 |  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 la ligne suivante :

```
print exemple_classe.__doc__  # affiche simule une suite de nombres aléatoires
```

Cette ligne est également équivalente à l'une ou l'autre des lignes suivantes.

```
print nb.__doc__
print nb.__class__.__doc__
```

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.

```
class essai_class:
    x = 5
    def meth(self):
        x = 6
        self.y = 7

a = essai_class()
print dir (a)             # affiche ['__doc__', '__module__', 'meth', 'x']
a.meth ()
print dir (a)             # affiche ['__doc__', '__module__', 'meth', 'x', 'y']
print dir (essai_class)   # affiche ['__doc__', '__module__', 'meth', 'x']
```

La fonction `dir` appliquée à la classe elle-même retourne une liste qui n'inclut pas les attributs déclarés à l'intérieur d'une méthode.

4.1.8 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 4.2.3). 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 :
        x = 0
        y = 0
        z = 0

    all = [ 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()`.

4.2 Constructeurs, opérateurs, itérateurs

Le paragraphe 4.1.1 introduit la syntaxe permettant de créer une instance de classe. Celle-ci se présente comme une fonction puisqu'elle fait intervenir des parenthèses. Dans l'exemple qui suit, il est impossible de savoir si `nouvelle_classe` est une classe ou une fonction à moins de connaître sur sa définition.

```
class nouvelle_classe:
    pass
x = nouvelle_classe ()
```

Et tout comme les fonctions, la création d'instance d'une classe peut faire intervenir des paramètres en définissant ce qu'on appelle un *constructeur*. De même, a priori, il est impossible d'utiliser une syntaxe telle que :

```
class nouvelle_classe:
    pass
x = nouvelle_classe () + nouvelle_classe ()
```

Et pourtant, les classes en langage *Python* sont conçues de telles sortes que cette écriture et bien d'autres sont possibles en définissant ce qu'on appelle un *opérateur*.

4.2.1 Un objet qui ressemble à une fonction

Le constructeur d'une classe se présente comme une méthode et suit la même syntaxee à ceci près que son nom est imposé : `__init__`. Hormis le premier paramètre, invariablement `self`, il n'existe pas de contraintes concernant la liste des paramètres. Le constructeur ne doit pas retourner de résultat.

```
class nom_classe :
    def __init__(self,param_1,...,param_n) :
        # code du constructeur
```

`nom_classe` est une classe, `__init__` est son constructeur et `param_1` à `param_n` sont les paramètres du constructeur. L'instruction `return` ne doit pas figurer à l'intérieur du corps du constructeur car celui-ci ne retourne pas de résultat.

La modification des paramètres du constructeur implique également la modification de la syntaxe de création d'une instance de cette classe.

```
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`.

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 supplémentaires. 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.

```
# -*- coding: cp1252 -*-
class classe1:
    def __init__(self):
        # pas de paramètres supplémentaires
        print "constructeur de la classe classe1"
        self.n = 1

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

x = classe2(5,10)  # affiche constructeur de la classe classe2
print x.n          # affiche 7
```

Le constructeur est exécuté lors de la création d'une instance, tout comme le code inclus dans une classe mais hors de toute méthode (voir la remarque 4.1.3, page 68). Toutefois, il est préférable de placer ce code dans un constructeur afin de rendre le programme plus lisible.

Remarque 4.2.1: constructeur et fonction
Le constructeur est avant tout une méthode, sa syntaxe de déclaration est aussi souple que celle des fonctions et autorise les paramètres par défaut, son appel selon un ordre différent des paramètres, un nombre variable de paramètres, autant de fonctionnalités présentés au paragraphe 3.4.

Remarque 4.2.2: constructeur, apparence d'une fonction
La création d'une instance suit la même syntaxe qu'une fonction et pourrait être considérée comme une fonction à ceci près que le type du résultat est imposé par la classe.

4.2.2 Opérateurs

Le langage *Python* offre la possibilité de définir des opérations sur des instances d'une classe. Le programme suivant contient une classe définissant un nombre complexe. La méthode `ajoute` définit ce qu'est une addition entre nombre complexe.

```
class nombre_complexe:
    def __init__ (self, a = 0, b= 0):
        self.a = a
        self.b = 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 *opérateur* sont encadrés 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.

```
class nom_class :
    def __add__(self, autre) :
        # corps de l'opérateur
        return...# retourne un objet de typenom_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 = a
        self.b = 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)

    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 est maintenant syntaxiquement correcte
print c.a, c.b
```

Chaque opérateur possède sa méthode clé associée. L'opérateur $+=$, différent de $+$ est associée à la méthode-clé `__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`. 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 = a
        self.b = 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
print c1.a, c1.b
```

Un autre opérateur souvent utilisé est `__str__` qui permet de redéfinir l'affiche d'un objet lors d'un appel à l'instruction `print`.

```
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 = a
        self.b = 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 + %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)
c1 += c2
print c1    # affiche 1.000000 + 1.000000 i
```

Il existe de nombreux opérateurs qu'il est possible de définir. La table 4.1 (page 83) 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.

L'exemple suivant définit un point de l'espace avec trois coordonnées. Il redéfinit les opérateurs `__getitem__` et `__setitem__` de manière à pouvoir accéder aux coordonnées comme cette classe était une liste de trois coordonnées. En règle général, lorsque les indices ne sont pas correct, 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
        raise IndexError, "indice impossible, 0,1,2 autorisés"

    def __setitem__(self,i,x):
        if i == 0 : _x = x
        if i == 1 : _y = y
        if i == 2 : _z = z
        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)
print "abscisse : ", a [0]   # affiche abscisse :  1
print "ordonnée : ", a [1]   # affiche ordonnée :  -2
print "altitude : ", a [2]   # affiche altitude :  3
```

Par le biais de l'exception `IndexError`, les expressions `a[i]` avec `i` $\neq 1, 2, 3$ sont impossibles et arrêtent le programme par un message comme celui qui suit :

```
Traceback (most recent call last):
  File "cour2.py", line 29, in ?
```

```
    print a [4]
  File "cour2.py", line 12, in __getitem__
    raise IndexError, "indice impossible, 0,1,2 autorisés"
IndexError: indice impossible, 0,1,2 autorisés
```

`__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`.
`__divmod__(self,x)`	Cet opérateur est appelé par la fonction `divmod`.
`__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.

`__int__(self)`
`__float__(self)`
`__complex__(self)`
	Ces opérateurs implémente la conversion de l'instance `self` en entier, réel ou complexe.

`__add__(self,x)`
`__div__(self,x)`
`__floordiv__(self,x)`
`__mul__(self,x)`
`__sub__(self,x)`
`__pow__(self,x)`
	Opérateurs appelés pour les opérations $+$, $/$, $//$, $*$, $-$, $**$

`__iadd__(self,x)`
`__idiv__(self,x)`
`__ifloordiv__(self,x)`
`__imul__(self,x)`
`__isub__(self,x)`
`__ipow__(self,x)`
	Opérateurs appelés pour les opérations $+=$, $/=$, $//=$, $*=$, $-=$, $**=$

Tab. 4.1: Opérateurs les plus utilisés.

4.2.3 Itérateurs

L'opérateur `__iter__` permet de définir ce qu'on appelle un itérateur. Un itérateur est un objet qui permet d'en explorer un autre. Un itérateur doit nécessairement contenir une référence à l'objet qu'il doit explorer, il doit également inclure une méthode `next` qui retourne l'élément suivant ou lève une exception s'il n'existe pas.

Par exemple, on cherche à explorer tous les éléments d'un objet de type `point_espace` défini au paragraphe précédent, page 82. 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 48).

```
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)

    # cette classe définit un itérateur pour point_espace
    class class_iter:

        # initialisation, self._ins permet de savoir quelle
        # instance de point_espace on explore, self._n
        # mémorise l'indice de l'élément exploré
        def __init__ (self,ins):
            self._n   = 0
            self._ins = ins

        # retourne l'élément d'indice self._n
        # et passe à l'élément suivant
        def next (self):
            if self._n <= 2:
                v = self._ins [self._n]
                self._n += 1
                return v
            else :
                # si cet élément n'existe pas, lève une exception
```

```
                raise StopIteration

    # opérateur de la classe point_espace, retourne un itérateur
    # permettant de l'explorer
    def __iter__(self):
        return point_espace.class_iter (self)

    def __getitem__ (self, n) :
        if n == 0 : return self._x
        elif n == 1 : return self._y
        elif n == 2 : return self._z
        else : raise IndexError

a = point_espace (1,-2,3)
for x in a:
    print x      # affiche successivement 1,-2,3

# ----- autre solution ------
a = point_espace (1,-2,3)
it = iter (a)
while True:
    try :
        print it.next ()  # affiche successivement 1,-2,3
    except StopIteration :
        break
```

4.3 Méthodes statiques et ajout de méthode

4.3.1 Méthode statique

> **Définition 4.3.1 : 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.

```
# -*- coding: cp1252 -*-
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`.

```
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.

```
# -*- coding: cp1252 -*-
class essai_class:
    def methode ():
        print "méthode statique"
    methode = staticmethod(methode)

essai_class.methode ()
```

Remarque 4.3.2: déclaration d'une méthode statique à l'extérieur de la classe
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).

```
# -*- coding: cp1252 -*-
def methode ():
    print "méthode statique"

class essai_class:
    pass

essai_class.methode = staticmethod(methode)
essai_class.methode ()
```

4.3.2 Attribut statique

> **Définition 4.3.3 : attribut statique**
> Les attributs statiques sont des attributs qui peuvent être utilisés même si aucune instance de la classe où ils sont définies n'a été créée. Ces attributs sont partagés par toutes les instances.

Les attributs statiques peuvent être déclarés à l'extérieur d'une méthode mais dans le corps de la classe où à l'intérieur d'une méthode précédé du nom de la classe. La déclaration d'un paramètre statique suit la syntaxe suivante :

```
class nom_class :
    attribut_statique = valeur
    def nom_methode(self, params, ...) :
        nom_class.attribut_statique2 = valeur2

    def nom_methode_st(self, params, ...) :
        nom_class.attribut_statique3 = valeur3
```

`nom_classe` est une classe, `nom_methode` est une méthode non statique, `nom_methode_st` est une méthode statique.

Remarque 4.3.4: ambiguïté
Lorsqu'un attribut est défini hors de toute méthode, il peut être un paramètre statique ou non selon l'utilisation qui en est faite. Dans le premier exemple, `x` est déclarée hors de toute méthode. Pourtant, il est utilisé comme un attribut non statique par la méthode `meth`. Par conséquent, les deux instructions `y.meth()` et `z.meth()` impliquent les attributs `y.x` et `z.x` qui ne désignent pas la même variable.

```
class essai_class:
    x = 5
    def meth(self):
        print self.x
        self.x += 1

y = essai_class ()
z = essai_class ()
y.meth()   # affiche 5
z.meth()   # affiche 5
```

Pour le programme suivant, la méthode `meth` n'utilise plus `self.x` mais `essai_class.x`. L'attribut `x` est alors un attribut statique, partagée par toutes les instances. C'est pourquoi 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
```

Il n'est pas conseillé de mélanger des attributs statiques et non statiques portant le même nom. L'exemple ne suit pas cette recommandation. La classe `essai_class` utilise l'identificateur `x` pour un attribut statique et un attribut non

```
class essai_class:
    x = 5
    def meth(self):
```

```
        print "essai_class.x = ", essai_class.x
        print "self.x = ", self.x
        essai_class.x += 1
        self.x = -5

y = essai_class ()
z = essai_class ()
y.meth()
z.meth()
y.meth()
z.meth()
```

Le programme affiche les lignes suivantes :

```
essai_class.x =  5
self.x =  5
essai_class.x =  6
self.x =  6
essai_class.x =  7
self.x =  -5
essai_class.x =  8
self.x =  -5
```

4.3.3 Ajout de méthode

Le langage *Python* 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`.

```
def nom_methode(cls) :
    # code de la fonction
class nom_class :
    # code de la classe
    nom_methode = classmethod(nom_methode)   # syntaxe 1
nom_methode.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_class:
    x = 5
    def meth(self):
        print "ok meth", self.x

    def meth2(cls):
        print "ok meth2", cls.x

    meth2 = classmethod (meth2)
    meth3 = classmethod (meth3)

x = essai_class ()
x.meth ()
x.meth2 ()
x.meth3 ()

essai_class.meth4 = classmethod (meth4)
x.meth4 ()
```

4.3.4 Propriétés

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 fait appelle une méthode qui calcule sa valeur. A chaque fois que le programmeur cherche à modifier la valeur de ce faux attribut, il fait appelle à une autre méthode qui modifie l'instance.

```
class nom_class :
    # code de la classe
    nom_propriete = property(fget, fset, fdel, doc)
```

`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`. `fdoc` 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)`.

Le programme suivant illustre l'utilisation des propriétés. Il s'agit d'une classe `nombre_complexe` qui ne contient que les parties réelle et imaginaire. Lorsqu'on cherche à obtenir le module, on fait appel à une méthode qui calcule ce module. Lorsqu'on cherche à modifier ce module, on fait appelle 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`.

```
# -*- coding: cp1252 -*-
import math
```

```
class nombre_complexe(object):
    def __init__ (self, a = 0, b= 0):
        self.a = a
        self.b = b

    def __str__ (self) :
        if (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)

    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, " argument : ", c.arg
                      # affiche module =  1.0  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, " argument : ", c.arg
                      # affiche module =  1.0  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 4.3.5: 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 paragraphe 4.6).

4.4 Copie d'instance

4.4.1 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` affiche la même chose. En réalité, `nb` et `nb2` désignent le même objet.

```
class exemple_classe:
    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 ()
nb2 = nb

print nb.rnd        # affiche 42
print nb2.rnd       # affiche 42

nb2.rnd = 0

print nb.rnd        # affiche 0, si nb et nb2 étaient des objets différents,
                    # cette ligne devrait afficher 42
print nb2.rnd       # affiche 0
```

Pour créer une copie de l'instance `nb`, il faut le dire explicitement en utilisant la fonction `copy` du module `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:
    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 ()

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 nb.rnd    # affiche 42
print nb2.rnd   # affiche 0
```

Remarque 4.4.1: Suppression de plusieurs variables associées à la même instance
Le symbol 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
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.

```
1   class CreationDestruction (object) :
2
3       def __init__ (self) :
4           print "constructeur"
5
6       def __new__ (self) :
7           print "__new__"
8           return object.__new__ (self)
9
10      def __del__ (self) :
```

```
11          print "__del__"
12
13  m = CreationDestruction ()
14  m2 = m
15  del m
16  del m2
```

La sortie de ce programme est la suivante :

```
__new__
constructeur
__del__
```

Le destructeur est appelé autant de fois que le constructeur.

4.4.2 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` est copié comme si l'instruction `nb2.inclus = nb.inclus` était exécutée. On se retrouve donc avec deux noms qui désignent encore le même objet.

```
class classe_incluse:
    attr = 3

class exemple_classe:
    inclus = classe_incluse ()
    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
print nb2.inclus.attr  # affiche 0
```

Pour effectivement copier les attributs dont le type est une classe, il faut utiliser l'opérateur `__copy__` et ainsi écrire le code associée à la copie des attributs de la classe.

```
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, cet 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_class` 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:
    attr = 3

class exemple_classe:
    inclus = classe_incluse ()
    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 à tous niveaux,
                       # 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
```

Remarque 4.4.2: 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 [4,5,6]
l = fonction_liste ()  # la liste [4,5,6] n'est pas recopiée,
                       # elle reçoit simplement le nom l
```

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 [4,5,6]
fonction_liste ()      # la liste [4,5,6] n'est pas recopiée,
                       # elle n'est pas non plus attribué à une variable,
                       # elle est alors détruite automatiquement par le langage Python
```

4.4.3 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]]
```

4.4.4 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__`.

```
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.

```
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, cet 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 4.4.2 et modifie la classe `classe_incluse` pour distinguer copie et copie profonde.

```
import copy
```

```
class classe_incluse:
    attr = 3

class exemple_classe:
    inclus = classe_incluse ()
    rnd = 42
    def __copy__ (self):
        copie = exemple_classe ()
        copie.rnd = self.rnd
        return copie
    def __deepcopy__ (self,memo):
        copie = copy.copy (self)
        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.__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
```

4.5 Attributs non liés

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 contre partie, 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 4.3.4), chaque instance n'a pas besoin d'avoir une liste d'attributs variable. Le langage *Python* offre la possibilité de figer cette liste.

```
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 4.6). 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) :

```
Traceback (most recent call last):
  File "cours4.py", line 15, in ?
    a.j = 6
AttributeError: 'point_espace' object has no attribute 'j'
```

Il en est de même lorsque l'attribut est ajouté à l'intérieur d'une méthode. En outre, l'attribut `__dict__` n'existe pas, par conséquent, l'expression `a.__dict__` génère la même exception. Toutefois, comme le programme suivant ne provoque pas d'erreur, il semble possible d'ajouter un attribut `j` à la classe s'il est déclaré hors d'une méthode.

```
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)

    j = 6

a = point_espace (1,-2,3)
print a
```

En fait, l'attribut `j` est un attribut statique, il est donc commun à toutes les instances de la classe `point_espace`.

4.6 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.

4.6.1 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

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.

```
import random

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.

```
import random

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[1]. On choisit alors de représenter une pièce normale par une classe.

```
import random

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.

```
import random

class piece_normale :
```

[1] 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 $\texttt{piece}_\texttt{t}\texttt{res}_\texttt{t}\texttt{ruquee}$ avait un paramètre caché. Cette solution n'est pas conseillée.

```
    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 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`.

```
import random

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`.

```
import random

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. Leur intérêt réside principalement dans la redéfinition de méthodes déjà définies chez l'ancêtre, ce procédé s'appelle la surcharge. On peut ainsi créer des versions différentes d'un même type d'objet en surchargeant certains méthodes et en en conservant d'autres.

Définition 4.6.1 : 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.

Définition 4.6.2 : 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. A moins qu'il ne soit explicitement fait appel à une méthode d'une classe donnée, c'est toujours la méthode surchargée qui est exécutée.

4.6.2 Syntaxe

L'héritage obéit à la syntaxe suivante.

```
class nom_classe (nom_ancetre) :
    # corps de la classe
    # ...
```

`nom_classe` désigne le nom d'une classe, `nom_ancetre` est le nom de la classe dont `nom_classe` hérite ou dérive. `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)
```

La rubrique `Methodresolutionorder` 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__` (voir paragraphe 4.1.4) d'une classe contient le (ou les ancêtres, voir paragraphe 4.6.4). 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
```

Dans les exemples précédents, `piece_normale` ne dérive qu'aucun 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 4.3.4 sur les propriétés ou encore paragraphe 4.5 sur les attributs non liés).

Le paragraphe précédent a montré qu'il était parfois utile d'appeler dans une méthode surchargée ou non d'une classe une méthode appartenant à l'ancêtre direct de cette classe ou à un de ses ancêtres. La syntaxe est la suivante.

```
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
        nom_autre_methode(self, ...)
            # 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.

Remarque 4.6.3: surcharge d'attributs
Contrairement aux mérthodes, la surcharge d'attributs n'est pas possible. Si un ancêtre possède un attribut d'identifieur `a`, les classes dérivées le possède aussi et ne peuvent en déclarer un autre du même nom. Cela tient au fait que quelques soit la méthode utilisée, celle de l'ancêtre et celle d'une classes dérivées, 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
```

4.6.3 Sens de l'héritage

Il n'est pas toujours évident de concevoir le sens d'un héritage. En mathématiques, 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. Néanmoins, 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 restes 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 contre partie, il n'y a pas d'information redondantes.

Ce problème d'héritage ne se pose pas à chaque fois. Dans l'exemple du paragraphe 4.6.1 autour des pièces truquées, il n'y a pas d'ambiguïtés sur le sens de l'héritage. Le programme perdrait en clarté si une pièce normale devait hériter d'une pièce truquée. Cette ambiguïté n'existe pas car les propriétés communes à toutes les classes héritant de la classe **piece_normale** sont connues, elles sont imposées par le problème à résoudre.

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.

4.6.4 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 ancêtre 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 **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 possède 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
```

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
```

4.6.5 Fonction `isubclass`

La fonction **isubclass** permet de savoir si une classe hérite d'une autre.

> **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.

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
```

4.7 Compilation de classes

La compilation de classe fonctionne de manière similaire à celle de la compilation de fonctions (voir paragraphe 3.4.10.2,page 64). 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 4.6.3. Elles sont incluses dans une chaîne de caractères, compilées puis incluses au programme (fonction **exec**).

```
# -*- coding: cp1252 -*-
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: cp1252 -*-
"""erreur de compilation inclusesdans 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 `variables` qui est le second paramètre envoyé à la fonction `compile`. Le numéro de ligne fait référence à la troisième ligne de la chaîne de caractère `s` et non la troisième ligne du programme.

4.8 Exemple plus complet

```
1  # -*- coding: cp1252 -*-
2  class fromage :
3
4      def __init__ (self, p,c,o) :
5          self.poids = p
6          self.couleur = c
7          self.odeur = o
8
9      def decouper (self,nb) :
10         l = []
11         for i in range (0,nb) :
12             f = fromage (self.poids/nb, \
13                             self.couleur, self.odeur)
14             l.append (f)
15         return l
16
17     def __str__ (self) :
18         s = "poids : " + str (self.poids)
19         s += " couleur : " + str (self.couleur)
20         s += " odeur : " + str (self.odeur)
21         return s
22
23     def __add__ (self,f) :
24         print "ajout fromage"
25         poids = self.poids + f.poids
26         couleur = [0,0,0]
27         for i in range (0,3) :
28             couleur [i] = (self.couleur [i] * self.poids \
29                         + f.couleur [i] * f.poids) / poids
30         odeur = (self.odeur * self.poids + \
31                  f.odeur * f.poids) / poids
32         couleur = ( couleur [0], couleur [1], couleur [2])
33         return fromage (poids, couleur, odeur)
34
35 class gruyere (fromage) :
36     def __init__ (self,p) :
37         fromage.__init__ (self, p, c = (150,150,0), o = 0.1)
38
```

```
39      def __str__ (self) :
40          s = fromage.__str__(self)
41          s = "gruyère, " + s
42          return s
43
44      def __add__ (self,f) :
45          print "ajout gruyère"
46          if not isinstance (f, gruyere) :
47              return fromage.__add__ (self, f)
48          else :
49              r = gruyere (self.poids + f.poids)
50              return r
51
52
53
54  fr = fromage (5.0, (255,0,0), 0.5)
55  fr2 = fromage (10.0, (0,255,0), 1)
56  fr3 = fr + fr2
57  print fr
58  print fr2
59  print fr3
60  print "----------------------"
61  g = gruyere (3.0)
62  g2 = gruyere (7.0)
63  g3 = g + g2
64  print g
65  print g2
66  print g3
67  print "----------------------"
68  print fr2 + g
```

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 5.1.1 : 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 est déclenche une erreur.

```
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 mot-clé `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.

```
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.

```
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 5.1.2: 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, le programme suivant n'affiche que le message *le programme a déclenché une erreur*. Dès la première erreur qui correspond au calcul d'une puissance non entière d'un nombre négatif, l'exécution du programme 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 déclenchées 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 appelé **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 attaper. 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 suit le schéma quit suit.

```
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
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.

Le langage *Python* propose une liste d'exceptions standards (voir table 5.1). 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 5.3). Lorsqu'une fonction ou une méthode déclenche une exception non standard, généralement, le commentaire qui lui est associée 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"
```

`AttributeError`	Une référence à un attribut 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.
`IndentationError`	L'interpréteur ne peut interpréter une partie du programme à cause d'un problème d'indentation.
`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 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`	Les chaînes de caractères usuelles ne permettent pas de représenter l'ensemble des langues possibles, d'autres chaînes, de type unicode sont utilisées. Il faut parfois convertir ces chaînes d'un format à l'autre, ce qui n'est pas toujours possible.
`ValueError`	Cette exception 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.

Tab. 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.

```
    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.

> `raise exception_type, message`
>
> Cette instruction lance l'exception `exception_type` associé 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 suivante.

```
try :
    # ... instructions à protéger
except exception_type, exc :
    # ... que faire en cas d'erreur de type exception_type
```

L'erreur de type `exception_type` est attrapée par l'instruction `except` et est associée à l'identificateur `exc`. L'instruction `str(exc)` retourne le message associé à l'exception.

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 :` 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 5.3).

5.1.5 Instructions `try`, `except` imbriquées

Comme pour les boucles, il est possible d'imbriquer les portions de code protégé 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 appel la fonction `inverse` qui intercepte les exceptions de type `ZeroDivisionError` pour retourner une grande valeur lorsque $\texttt{x} = 0$. La seconde exception générée survient lors de l'appel à la fonction `inverses` qui déclenche l'exception `NameError`, elle aussi intercéptée.

```
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 --> ZeroDivisionError
    print inverses (0)  # fonction inexistante --> exception NameError
except NameError:
    print "appel à une fonction non définie"
```

5.2 Exemple d'utilisation des exceptions : les itérateurs

Les itérateurs sont des outils qui permettent de parcourir des objets qui sont des ensembles, comme une liste, un dictionnaire. Ils sont fonctionnent toujours de la même manière. Dans l'exemple qui suit (déjà présenté au paragraphe 4.2.3), on définit une classe contenant trois coordonnées, ainsi qu'un itérateur permettant de parcourir ces trois coordonnées. Arrivé à la troisième, l'exception `StopIteration` est déclenchée. Cette exception indique à une boucle `for` de s'arrêter. Une boucle `while` fait apparaître explicitement le rôle de l'exception.

```
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)

    # cette classe définit un itérateur pour point_espace
    class class_iter:

        # initialisation, self._ins permet de savoir quelle
        # instance de point_espace on explore, self._n
        # mémorise l'indice de l'élément exploré
        def __init__ (self,ins):
            self._n   = 0
            self._ins = ins

        # retourne l'élément d'indice self._n
        # et passe à l'élément suivant
        def next (self):
            if self._n <= 2:
                v = self._ins [self._n]
                self._n += 1
                return v
            else :
                # si cet élément n'existe pas, lève une exception
                raise StopIteration

    # opérateur de la classe point_espace, retourne un itérateur
    # permettant de l'explorer
    def __iter__(self):
        return point_espace.class_iter (self)

    def __getitem__ (self, n) :
        if n == 0 : return self._x
        elif n == 1 : return self._y
        elif n == 2 : return self._z
        else : raise IndexError

a = point_espace (1,-2,3)
for x in a:
    print x      # affiche successivement 1,-2,3

# ----- autre solution ------
a = point_espace (1,-2,3)
it = iter (a)
while True:
    try :
        print it.next ()  # affiche successivement 1,-2,3
```

```
    except StopIteration :
        break
```

Cet exemple montre seulement que les exceptions n'interviennent pas seulement lors d'erreurs mais font parfois partie intégrante d'un algorithme. C'est souvent le cas lorsqu'une fonction retourne un résultat dans certains cas seulement. Dans l'exemple qui suit, la fonction `racine_carree` retourne un couple de résultat, `True` ou `False` pour savoir si le calcul est possible, suivi du résultat qui n'a un sens que si `True` est retourné en première valeur.

```
import math
def racine_carree(x) :
    if x < 0 : return False, 0
    else : return True, math.sqrt (x)

print racine_carree (-1)  # (False, 0)
print racine_carree (1)   # (True, 1.0)
```

Plutôt que de compliquer le programme avec deux résultats, 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.

```
import math
def racine_carree(x) :
    if x < 0 : raise ValueError, "valeur négative"
    return math.sqrt (x)

print racine_carree (-1)  # déclenche une exception
print racine_carree (1)
```

5.3 Définir ses propres exceptions

5.3.1 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
```

5.3.2 Personnalisation d'une classe d'exception

Il est parfois utile qu'une exception contiennent 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 tuple 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, lancée 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 AucunChiffre.__doc__, " : ", exc
    print "fonction : ", exc.f
```

Etant donné que le programme déclenche une exception dans la section de code protégée, les deux dernièrs affichages sont les seuls exécutés correctement. Ils produisent les quatre lignes qui suivent.

```
chaîne de caractères contenant
    aussi autre chose que des chiffres  :  exception AucunChiffre,
lancée depuis la fonction conversion avec le paramètre 123a
fonction :  conversion
```

Chapitre 6

Modules

Il est préférable souvent 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 6.1.1 : 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 est égale au nom du fichier sans son extension.

Fichier : `module_exemple.py`

```
1   # -*- coding: cp1252 -*-
2   """exemple de module, aide associée"""
3
4   exemple_variable = 3
5
6   def exemple_fonction () :
7       """exemple de fonction"""
8       return 0
9
10  class exemple_classe :
11      """exemple de classe"""
```

```
12      def __str__ (self) :
13          return "exemple_classe"
```

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. L'exemple qui suit importe le module défini ci-dessus.

Fichier : exemple.py

```
1  import module_exemple
2
3  c = module_exemple.exemple_classe ()
4  print c
5  print module_exemple.exemple_fonction ()
6  help (module_exemple)
```

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 le identificateur `alias` comme le montre l'exemple suivant.

Fichier : exemple2.py

```
1  import module_exemple as alias
2
3  c = alias.exemple_classe ()
4  print c
5  print alias.exemple_fonction ()
6  help (alias)
```

Une dernière 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

```
1  from module_exemple import *
2
3  c = exemple_classe ()
4  print c
5  print exemple_fonction ()
```

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"
```

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")
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.

6.2 Modules disponibles

Le petit exemple suivant permet d'obtenir la liste des modules qu'il est possible d'importer.

```
import sys
import pprint
pprint.pprint (sys.modules)
```

Voici le début de la liste.

```
{'ConfigParser': <module 'ConfigParser' from 'C:\Python23\lib\ConfigParser.pyc'>,
 'FixTk': <module 'FixTk' from 'C:\Python23\lib\lib-tk\FixTk.pyc'>,
 'Queue': <module 'Queue' from 'C:\Python23\lib\Queue.pyc'>,
 'SimpleDialog': <module 'SimpleDialog' from 'C:\Python23\lib\lib-tk\SimpleDialog.pyc'>,
 'SocketServer': <module 'SocketServer' from 'C:\Python23\lib\SocketServer.pyc'>,
 'StringIO': <module 'StringIO' from 'C:\Python23\lib\StringIO.pyc'>,
 'Tkconstants': <module 'Tkconstants' from 'C:\Python23\lib\lib-tk\Tkconstants.pyc'>,
 'Tkinter': <module 'Tkinter' from 'C:\Python23\lib\lib-tk\Tkinter.pyc'>,
 'UserDict': <module 'UserDict' from 'C:\Python23\lib\UserDict.pyc'>,
 ... # liste tronquée
```

Mais il est plus facile de vérifier la présence d'un module grâce aux quelques lignes suivantes.

```
import sys
print "math" in sys.modules  # retourne True normalement puisque le module
                             # math est installé par défaut
```

La liste de tous les modules disponibles 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 qui suit regroupe certains des modules les plus utilisés.

`re`	module gérant les expressions régulières
`os`	module gérant les fichiers
`os.path`	module gérant les noms de fichiers
`time`	module gérant tout ce qui est relatif au temps
`math`	fonctions mathématiques
`cmath`	fonctions mathématiques complexes
`Numeric`	module gérant les tableaux
`Tkinter`	module gérant les interfaces graphiques

Tab. 6.1: Liste de modules souvent utilisés.

6.3 Modules externes

De nombreux modules sont accessibles depuis le site officiel du langage *Python*. En voici quelques-uns, regroupés dans la table 6.3.

`SciPy`	Le module SciPy 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 courbe 2D, 3D.
`PIL`	Le module PIL (Python Image Library) permet d'utiliser les images, quelque soit leur format. Elle propose également quelques traitements d'images simples, rotation, zoom, histogrammes.
`Numarray`	Le module Numarray manipule efficacement les tableaux de grandes dimensions, intègre le calcul matriciel, la transformée de Fourier.
`PyGame`	Le module PyGame 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. Ce module requiert le module Numarray pour la manipulation des tableaux.
`Biggles`	Le module Biggles permet de dessiner des courbes (2D,3D) et de convertir ses graphiques sous forme d'images de tout format. Ce module nécessite l'installation de fichiers supplémentaires (libpng, zlib, plotutils) accessibles depuis l'adresse http ://sourceforge.net/projects/gnuwin32/.

`Psyco`	Le module Psyco permet de multiplier la vitesse d'exécution d'un programme par trois ou quatre. Le problème d'optimisation de la tournée d'un voyageur de commerce permet de constater le gain non négligeable en vitesse. Il suffit d'ajouter au programme les instructions suivantes :
	
	`import psyco`
	`psyco.full ()`

Tab. 6.2: Liste de modules externes souvent utilisés.

Certains modules externes comme `SciPy` peuvent être installés à l'aide d'un fichier exécutable. 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 apparaissent compressés dans un fichier. Une fois décompressés, ils incluent un fichier `setup.py` ou un fichier `__init__.py`. Le langage *Python* fournit une procédure standard. Pour un ordinateur fonctionnant sous Windows, si le moteur *Python* est installée dans le répertoire `c:\python`, il suffit d'écrire des quelques lignes dans une fenêtre de commande ouverte dans le répertoire où a été décompressé le fichier contenant le module à installer.

```
set PATH=%PATH%;C:\python
python.exe setup.py install
```

La première ligne permet d'ajouter le répertoire `c:\python` à la liste des répertoires dans lesquels le système Windows ira chercher le programme `python.exe`. Ce procédé d'installation est celui utilisé par exemple par le module `py2html` qui permet de transformer un programme écrit en *Python* en une page HTML.

6.4 Python et les autres langages

6.4.1 Langage C

Inclure des programmes écrits en langage C est prévu par le langage *Python* ou plutŹt par le logiciel qui interprète les programmes *Python*. Mais il est nécessaire que ces parties écrites en langage C ne soit pas trop complexe.

Dans le cas où le langage *Python* n'est en fait que la partie émergée et ne sert que d'interface à un programme plus conséquent écrit en langage C ou C++, il est préférable d'utiliser des outils comme la librairie *SWIG*, ou encore la librairie *boost/python*.

Chapitre 7

Fichiers

Lorsqu'un programme termine son exécution, toutes les informations stockées dans des variables sont perdues. Le seul moyen de les conserver est de les enregistrer dans un fichier sur disque dur. Dans ce fichier, ces informations peuvent apparaître sous un format texte qui est lisible par n'importe quelle personne ou sous un format binaire souvent lorsque les informations sont trop nombreuses. Le format binaire est illisible excepté pour celui qui l'a conçu. C'est le cas par exemple d'un fichier Word, sans Word, il est incompréhensible.

7.1 Format texte

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 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

Le fichier dans lequel seront écrites les informations est créé s'il n'existe ou nettoyé s'il existe déjà. La fermeture permet à d'autres programme 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
```

Certains codes sont fort utiles lors de l'écriture de fichiers texte :

`\n` passage à la ligne
`\t` insertion d'une tabulation, indique un passage à la colonne suivante dans le logiciel Excel

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 du précédent contenu de ce fichier lors de son ouverture pour écriture. Le second cas consiste à ajouter toutes nouvelles informations déjà présentes lors de l'ouverture du fichier. Le premier cas est traité au paragraphe précédent, le second est presque identique au suivant hormis la première ligne qui changent :

```
f = open ("nom-fichier", "a")    # ouverture en mode ajout (la fin de la ligne change)
...
```

Pour comprendre la différence entre ces deux modes d'ouverture, voici un premier programme qui n'utilise pas le mode ajout :

```
f = open ("essai.txt", "w")
f.write (" première fois ")
f.close ()
f = open ("essai.txt", "w")
f.write (" seconde fois ")
f.close ()
```

Ce programme crée un fichier "essai.txt" qui ne contient que les informations écrites lors de la seconde phase d'écriture, soit `seconde fois`. Le suivant utilise le mode ajout lors de la seconde ouverture.

```
f = open ("essai.txt", "w")
f.write (" premiere fois ")
f.close ()
f = open ("essai.txt", "a")
f.write (" seconde fois ")
f.close ()
```

Au final, 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`.

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. Dans un éditeur de texte, un fichier texte s'étend sur plusieurs lignes et sa lecture en *Python* s'effectue de même, ligne par ligne[1].

[1] Alors que l'écriture ne suit pas forcément un découpage en ligne.

```
f = open ("essai.txt", "r")  # ouverture du fichier en mode lecture
s = f.readline ()                 # lecture de la première ligne
while not s.empty ()         # tant que cette ligne n'est pas vide
    print s                            # on l'affiche
    s = f.readline ()             # puis on lit la ligne suivante
f.close ()                            # fermeture du fichier
```

Le programme précédent lit des lignes dans un fichier texte tant qu'il n'est pas arrivé à la fin de ce fichier. La méthode **readline** des fichiers retourne la prochaine ligne du fichier ou une chaîne de caractères vide si elle n'existe pas, c'est-à-dire si la lecture du fichier est arrivée à son terme. 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 7.1.1: code de fin de ligne
Lors 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

l_net = []                       # contiendra la liste nettoyée des lignes du fichier
for s in l :
    s2 = s.replace ("\n", "")        # on supprime le code de fin de ligne \n
    s2 = s2.replace ("\r", "")  # on supprime le code de fin de ligne \r (Windows uniquement)
    l_net.append (s2)               # on ajoute le résultat à la liste nettoyée
```

Remarque 7.1.2: fichier formaté
Les fichiers textes ont de nombreux formats, on peut citer HTML ou XML qui sont des formats à balises. Leur lecture utilise des modules comme **HTMLParser** ou **xml.sax** dont la description sort du cadre de ce document. Un autre format est souvent utilisé avec le logiciel *Excel* ou tout autre tableur. Lorsqu'on enregistre une feuille de calcul sous format texte, le fichier obtenu est organisé en colonne : sur une même ligne, les informations sont disposées en colonne délimitées par un séparateur qui est souvent une tabulation (\t) ou un point virgule.

```
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.

```
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 :
    s2 = s.replace ("\n", "")        # on supprime le code de fin de ligne \n
    s2 = s2.replace ("\r", "")  # on supprime le code de fin de ligne \r (Windows uniquement)
    case = s2.split (";")
    if len (case) >= 3 :
        print case [1], " ", case [0], " a écrit ", case [2]
```

7.2 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, même si elle permet de relire les informations écrites grâce à n'importe quel éditeur de texte, est parfois jugée coûteuse en temps de traitement. C'est pourquoi, il est préférable pour une grande masse d'informations à stocker dans un fichier 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 selon 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`).

7.2.1 Ecriture dans un fichier binaire

ouverture un mode binaire "wb"

```
def lire_image (fim, nl, nc) :
    """lit une image dans le fichier fim, cette image a nl lignes et nc colonnes,
    retourne l'image sous la forme d'un t-uple"""
    nb  = nl * nc
    str = fim.read (nb)
    t   = struct.unpack ("B" * nb, str)
    return t

def lire_label (fla) :
    """lit un label (un chiffre) dans le fichier fla, retourne un entier"""
    str = fla.read (1)
    t   = struct.unpack ("B", str)
    return t [0]
```

7.3 Fichiers zip

Les fichiers zip sont très répandus de nos jours et constitue 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 zlib.

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. 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 du programme reste la même.

8.1 Introduction

Prenons par exemple, le cas d'un programme qui demande à l'utilisateur de chercher un nom de fichier. On part donc d'un répertoire initial. A chaque étape, il doit être possible de :

1. Aller dans le répertoire précédent dans l'arborescence.
2. Entrer dans un sous-répertoire.
3. Annuler et quitter le programme.
4. Sélectionner un fichier.

Un programme sans interface graphique va inclure une boucle principale. A chaque itération, le programme demande laquelle de ses actions l'utilisateur désire exécuter. La prémière et la troisième ne nécessitent pas d'autres démarche. Pour la seconde et la dernière, l'utilisateur doit entrer soit un nom de sous-répertoire, soit un nom de fichier. Le partie principale du programme ressemble à l'extrait qui suit. La variable `chemin` est le résultat cherché. Le paragraphe 8.7.1 (page 155) inclut le programme complet.

```
while True :
    print "chemin actuel : ", chemin
    print "liste des fichiers et répertoire inclus"
```

```
affiche_list ()
print "quelle action : Précédent, Entre, Annuler, Ok (entrer la lettre majuscule)\n"
str = raw_input ("")
if str == "P" :
    precedent ()
elif str == "E" :
    sel = raw_input ("entrer un nom de répertoire\n")
    suivant (sel)
elif str == "A" :
    annuler ()
    break     # on sort de la boucle
elif str == "O" :
    print "Entrer un nom de fichier"
    str = raw_input ("\n")
    chemin += "\\" + str
    accepter ()
    break     # on sort de la boucle
else :
    print "choix incompréhensible"
```

Le même programme muni d'une interface graphique fonctionne différemment. Après avoir affiché la fenêtre (voir figure 8.1), le programme ne demande pas de manière explicite à l'utilisateur ce qu'il veut faire. Il attend que l'utilisateur choisisse un bouton de la fenêtre pour savoir quoi faire. Pour cela, le programmeur va devoir associer une fonction à chaque bouton qu'il insère dans sa fenêtre. Cette fonction ne sera exécutée que si le bouton est pressé. D'une certaine manière, on peut considérer qu'il y a quatre points d'entrée du programme, quatre fonctions associées chacune à un bouton.

Fig. 8.1: Exemple d'une fenêtre graphique. Celle-ci permet de sélectionner un fichier. La première boîte contient une liste de fichiers et de répertoires (ceux-ci sont précédés du signe +). Parmi les quatre boutons, le premier intitulé "Précédent" permet de descendre d'un cran dans l'arborescence des fichiers. Le second "Entre" permet d'entrer dans un répertoire. Le troisième "Annuler" ferme la fenêtre sans sélectionner de fichier. Le dernier "Ok" ferme la fenêtre et sélecitonne un fichier.

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 une fonction qui sera exécutée 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. Le paragraphe qui suit décrit tous les objets que propose le module `Tkinter`. Le paragraphe suivant présente la manière de les disposer dans une fenêtre. Le paragraphe d'après décrit les événements et le moyen de les relier à des fonctions du programme. Le dernier paragraphe 8.7 contient le code des deux programmes brièvement présentés ici, le premier sans interface graphique, le second avec.

8.2 Les objets

Les interfaces graphiques sont composées d'*objets* ou *widgets*[1]. Chaque objet peut recevoir un contenu sous formes de chaînes de caractères, chaque objet a une apparence, et chaque objet peut recevoir des événements. 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 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 mais qui toutefois ne s'affichera pas sans deux 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
root = Tkinter.Tk ()
# ...
zone_texte = Tkinter.Label (text = "zone de texte")
# ...
zone_texte.pack ()
root.mainloop ()
```

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")
```

[1] 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.

```
# ...
# pour changer de texte
zone_texte.config (text = "second texte")
```

Fig. 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)
```

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")
```

Fig. 8.3: Exemple de boutons, non pressé, pressé, grisé. Le bouton grisé ne peut être pressée.

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)
```

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 (voir figure 8.4).

```
b = Tkinter Button ()
im = Tkinter.PhotoImage (file = "bette_davis.gif")
b.config (image = im)
```

Fig. 8.4: Exemple de bouton contenant une image.

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 créer une zone de saisie, 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 est inséré 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 ()))
```

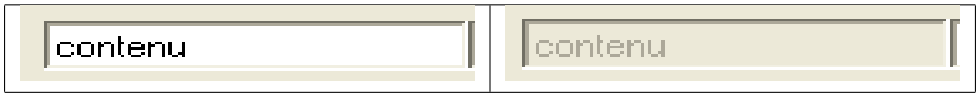

Fig. 8.5: Exemple de zone de saisie, normale et grisée, la zone grisée ne peut être modifiée.

La figure 8.5 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 l'instruction suivante :

```
saisie.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
saisie.config (state = Tkinter.NORMAL)
```

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 est inséré 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 insérer 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)
```

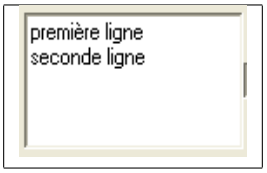

Fig. 8.6: Exemple de zone de saisie à plusieurs lignes. Dans la version 2.3 de *Python*, la zone grisée a le même aspect que la zone non grisée à ceci près qu'il est toujours impossible de la modifier.

La figure 8.6 montre une zone de saisie à plusieurs lignes. Pour griser cette zone, il suffit d'utiliser l'instruction suivante :

```
saisie.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
saisie.config (state = Tkinter.NORMAL)
```

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)
```

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 la case, il faut utiliser l'instruction suivante :

```
case.select ()
```

Pour décocher la case, il faut utiliser l'instruction suivante :

```
case.deselect ()
```

Il est possible d'associer du texte à l'objet case à cocher :

```
case.config (text = "case à cocher")
```

Fig. 8.7: Exemple de case à cocher, non cochée, cochée, grisée. Lorsqu'elle est grisée, son état ne peut être modifiée. La deuxième image montre une case à cocher associée à un texte.

La figure 8.7 montre trois cases. La troisième est grisée par rapport au premier. Les cases grisées ne peuvent pas être cochées. Pour obtenir cet état, il suffit d'utiliser l'instruction suivante :

```
case.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
case.config (state = Tkinter.NORMAL)
```

8.2.6 Case ronde

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 pour récupérer le numéro du bouton pressé
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 à une valeur de `v`. Si `v` == `10`, seul le premier bouton sera sélectionné. Si `v` == `20`, seul le second bouton le sera. Si deux valeurs sont identiques pour deux boutons, il seront cochés et décochés en même temps. Et pour savoir quels boutons est coché ou non, il suffit d'exécuter l'instruction :

```
v.get ()  #  retourne le numéro du bouton radio coché
          #  et associé à la variable v
```

Pour cocher un des boutons, il faut utiliser l'instruction suivante :

```
v.set (numero)  # numéro du bouton à cocher
                # pour cet exemple, 10, 20 ou 30
```

Il est possible d'associer du texte à un bouton (vour figure 8.8) :

```
case1.config (text = "premier bouton")
case2.config (text = "second bouton")
case3.config (text = "troisième bouton")
```

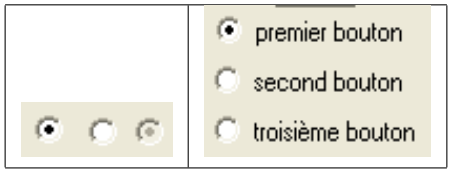

Fig. 8.8: Exemple de case ronde ou bouton radio, non cochée, cochée, grisée. Lorsqu'elle est grisée, son état ne peut être modifiée. La seconde image présente un groupe de bouton radio. Un seul peut être sélectionné à la fois.

La figure 8.8 montre trois cases. La troisième est grisée par rapport à la première. Les boutons grisés ne peuvent pas être pressés. Pour obtenir cet état, il suffit d'utiliser l'instruction suivante :

```
case1.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
case1.config (state = Tkinter.NORMAL)
```

Si un des boutons est grisé parmi les trois, le choix du bouton à cocher s'effectue parmi les deux restants ou aucun si le bouton 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")
```

Réciproquement, 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'obtenir 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 ()
```

Fig. 8.9: Exemple de liste. La second liste est grisée et ne peut être modifiée.

La figure 8.9 montre deux listes. La seconde est grisée par rapport à la première. Elle ne peut être modifiée. Pour obtenir cet état, il suffit d'utiliser l'instruction suivante :

```
li.config (state = Tkinter.DISABLED)
```

Et pour revenir à un état normal :

```
li.config (state = Tkinter.NORMAL)
```

Il est possible d'ajoindre une barre de défilement verticale. il faut pour cela inclure l'object dans une sous-fenêtre `Frame` 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.

Remarque 8.2.1: plusieurs Listbox
Lorsqu'on insère plusieurs objets `Listbox` dans une seule fenêtre, ces objets partagent la même sélection. Autrement dit, lorsque on clique sur un élément de la seconde `Listbox`, l'élément sélectionné dans la première de 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)
```

8.2.8 Canevas

Pour dessiner, il faut utiliser un objet cenvas, correspondant à la classe `Canvas` du module `Tkinter`. Pour la créer, il suffit d'écrire la ligne suivante :

```
ca  = Tkinter.Canvas ()
```

Pour modifier les dimensions de la zone de saisie à plusieurs lignes, on utilise l'instruction suivante :

```
# modifie les dimensions du canevas
# width <--> largeur
# height <--> hauteur en lignes
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 8.10 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
li.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
li.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
li.create_rectangle (300,100,60,120, fill = "yellow", width = 2)
# écrit du texte de couleur noire au point 80,80 et avec la police arial
li.create_text (80,80, text = "écrire", fill = "black", font = "arial")
```

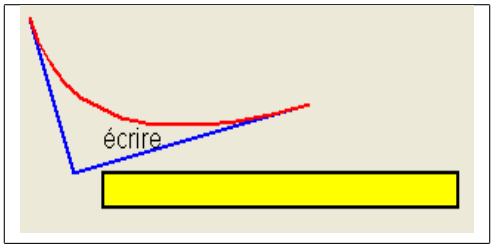

Fig. 8.10: Exemple de canevas.

8.2.9 Méthodes communes

Nous avons vu que tous les objets présentés dans ce paragraphe possède une méthode `config` qui permet de définir l'état du widget (grisé ou 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[2] 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`[3] 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,
```

[2] Par exemple pour la classe `Label`, cette aide est affichée grâce à l'instruction `help(Tkinter.Label.config)`.
[3] Affichée avec l'instruction `help(Tkinter.Label.__init__)`.

```
        highlightbackground, highlightcolor,
        highlightthickness, image, justify,
        padx, pady, relief, takefocus, text,
        textvariable, underline, wraplength

    WIDGET-SPECIFIC OPTIONS

        height, state, width
```

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*[4] 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`. Seules les deux premières seront présentées. 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 placées avec la même méthode auquel il y aura un conflit.

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 8.11.

```
l = Tkinter.Label (text = "première ligne")
l.pack ()
s = Tkinter.Entry ()
s.pack ()
e = Tkinter.Label (text = "seconde ligne")
e.pack ()
```

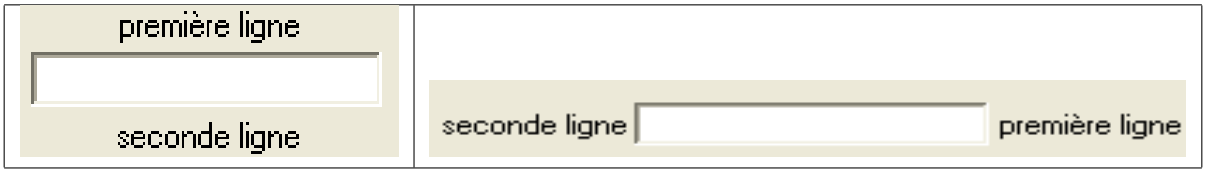

Fig. 8.11: Les objets sont empilés à l'aide de la méthode `pack` les uns en dessous des autres pour la première image, les uns à droite des autres pour la seconde image.

On peut aussi les empiler les uns à droite des autres grâce à l'option `side`.

[4] voir paragraphe 8.4.3

```
l = Tkinter.Label (text = "première ligne")
l.pack (side = Tkinter.RIGHT)
s = Tkinter.Entry ()
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 autre valeurs, `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 `pack` empile les objets les uns à la suite des autres. 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 8.12.

```
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)
```

Fig. 8.12: Les objets sont placés dans une grille à l'aide de la méthode `grid` les uns en dessous des autres pour la première image, les uns à droite des autres pour la seconde image.

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
```

8.3.2 Sous-fenêtre

Les deux 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 8.13 montre deux objets regroupés dans un rectangle avec à sa gauche une zone de texte. Les boîtes sont des instances de la classe `Frame` sont des objets comme les autres excepté le fait qu'une boîte contient d'autres objets. 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 8.13 :

```
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
```

8.4 Evénements

8.4.1 Fenêtre principale

Tous les exemples des paragraphes précédents décrivent les différentes objets disponibles et comment les disposer dans une fenêtre. Pour afficher cette fenêtre, il suffit d'ajouter au programme les deux lignes suivantes :

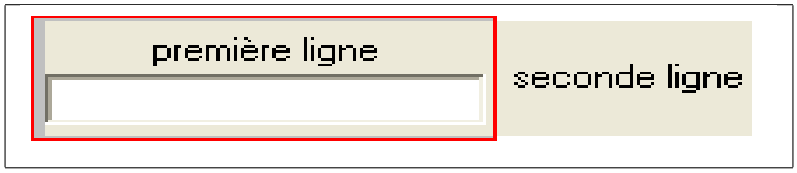

Fig. 8.13: Les deux premiers objets - une zone de texte au-dessus d'une zone de saisie - sont regroupées dans une boîte - rectangle rouge, invisible à l'écran. A droite et centrée, une dernière zone de texte. Cet alignement ne serait pas possible sans regrouper les deux premiers objets dans un rectangle.

```
root = Tkinter.Tk ()
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 message*. 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 interessé, cet objet prend l'événement et le traite. La fonction `mainloop` attend l'événement suivant. Les paragraphes suivant permettent d'associer un ou plusieurs événements à un objet.

Remarque 8.4.1: 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 Lancer une fonction lorsqu'un bouton est pressé

Le code suivant permet de créer un bouton dont l'identificateur est `b`. Il a pour intitulé `fonctionchange_legende`. On définit ensuite une fonction `change_legende` qui change la légende de ce bouton. La ligne 11 est le seul changement par rapport aux exemples précédents : elle 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.

```
1   # -*- coding: cp1252 -*-
2   import Tkinter
3   root = Tkinter.Tk ()
4   b = Tkinter.Button (text = "fonction change_legende")
5   b.pack ()
6
7   def change_legende () :
8       global b
9       b.config (text = "nouvelle légende")
10
11  b.config (command = change_legende)
12
13  root.mainloop ()
```

Lorsque le bouton `b` est pressé, on vérifie qu'il change bien de légende (voir figure 8.14).

Fig. 8.14: Ces deux fenêtres sont celles qui apparaissent lors de l'exécution du programme du paragraphe 8.4.2. Le bouton change de légende la première fois qu'il est pressé.

8.4.3 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 le focus d'un objet à l'autre. La figure 8.15 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 ()
```

Fig. 8.15: Ce bouton est entouré d'un cercle noir en pointillé, il a le *focus*.

8.4.4 Associer un 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.

> **Définition 8.4.2 : événement**
> Lorsqu'il s'agit d'interface graphique, un événement signifie tout pression d'une touche du clavier, toute pression d'un bouton de la souris, tout déplacement de la souris suceptible d'être utilisé par un objet de l'interface graphique - un bouton, une liste, une zone de saisie... Lorsqu'un objet de l'interface graphique exécute des instructions après la détection d'un événement, on dit qu'il *intercepte* cet événement. Aucun autre objet ne pourra l'intercepter.

Les événements cités sont les plus courants, un événement regroupe toute information extérieure à l'interface graphique et interagissant avec elle - la fin d'une impression par exemple. Toutefois, ceux-ci ne sont pas pris en compte par le module `Tkinter`, seuls la souris et clavier le sont. Les événements sont décrits par la classe `Event`. Cette classe contient les attributs décrits par la table 8.1.

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 :

`char`	Lorsqu'une touche a été pressée, contient son code, ne tient pas compte des touches dites muets comme les touches `shift`, `ctrl`, `alt`. Ne 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 identifiant de l'objet ayant reçu l'événement.
`x`, `y`	Coordonnées relative 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.

Tab. 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)`.

`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.
$<$ `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 sort la zone graphique de l'objet.

Tab. 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 et toute pression du premier bouton de la souris lorsque ce clic a lieu au dessus de la zone graphique du bouton (celui de l'écran). La fenêtre crée par ce programme ainsi que l'affichage qui en résulte apparaissent dans la figure 8.16.

```
1  import Tkinter
2
3  root = Tkinter.Tk ()
4
5  b = Tkinter.Button (text = "appuyer sur une touche")
6  b.pack ()
7
```

```
8   def affiche_touche_pressee (evt) :
9       print "--------------------------- touche pressée"
10      print "evt.char = ", evt.char
11      print "evt.keysym = ", evt.keysym
12      print "evt.num = ", evt.num
13      print "evt.x,evt.y = ", evt.x, ",", evt.y
14      print "evt.x_root,evt.y_root = ", evt.x_root, ",", evt.y_root
15      print "evt.widget = ", evt.widget
16  b.bind ("<Key>", affiche_touche_pressee)
17  b.bind ("<button-1>", affiche_touche_pressee)
18  b.focus_set ()
19
20  root.mainloop ()
```

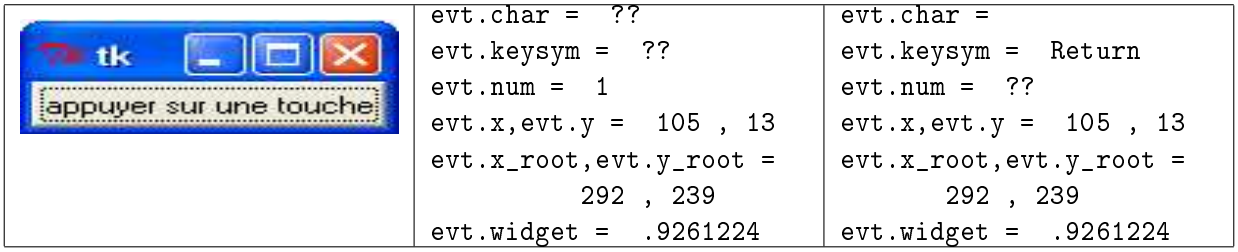

Fig. 8.16: Fenêtre affichée par le programme du paragraphe 8.4.4. La pression d'une touche déclenche l'affichage des caractéristiques de l'événements. La seconde colonne correspond à la pression du premier bouton de la souris. La dernière colonne correspond à la pression de la touche `Return`.

Remarque 8.4.3: 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 8.4.4: mauvais événement

Les messages d'erreur liées aux messages ne sont pas forcément très explicite. Ainsi l'instruction suivante inclut un événement inexistant.

```
b.bind ("<button-1>", affiche_touche_pressee)
```

Lors de l'exécution, le programme déclenche la succession d'exception suivante 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:\python23\lib\lib-tk\Tkinter.py", line 933, in bind
    return self._bind(('bind', self._w), sequence, func, add)
  File "c:\python23\lib\lib-tk\Tkinter.py", line 888, in _bind
    self.tk.call(what + (sequence, cmd))
_tkinter.TclError: bad event type or keysym "button"
```

Remarque 8.4.5: 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)
```

Remarque 8.4.6: 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.

```
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.

8.4.5 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 8.17 et change son contenu toutes les secondes.

```
1  # -*- coding: cp1252 -*-
2  import Tkinter
3  root = Tkinter.Tk ()
4  l = Tkinter.Label (text = "0 secondes")
5  l.pack ()
6  sec = 0
7  id = None
8
9  def change_legende() :
10     global l
11     global sec
12     global id
13     sec += 1
14     l.config (text = "%d secondes" % sec)
15     id = l.after (1000, change_legende)
16
17 l.after (1000, change_legende)
18
19 root.mainloop ()
```

Fig. 8.17: Résultat de l'exemple du paragraphe 8.4.5, 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)
```

8.4.6 Menu

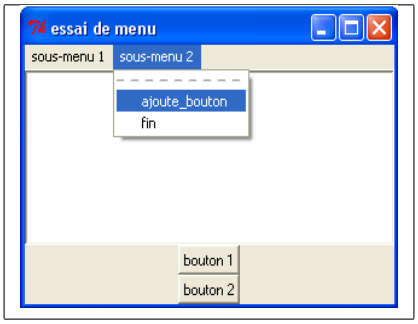

Fig. 8.18:

```
1   import Tkinter
2
3   root = Tkinter.Tk ()
4
5   e = Tkinter.Text (width = 50, height = 10)
6   e.pack ()
7
8   m = Tkinter.Menu ()
9
10  sm1 = Tkinter.Menu ()
11  sm2 = Tkinter.Menu ()
12
13  m.add_cascade (label = "sous-menu 1", menu = sm1)
14  m.add_cascade (label = "sous-menu 2", menu = sm2)
15
16  nb = 0
17
18  def affiche () : print "fonction affiche"
19  def calcul () : print "fonction calcul ", 3 * 4
20  def ajoute_bouton () :
21      global nb
```

```
22      nb += 1
23      b = Tkinter.Button (text = "bouton " + str (nb))
24      b.pack ()
25
26  sm1.add_command (label = "affiche", command = affiche)
27  sm1.add_command (label = "calcul", command = calcul)
28  sm2.add_command (label = "ajoute_bouton", command = ajoute_bouton)
29  sm2.add_command (label = "fin", command = root.destroy)
30
31  root.config (menu = m, width = 200)
32  root.title ("essai de menu")
33  help (Tkinter.Tk)
34  root.mainloop ()
```

8.4.7 Fonctions prédéfinies

root.destroy

root.iconify

root.title (smdfigksjdfmg)

8.5 D'autres fenêtres

8.6 Barre de défilement

8.6.1 Une barre de défilement pour une liste

8.6.2 Une barre de défilement pour une zone de saisie

8.7 Programmes

Le paragraphe 8.1 a utilisé comme exemple la réalisation d'une fenêtre permettant de sélectionner un fichier, fenêtre illustré par la figure 8.1 (page 135). Le premier programme permet de sélectionner un fichier sans interface graphique. Le second fait exactement la même chose mais à l'aide d'une fenêtre graphique.

8.7.1 Sélection d'un fichier sans interface graphique

```
1   # -*- coding: cp1252 -*-
2   """module permettant de sélection un fichier,
3   fonctionne sans interface graphique"""
4   import os.path
5   import os
6
7   class FileSelection (object) :
8       """classe permettant de sélectionner un fichier
9       sans boîte de dialogue"""
10
```

```
11      def __init__ (self, titre = "Sélection de fichier", \
12                    chemin = None, file = True, exist= True) :
13          """initialise la classe
14          @param      titre           titre de la fenêtre
15          @param      chemin          fichier ou répertoire par défaut
16          @param      file            True : fichier, False : répertoire
17          @param      exist           True : le répertoire ou le fichier
18                                              sélectionné doit exister"""
19          self.titre  = titre
20          self.chemin = chemin
21          self.file   = file
22          self.exist  = exist
23
24          if self.chemin == None :  self.chemin = os.getcwd ()
25
26      def get_list (self) :
27          """retourne la liste des fichiers et des répertoires (2 listes),
28          répertoires seulement et [] si self.file == False"""
29          if os.path.isdir (self.chemin) :
30              list    = os.listdir (self.chemin)
31          else :
32              ch,fi   = os.path.split (self.chemin)
33              list    = os.listdir (ch)
34
35          lifile  = []
36          lidir   = []
37          for l in list :
38              if os.path.isdir (self.chemin + "\\" + l) :
39                  lidir.append (l)
40              elif self.file :
41                  lifile.append (l)
42
43          lidir.sort ()
44          lifile.sort ()
45          return lidir, lifile
46
47      def run (self) :
48          """lance la sélection d'un fichier"""
49
50          def update_chemin () :
51              """mise à jour du chemin dans la boîte de dialogue"""
52              pass
53
54          def update_list () :
55              """mise à jour de la liste des fichiers et répertoires
56              à partir de la chaîne dans la boîte de dialogue"""
57              lidir, lifile   = self.get_list ()
58              print "  répertoires"
59              for l in lidir : print "      ", l
60              print "  fichiers"
```

```
61              for l in lifile : print "      ", l
62
63          def precedent () :
64              """passe au répertoire précédent"""
65              if os.path.isdir (self.chemin) :
66                  ch, last    = os.path.split (self.chemin)
67                  self.chemin = ch
68              else :
69                  ch, last    = os.path.split (self.chemin)
70                  ch, last    = os.path.split (ch)
71                  self.chemin = ch
72              #update_chemin ()
73              #update_list ()
74
75          def suivant (sel) :
76              """rentre dans un répertoire"""
77              sel2 = self.chemin + "\\" + sel
78              if os.path.isdir (sel2) :
79                  self.chemin = sel2
80                  #update_chemin ()
81                  #update_list ()
82
83          def update_sel () :
84              """mise à jour de la chaîne de caractères
85              dans la boîte de dialogue à partir de la ligne
86              sélectionnée dans la liste"""
87              pass
88
89          def annuler () :
90              """annule la recherche"""
91              self.resultat = False
92
93          def accepter () :
94              """accepte le résultat"""
95              self.resultat = True
96
97          while True :
98              print "chemin actuel : ", self.chemin
99              print "liste des fichiers et répertoire inclus"
100             update_list ()
101             print "quelle action : Précédent, Entre, Annuler, " \
102                    "Ok (entrer la lettre majuscule)\n"
103             str = raw_input ("")
104             if str == "P" :
105                 precedent ()
106             elif str == "E" :
107                 sel = raw_input ("entrer un nom de répertoire\n")
108                 suivant (sel)
109             elif str == "A" :
110                 annuler ()
```

```
111                 break   # on sort de la boucle
112             elif str == "O" :
113                 print "Entrer un nom de fichier"
114                 str = raw_input ("\n")
115                 self.chemin += "\\" + str
116                 accepter ()
117                 break   # on sort de la boucle
118             else :
119                 print "choix incompréhensible"
120             print "-----------------------------------------"
121
122         if self.resultat : return self.chemin
123         else : return None
124
125
126 if __name__ == "__main__" :
127     r = FileSelection ("sélection d'un fichier", "c:\\")
128     s = r.run ()
129     print "fichier sélectionné ", s
130
131
```

8.7.2 Sélection d'un fichier avec interface graphique

Le programme suivant affiche une fenêtre identique à celle de la figure 8.1 (page 135). Ce programme montre entre autre comment associer une barre de défilement à une liste (lignes 56 à 59 et 72).

```
1  # -*- coding: cp1252 -*-
2  """module contenant une boîte de dialogue permettant
3  de sélectionner un fichier ou un répertoire,
4  il utilise l'interface Tkinter"""
5  import Tkinter as Tk
6  import os.path
7  import os
8
9  class FileSelection (object) :
10     """classe permettant de sélectionner un fichier
11     ou un répertoire à travers une boîte de dialogue"""
12
13     def __init__ (self, titre = "Sélection de fichier", \
14                   chemin = None, file = True, exist= True) :
15         """initialise la classe
16         @param      titre           titre de la fenêtre
17         @param      chemin          fichier ou répertoire par défaut
18         @param      file            True : fichier, False : répertoire
19         @param      exist           True : le répertoire ou le fichier
20                                            sélectionné doit exister"""
21         self.titre  = titre
22         self.chemin = chemin
23         self.file   = file
```

```
24          self.exist  = exist
25
26          if self.chemin == None :  self.chemin = os.getcwd ()
27
28      def get_list (self) :
29          """retourne la liste des fichiers et des répertoires (2 listes),
30          répertoires seulement et [] si self.file == False"""
31          if os.path.isdir (self.chemin) :
32              list    = os.listdir (self.chemin)
33          else :
34              ch,fi   = os.path.split (self.chemin)
35              list    = os.listdir (ch)
36
37          lifile  = []
38          lidir   = []
39          for l in list :
40              if os.path.isdir (self.chemin + "\\" + l) :
41                  lidir.append (l)
42              elif self.file :
43                  lifile.append (l)
44
45          lidir.sort ()
46          lifile.sort ()
47          return lidir, lifile
48
49      def run (self) :
50          """lance la boîte de dialogue et retourne la chaîne sélectionnée"""
51          top         = Tk.Toplevel ()
52          top.wm_title (self.titre)
53          self.resultat = False
54
55          fli     = Tk.Frame (top)
56          scrollbar = Tk.Scrollbar (fli)
57          li      = Tk.Listbox (fli, width = 120, height = 15, \
58                                  yscrollcommand = scrollbar.set)
59          scrollbar.config (command = li.yview)
60          ch      = Tk.Entry (top, width = 120)
61          f       = Tk.Frame (top)
62          prec    = Tk.Button (f, text = "Précédent")
63          suiv    = Tk.Button (f, text = "Entre")
64          annul   = Tk.Button (f, text = "Annuler")
65          ok      = Tk.Button (f, text = "Ok")
66
67          prec.grid (column = 0, row = 0)
68          suiv.grid (column = 1, row = 0)
69          annul.grid (column = 3, row = 0)
70          ok.grid (column = 4, row = 0)
71          li.pack (side = Tk.LEFT)
72          scrollbar.pack(side = Tk.RIGHT, fill = Tk.Y)
73          fli.pack ()
```

```
74          ch.pack ()
75          f.pack ()
76
77          def update_chemin () :
78              """mise à jour du chemin dans la boîte de dialogue"""
79              s = ch.get ()
80              ch.delete (0, len (s))
81              ch.insert (0, self.chemin)
82
83          def update_list () :
84              """mise à jour de la liste des fichiers et répertoires
85              à partir de la chaîne dans la boîte de dialogue"""
86              self.chemin     = ch.get ()
87              lidir, lifile   = self.get_list ()
88              li.delete (0, Tk.END)
89              if len (lidir) > 0 :
90                  for l in lidir : li.insert (Tk.END, "+ "+ l)
91              if len (lifile) > 0 :
92                  for l in lifile : li.insert (Tk.END, "  "+ l)
93
94          def precedent () :
95              """passe au répertoire précédent"""
96              if os.path.isdir (self.chemin) :
97                  ch, last    = os.path.split (self.chemin)
98                  self.chemin = ch
99              else :
100                 ch, last    = os.path.split (self.chemin)
101                 ch, last    = os.path.split (ch)
102                 self.chemin = ch
103             update_chemin ()
104             update_list ()
105
106         def suivant () :
107             """rentre dans un répertoire"""
108             sel = ch.get ()
109             if os.path.isdir (sel) :
110                 self.chemin = sel
111                 update_chemin ()
112                 update_list ()
113
114         def update_sel () :
115             """mise à jour de la chaîne de caractères
116             dans la boîte de dialogue à partir de la ligne
117             sélectionnée dans la liste"""
118             li.after (200, update_sel)
119             sel = li.curselection ()
120             if len (sel) == 1 :
121                 t = li.get (sel [0])
122                 c = self.chemin + "\\" +  t [2:len (t)]
123                 s = ch.get ()
```

```
124                 ch.delete (0, len (s))
125                 ch.insert (0, c)
126
127         def annuler () :
128             """annule la recherche"""
129             self.resultat = False
130             top.destroy ()
131             top.quit ()
132
133         def accepter () :
134             """accepte le résultat"""
135             self.resultat    = True
136             self.chemin = ch.get ()
137             top.destroy ()
138             top.quit ()
139
140         prec.config (command = precedent)
141         suiv.config (command = suivant)
142         annul.config (command = annuler)
143         ok.config (command = accepter)
144
145         update_chemin ()
146         update_list ()
147         update_sel ()
148         ch.focus_set ()
149         top.mainloop ()
150
151         if self.resultat : return self.chemin
152         else : return None
153
154
155 if __name__ == "__main__" :
156     #root = Tk.Tk ()
157
158     def run () :
159         r = FileSelection ("sélection d'un fichier", "c:\\")
160         s = r.run ()
161         print "fichier sélectionné ", s
162
163     run ()
```

10.2 Les variables

Le nom d'une variable commence par une lettre ou un blanc souligné, il peut également inclure par la suite des chiffres. *Python* distingue les minuscules des majuscules. La portée d'une variable, c'est-à-dire la portion de code où elle définie, s'étend depuis sa déclaration (première affectation) jusqu'à la fin du programme ou de la fonction ou de la boucle ou du test où elle est définie.

Pour déclarer une variable portant le nom `va`, il suffit d'écrire :

```
va = <valeur>
```

Le type de < `valeur` > détermine le type de la variable `va`. Si une variable de même portée portait déjà ce nom-là, son contenu est écrasé (perdu aussi). L'instruction `type(x)` retourne le type de la variable `x`.

Un identifiant ne peut être utilisé qu'une seule fois, qu'il désigne une variable, une fonction, une classe ou un module.

10.2.1 Les types immuables

Les variables de type immuable ne peuvent pas être modifiées.

1. `None`, ce type veut dire rien, il est utilisé comme convention de programmation pour dire qu'un algorithme, un calcul ne s'est pas terminé correctement ou une valeur n'a pas encore été calculée.
2. `bool` : un booléen (résultat d'un test)
3. `int` : un entier
4. `float` : un réel
5. `complex` : un complexe
6. `str` : une chaîne de caractères ou string, elle apparaît entre guillements, entre apostrophes, entre trois guillements (`"""`) si elle s'étend sur plusieurs lignes. `s` = "`exemple`".
7. `tuple` : un vecteur d'éléments de types identiques ou différents, il apparaît entre parenthèses, on accède à un de ses éléments à l'aide de crochets. Les éléments d'un tuple `t` sont indicés de 0 à `len(t)` − `1` inclus.
   ```
   t = ()  # tuple vide
   t = (2, "e") # tuple de deux éléments
   print t[0]   # affiche le premier éléments
   ```

L'affectation d'une valeur de type immuable à une variable est une copie. On peut appliquer sur les types numériques les opérations usuelles (+ * - / % ** += *= -= /= %= **=)[1]. On rappelle que `a+ = 10` est équivalent à `a = a + 10`, ceci signifie que la valeur de `a` avant le calcul n'a plus besoin d'exister. Le *et* logique et le *ou* logique sont notés `and` et `or`. Les priorités sont celles usuellement utilisées en mathématique, en cas de doute, il faut utiliser des parenthéses.

Les opérateurs de comparaison (< > == <= >=) s'appliquent sur tous les types numériques ainsi que sur les chaînes de caractères. Rappel : les minuscules sont classées après les majuscules.

Fréquente source de bug : une division entiére a pour résultat le quotient et non un nombre décimal. Autrement dit : `1/2 = 0` **et non** `0.5`.

Pour convertir une information d'un type à un autre, il suffit d'utiliser le nom de ce type suivi de la valeur à convertir entre parenthèses : `b = float("2.145")` équivaut à la conversion d'une chaîne de caractères en réel.

[1] ** est le symbole pour puissance : $3**4 = 3^4$

L'addition d'un tuple et d'une valeur retourne un tuple incluant cette valeur à la fin (plus long d'un élément). L'addition de deux t-uples concatène les deux tuples. L'addition de deux chaînes de caractères retourne leur concaténation.

Pour savoir si un élément `x` fait partie d'un tuple `t`, il faut utiliser la syntaxe `x in t` dont la réciproque est `x not in t`.

La fonction `len` retourne la longueur d'un tuple ou d'une chaîne de caractères. Les éléments ou les caractères d'un tuple ou d'une chaîne de caractères `t` sont indicés de 0 à `len(t) - 1` inclus.

```
t [i:j]  # correspond à un sous-ensemble allant des indices i à j exclu
t [:j]   #  = t[0:j]
t [i:]   # = t [i: len (t)]
```

Pour les chaînes de caractères, on utilise fréquemment les méthodes de la table 10.1, exemple :

```
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
```

`count(sub[, start[, end]])`	Retourne le nombre d'occurences de la chaîne de caractères `sub`, les paramètres par défaut `start` et `end` permettent de réduire la recherche entre les caractères d'indice `start` et `end` exclu. Par défaut, `start` est nul tandis que `end` corresond à la fin de la chaîne de caractères.
`find(sub[, start[, end]])`	Recherche une chaîne de caractères `sub`, les paramètres par défaut `start` et `end` ont la même signication 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[, count])`	Retourne une copie de la chaîne de caractères en remplaçant toutes les occurrences de la chaîne `sub` par `new`. Si le paramètre optionnel `count` est renseigné, alors seules les `count` premiéres occurrences seront remplacées.
`split([sep[, maxsplit]])`	Découpe la chaîne de caractères en se servant de la chaîne `split` comme délimiteur. Si le paramètre `maxsplit` est renseigné, au plus `maxsplit` coupures seront effectuées.
`upper()`	remplace les minuscules par des majuscules
`lower()`	remplace les majuscules par des minuscules

Tab. 10.1: Quelques fonctions s'appliquant aux chaînes de caractères, l'aide associée au langage *Python* fournira la liste complète. Certains des paramètres sont encadrés par des crochets, ceci signifie qu'ils sont facultatifs (voir également page 25).

L'affichage de réels nécessite parfois de tronquer la partie décimale ce qui est fait grâce à la syntaxe suivante (voir également page 25) :

```
x = 0.123456789
print "%1.2f" % x    # donne 0.12
s = "%2.2e %s" % (3.14, "est une approximation de pi")
print s
```

10.2.2 Les types modifiables

Python fournit deux types modifiables : les listes et les dictionnaires. Pour ces deux types, **il faut faire attention à chaque affectation**.

```
a = [1,2]
b = a
```

La seconde ligne ne fait pas une copie de la première liste, elle ne fait que créer un second nom pour nommer la même liste. Pour copier une liste ou un dictionnaire, il faut utiliser :

```
a = [1,2]
import copy
b = copy.copy (a)
```

ou, si la liste inclut également d'autres listes ou dictionnaires :

```
a = [1,2]
import copy
b = copy.deepcopy (a)
```

Cette remarque s'applique à tout type modifiable, liste, dictionnaire ou tout autre classe.

La suppression d'une variable n'implique pas la suppression de toutes les variables se référant à une seule et même instance de classe.

10.2.2.1 Liste

Une liste permet est une sorte de tableau mémoriser un ensemble d'éléments de types variés. C'est un tuple modifiable. La table 10.2 regroupe les opérations qu'une liste supporte et la table 10.3 les méthodes dont elle dispose.

```
x = [4,5]               # création d'une liste composée de deux entiers
x = ["un",1,"deux",2]   # création d'une liste composée 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
```

Les listes peuvent aussi être définies à partir d'une écriture abrégée :

```
x = range(0,5)                          # liste des entiers de 0 à 5 exclu
y = [ i for i in x if i % 2 == 0]   # sélection les éléments pairs
print y                             # affiche [0,2,4]
z = [ i+j for i in x for j in x]    # 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]
```

`x in s`	vrai si `x` est un des éléments de `l`
`x not in s`	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 le $i^{ème}$ élément de `l`, à la différence des T-uples, l'instruction `l[i] = "3"` est valide, elle remplace l'élément `i` par 3.
`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`, résultat difficile à prévoir lorsque les types des éléments sont différents
`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

Tab. 10.2: 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 (voir également page 29).

10.2.2.2 Dictionnaire

Un dictionnaire est un tableau pour lequel les indices ou clés ne sont pas uniquement des entiers mais tout type non modifiable (le plus souvent un entier, un réel, une chaîne de caractères, un tuple).

```
x = { "cle1":"valeur1", "cle2":"valeur2" }
print x ["cle1"]   # affiche valeur1
x [(0,1)]  = "clé tuple"  # ajoute une nouvelle valeur dont la clé est (0,1)
y = { }         # crée un dictionnaire vide
z = dict ()     # crée aussi un dictionnaire vide
```

La table 10.4 regroupe les opérations qu'un dictionnaire supporte. La table 10.5 regroupe les méthodes d'un dictionnaire.

10.2.2.3 Tableaux

Ce type ne fait pas partie du langage *Python* standard mais il est couramment utilisé.

```
import Numeric
a = Numeric.array ( [0,1] )
```

Il permet de convertir des listes en une structure plus appropriée au calcul. En contrepartie, il n'est pas aussi rapide d'ajouter ou supprimer des éléments.

`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, une exception est déclenchée (voir le paragraphe 5).
`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 occurence 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 dont la valeur est d'abord retournée puis il est 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])`	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).

Tab. 10.3: Opérations permettant de modifier une listes on suppose que `l` est une liste, `x` est quant à lui quelconque (voir également page 30).

10.3 Tests et boucles

10.3.1 Tests

Les tests permettent d'exécuter telle ou telle instruction selon la valeur d'une condition. Le test est suivi de : et les instructions dépendant de ce test sont indentées (décalées vers la droite).

```
if x < 5 :
    print x
    ...
```

Il peut y avoir une contrepartie :

```
if x < 5 :
    print x
    ...
else :
    print 5-x
    ...
```

S'il n'y a qu'une seule instruction, elle peut s'écrire en bout de ligne :

`x in d`	vrai si `x` est une des clés de `d`
`x not in d`	réciproque de la ligne précédente
`l[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 l[i]`	supprime l'élément associée 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, particulier, `d` est égal à `dict(d.items())`

Tab. 10.4: Opérations disponibles sur les dictionnaires, `d` est un dictionnaire, `x` est quant à lui quelconque (voir également page 34).

`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.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)`	Pour chaque clé de `d1`, `d[k]` = `d1[k]`
`d.setdefault(k[,x])`	Retourne `d[k]` si la clé `k` existe, sinon, affecte `x` à `d[k]`.
`d.popitem()`	Retourne un élément et le supprime du dictionnaire.

Tab. 10.5: Méthodes associées aux dictionnaires, `d`, `d2` sont des dictionnaires, `x` est quant à lui quelconque (voir également page 35).

```
if x < 5 :  print x
else :    print 5-x
```

Il peut y avoir plusieurs conditions qui s'enchaînent :

```
if x < 5 :  print "x < 5"
elif x > 5 : print "x > 5"
else :  print "x == 5"
```

10.3.2 Boucles

Il y a deux types de boucles, la boucle `for` parcourt un ensemble, la boucle `while` continue tant qu'une condition est vraie. Comme pour les tests, une boucle est suivie de :, les lignes incluses dans cette boucle sont indentées à moins qu'il n'y en ait qu'une seule auquel cas elle peut être écrite après le symbole : sur la même ligne.

```
while condition :
    # lignes décalées
    # contenu de la boucle
```

Quelques exemples de boucles `for` :

```
for i in range(0,n) :   # parcourt tous les entiers de 0 à n-1 inclus
for i in xrange(0,n) :   # même chose mais en plus rapide
for i in xrange(n,0,-1) :   # parcourt tous les entiers de n à 1 inclus dans le sens décroissant
for i in xrange(2,1000,3) :   # parcourt tous les entiers de 2 à 1000 de 3 en 3 (2,5,8,...)
for e in li :   # parcourt tous les éléments de la liste li
for cle,valeur in di.items () :   # parcourt tous les éléments du dictionnaire di
```

Pour toutes les boucles, l'instruction **`break`** permet de sortir de la boucle, l'instruction **`continue`** passe directement à l'itération suivante sans exécuter les instructions qui suivent l'instruction **`continue`**.

10.4 Fonctions

Les fonctions ou sous-programmes permettent de faire la même chose sans avoir à recopier le code informatique plusieurs fois dans le programme. Elle accepte aucun ou plusieurs paramètres, elle peut retourner aucun ou plusieurs résultats. Leur déclaration suit le schéma suivant :

```
def exemple_fonction (p1, p2, p3) :
    # code de la fonction
    return r1, r2

a,b = exemple_fonction (1,2,3)   # exemple d'appel de la fonction
```

L'instruction **`return`** n'est pas obligatoire mais si elle est présente à un ou plusieurs endroits, aucune autre instruction de la fonction ne sera exécutée après l'exécution de la première instruction **`return`** rencontrée lors de l'exécution de la fonction. Les fonctions peuvent être récursives et inclure des paramètres par défaut : ces paramètres reçoivent une valeur même si celle-ci n'est pas précisée lors de l'appel.

```
def exemple_fonction (p1, p2 = 4, p3 = 7) :
    # code de la fonction
    return r1, r2

a,b = exemple_fonction (1)   # = exemple_fonction (1,4,7)
a,b = exemple_fonction (1,2,3)   # = exemple_fonction (1,2,3)
a,b = exemple_fonction (1,2)   # = exemple_fonction (1,2,7)
a,b = exemple_fonction (1,p3 = 2)   # = exemple_fonction (1,4,2)
```

Les paramètres par défaut doivent tous être mis en fin de déclaration, l'exemple suivant n'est pas correcte :

```
def exemple_fonction (p1, p2 = 4, p3) :
    # code de la fonction
    return r1, r2
```

En ce qui concerne les paramètres, les paramètres de type non modifiable sont passés par valeur (une modification à l'intérieur de la fonction n'a pas de répercution à l'extérieur).

```
def exemple_fonction (p1) :
    p1 = 3
a = 1
exemple_fonction (a)
print a  # affiche 1
```

Les paramètres de type modifiables sont passés par référence (une modification à l'intérieur de la fonction a des répercutions à l'extérieur).

```
def exemple_fonction (p1) :
    p1[0] = 3
a = [1]

exemple_fonction (a)
print a  # affiche [3]
```

10.5 Classes

Les classes sont un moyen de définir de nouveaux types modifiables de variables. C'est un ensemble d'attributs (ou variables) et de méthodes (ou fonctions). Un programme utilisant les classes est orienté objet. Peu de programmes ne les utilisent pas. Il est possible de faire les mêmes choses avec ou sans classes mais leur utilisation rend d'ordinaire les grands programmes plus facile à comprendre et à construire.

10.5.1 Déclaration d'une classe

Pour déclarer une classe, on procède comme suit :

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
        self.att4 = att1 * att2 * att3

a = ma_classe (-1,1,2) # déclare une variable de type ma_classe
print a.att1  # affiche -1
print a.att2  # affiche 3
print a.att3  # affiche 4
print a.att4  # affiche -12
```

Lors de la déclaration de la variable `a`, le langage *Python* exécute la méthode `__init__` aussi appelée constructeur. Elle permet de définir les attributs de la classe directement à partir des paramètres ou comme le résultat d'un calcul ou d'une fonction. Le constructeur comme toutes les autres méthodes possède comme premier paramètre `self` qui permet d'accéder aux attributs et aux méthodes de la classe. Le programme suivant est équivalent au premier.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
        self.att4 = self.calcule4 ()

    def calcule4 (self) :
        return self.att1 * self.att2 * self.att3

a = ma_classe (-1,1,2) # déclare une variable de type ma_classe
print a.att1  # affiche -1
print a.att2  # affiche 3
print a.att3  # affiche 4
print a.att4  # affiche -12
```

10.5.2 Attributs et méthodes

Les attributs sont déclarés le plus souvent à l'intérieur du constructeur, plus généralement à l'intérieur de toute méthode, voire à l'extérieure de la classe. Pour y accéder à l'intérieur d'une méthode on fait précéder le nom de l'attribut de `self.`, à l'extérieur de la classe, c'est le nom de l'instance suivi d'un point `a.` qui précède le nom de l'attribut comme le montre le précédent exemple.

Une méthode est déclarée à l'intérieur de la classe. Elle accepte invariablement au moins un paramètre qui est `self` comme dans le précédent exemple. Les règles d'accès sont les mêmes que pour les attributs. Elles acceptent également la récursivité et les paramètres par défaut à l'exception du premier.

Chaque instance de classe est également munie d'un dictionnaire `__dict__` qui recense tous les attributs.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
        self.att4 = att1 * att2 * att3

a = ma_classe (1,2,3)
print a.att1                          # affiche 1
print a.__dict__ ["att1"]    # affiche aussi 1, ligne équivalente à la précédente
```

10.5.3 Opérateurs

Les opérateurs sont des méthodes qui permettent une manipulation plus simple des objets. Leur nom est fixé par convention par le langage *Python*, ils commencent et terminent par `__`.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
```

```
        self.att4 = att1 * att2 * att3

    def __add__ (self, a) :
         return ma_classe (self.att1 + a.att1, self.att2 + a.att2, \
                       self.att3 + a.att3, self.att4 + a.att4)

a = ma_classe (1,2,3)
b = ma_classe (4,5,6)
c = a + b  # n'a de sens que si l'opérateur __add__ a été redéfini
```

Il existe un opérateur spécifique pour chaque opération, cet opérateur permet de donner un sens à une addition, une soustraction, ... de deux instances d'une classe. L'opérateur **__str__** retourne une chaîne de caractères et est appelée par l'instruction **print**. L'opérateur **__cmp__** retourne un entier permettant à des instances de la classe d'être comparées et triées par une liste.

10.5.4 Copie d'instances

Les instances de classes sont des objets modifiables, comme pour les listes, une simple affectation ne signifie pas une copie mais un second nom pour désigner le même objet.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
        self.att4 = att1 * att2 * att3

a = ma_classe (1,2,3)
b = a
b.att1 = -16
print a.att1  # affiche -16
print b.att1  # affiche -16
```

Il faut donc copier explicitement l'instance pour obtenir le résultat souhaité.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
        self.att4 = att1 * att2 * att3

a = ma_classe (1,2,3)
import copy
b = copy.copy (a)
b.att1 = -16
print a.att1  # affiche 1
print b.att1  # affiche -16
```

Lorsque une classe inclut une variable de type classe, il faut utiliser la fonction **deepcopy** et non **copy**.

10.5.5 Héritage

L'héritage est l'intérêt majeur des classes et de la programmation orientée objet. Lorsqu'une classe hérite d'une autre, elle hérite de ses attributs et de ses méthodes. Le simple fait d'hériter crée donc une classe équivalente.

```
class ma_classe :
    def __init__ (self, att1, att2, att3) :
        self.att1 = att1
        self.att2 = att2
        self.att3 = att3
        self.att4 = att1 * att2 * att3

class ma_classe2 (ma_classe) :  # héritage simple
    pass                                    # pour dire que la classe est vide
```

Mais hériter permet de faire deux choses :

1. ajouter des attributs et ajouter des méthodes
2. modifier le comportement d'une méthode existante

```
class ma_classe :
    def __init__ (self, att1) :
        self.att1 = att1
        self.att2 = self.calcul ()

    def calcul (self) :
        return self.att1 ** 2

class ma_classe2 (ma_classe) :
    def calcul (self) :
        # dans cette méthode, on change le comportement
        # de la méthode calcul tout en se servant de celui
        # de la classe mère
        return ma_classe.calcul (self) * self.att1

a = ma_classe (2)
b = ma_classe2 (2)
print a.att2   # affiche 4  = 2 * 2
print b.att2   # affiche 8   = (2*2) * 2
```

10.6 Fichiers

L'écriture et la lecture dans un fichier s'effectue toujours de la même manière. On ouvre le fichier en mode écriture ou lecture, on écrit ou on lit, puis on ferme le fichier, le laissant disponible pour une utilisation ultérieure. Ce paragraphe ne présente l'écriture ou l'écriture dans un format binaire car celle-ci est peu utilisée dans ce langage.

10.6.1 Ecriture dans un fichier texte

L'écriture dans un fichier texte s'effectue toujours selon le même schéma :

```
f = open ("nom-fichier", "w")    # ouverture en mode écriture "w" ou écriture ajout "a"

f.write (  s )    # écriture de la chaîne de caractères  s
f.write (  s2 )    # écriture de la chaîne de caractères  s2
...

f.close ()  # fermeture
```

Certains codes sont fort utiles lors de l'écriture de fichiers texte :

- `\n` passage à la ligne
- `\t` insertion d'une tabulation, indique un passage à la colonne suivante dans le logiciel Excel

10.6.2 Lecture dans un fichier texte

La lecture est le symétrique de l'écriture. En voici un exemple, la seule chose qui change d'un programme à l'autre est ce qu'on fait des lignes extraites.

```
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 10.6.1: code de fin de ligne
Lors 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

l_net = []                      # contiendra la liste nettoyée des lignes du fichier
for s in l :
    s2 = s.replace ("\n", "")        # on supprime le code de fin de ligne \n
    s2 = s2.replace ("\r", "")  # on supprime le code de fin de ligne \r (Windows uniquement)
    l_net.append (s2)               # on ajoute le résultat à la liste nettoyée
```

Remarque 10.6.2: fichier formaté
Les fichiers textes ont de nombreux formats, on peut citer HTML ou XML qui sont des formats à balises. Leur lecture utilise des modules comme `HTMLParser` ou `xml.sax` dont la description sort du cadre de ce document. Un autre format est souvent utilisé avec le logiciel *Excel* ou tout autre tableur. 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 colonnes délimitées par un séparateur qui est souvent une tabulation (`\t`) ou un point virgule.

```
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.

```
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 :
    s2 = s.replace ("\n", "")        # on supprime le code de fin de ligne \n
    s2 = s2.replace ("\r", "")  # on supprime le code de fin de ligne \r (Windows uniquement)
    case = s2.split (";")
    if len (case) >= 3 :
        print case [1], " ", case [0], " a écrit ", case [2]
```

10.7 Modules

Le concept de module permet de répartir différentes parties d'un programme sur plusieurs fichiers. Il existe deux types de modules : ceux disponibles sur Internet (programmés par d'autres) et ceux que l'on programme soi-même. Les premiers sont souvent fournis avec un programme d'installation automatique ou dans le cas où ils sont manquants, des instructions permettant de l'installer. Les seconds sont écrits dans le même répertoire que le fichier principal.

On enregistre le module suivant sous le nom `geometrie.py`.

```
# définition du module geometrie.py

def carre (x) :
    return x ** 2

class point :
    def __init__ (self,x,y) :
        self.x, self.y = x,y

    def norme (self) :
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

Pour utiliser une fonction ou une classe du module `geometrie.py`, on utilise une des syntaxes suivantes :

1. Première syntaxe :
   ```
   import geometrie
   print geometrie.carre (1.5)
   p = geometrie.point (1,2)
   ```
2. Deuxième syntaxe :

```
import geometrie as GEO  # on donne un pseudonyme au module geometrie
print GEO.carre (1.5)
p = GEO.point (1,2)
```

3. Troisième syntaxe : le module est utilisé très souvent, même un pseudonyme est trop long, il faut néanmoins s'assurer que les modules importés de cette même manière n'incluent pas des fonctions ou classes portant des noms identiques. Dans ce cas, c'est toujours le dernier qui gagne.

```
from  geometrie import *
print carre (1.5)
p = point (1,2)
```

Dans le cas des modules installés, les trois syntaxes d'utilisation sont aussi valables. On voit aussi souvent apparaître dans un module la condition :

```
if __name__ == "__main__" :
    # quelques instructions ici
```

Ces instructions ne sont exécutées que si le module est utilisé en tant que programme principal. Lorsque ce fichier est importé, elles ne sont jamais exécutées. Cela permet d'écrire des instructions qui permettent de vérifier si le module ne contient pas d'erreurs. Une fois cette étape effectuée, il ne sert à rien de la répéter à chaque fois que le module est importé. C'est pourquoi elles ne sont exécutées que si la condition `if __name__ == "__main__"` : est vérifiée, c'est-à-dire si le module est le programme principal et non un module.

10.8 Exceptions

Le petit programme suivant déclenche une erreur parce qu'il effectue une division par zéro.

```
def inverse (x):
    y = 1.0 / x
    return y
b = inverse (0)
print b
```

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 :

```
def inverse (x):
    y = 1.0 / x
    return y
try :
    b = inverse (0)  # déclenche une exception
```

```
    print b
except :
    print "le programme a déclenché une erreur"
```

On protège la partie du code à l'aide des mots-clé `try` et `except`. Entre ces deux instructions, s'il se produit une erreur, le programme passe immédiatement à ce qui suit l'instruction `except`. On peut même récupérer le message d'erreur correspondant :

```
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
```

On peut aussi décider que le programme agira différemment selon l'erreur produite. Dans l'exemple suivant, le programme teste d'abord si l'erreur est de type `ZeroDivisionError` auquel cas il affiche le message *division par zéro*. Pour un autre type d'erreur, il regarde s'il y a d'autres instructions `except` qui s'y rapportent. S'il y en a une, il exécute les lignes qui la suivent, sinon, le programme s'arrête et déclenche une erreur.

```
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
```

Les instructions `try` et `except` peuvent apparaître dans le programme principal, dans une boucle, un test, une fonction, s'imbriquer les unes dans les autres. Il est possible de déclencher soi-même une exception avec l'instruction `raise` et ou de définir ses propres exceptions en créant une classe héritant d'une classe d'exception. L'exemple suivant regroupe tous ces cas.

```
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, lancée 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 AucunChiffre.__doc__, " : ", exc
    print "fonction : ", exc.f
```

10.9 Erreurs, confusions fréquentes

10.9.1 Variables

10.9.1.1 Chaînes de caractères = tableau de caractères

Une chaîne de caractères est un tableau de caractères : pour accéder à un caractère, on procède comme pour une liste.

```
s = "abcdefghijklmnopqrstuvwxyz"
print s [4] # affiche "e"
print s [4:6] # affiche "ef"
```

10.9.1.2 Guillemets ou pas

Doit-on mettre des guillemnts ou non ?

```
l = [ un, deux, trois, quatre ]
up = []
for i in range (0, len (l)) :
    up.append ( l [i].upper () )
```

Le code précédent ne fonctionne pas car il n'y a pas de guillemet autour de `un`, `deux`, `trois`, `quatre`. Le langage considère alors ces quatre mots comme des variables : un identifiant qui désigne une information. Mais comme ces variables n'existent pas, ces identifiants ne sont reliés à aucun contenu et l'interpréteur Python ne comprend pas.

Un mot entouré de guillemets définit un contenu. Sans guillemet, il définit une variable qui permet de manipuler un contenu tout simplement en donnant la possibilité au programmeur de le nommer. Autrement dit, pour manipuler une chaîne de caractères, il faut affecter ce contenu à une variable. Les guillemets n'apparaissent plus par la suite car on doit utiliser la variable pour la manipuler.

10.9.2 Boucles

10.9.2.1 `range` ou pas `range`

Les deux programmes suivant sont équivalents. La seule différences réside dans l'écriture dans la boucle `for` qui utilise dans le premier cas la fonction `range` et dans l'autre non.

```
l = [ "un", "deux", "trois", "quatre" ]
up = []
for i in range (0, len (l)) :
    up.append ( l [i].upper () )
```

Lorsqu'on utilise la fonction `range`, on dispose lors de la boucle de deux informations, l'indice `i` et l'élément `l`[`i`]. Si l'indice n'est pas utile, il est possible de simplifier la boucle comme suit.

```
l = [ "un", "deux", "trois", "quatre" ]
up = []
for m in l :
    up.append ( m.upper () )
```

En général, on se sert de la boucle qui utilise la fonction `range` dans deux cas :

1. On souhaite faire des opérations sur les éléments qui précèdent ou suivent l'élément en question, ce qui nécessite de connaître l'indice.
2. On parcourt deux listes de même taille à la fois : l'indice désigne la position de deux éléments, un dans chaque liste.

10.9.2.2 Initialisation

Une boucle est souvent utilisée pour faire une somme, calculer un maximum : garder un unique résultat en parcourant une liste. Une boucle de ce type est toujours précédée d'une étape d'initialisation qui consiste à donner une valeur au résultat : celle qu'il aurait si la liste était vide.

```
l = [ "un", "deux", "trois", "quatre" ]
s = ""
for m in l :
    s += m # concaténation des mots en une seule chaîne de caractères
```

10.9.3 Fonctions

10.9.3.1 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 seule 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 : 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, `calcul4` par exemple.

Bibliographie

[Martelli2004] Alex Martelli, *Python en concentré*, O'Reilly, édition française (2004)

[Swinnen2003] Gérard Swinnen, *Apprendre à programmer avec Python*, O'Reilly, édition française (2003)

[www-DocF] , *documents rédigés en langue française*, http ://www.developpez.com/cours/ (2004)

[www-GPL] , *site officiel de la licence GPL*, http ://www.gnu.org/copyleft/gpl.html (2004)

[www-Python] , *site officiel du langage Python*, http ://www.python.org (2004)

[www-PyDoc] , *site officiel du langage Python, documentation*, http ://docs.python.org/download.html (2004)

[www-PyEdit] , *éditeurs de texte pour le langage Python*, http ://www.python.org/cgi-bin/moinmoin/PythonEditors (2004)

[www-PyDownload] , *site officiel du langage Python, téléchargement du programma d'installation*, http ://www.python.org/download/ (2004)

[www-PyLicence] , *site officiel du langage Python, licence*, http ://www.python.org/psf/license.html (2004)

[www-PyModule] , *site officiel du langage Python, liste des modules existant*, http ://www.python.org/pypi (2004)

[www-Scite] , *éditeur de texte Scite*, http ://www.scintilla.org/SciTE.html (2004)

[www-SciteDownload] , *éditeur de texte Scite, page de téléchargement*, http ://scintilla.sourceforge.net/SciTEDownload.html (2004)

[www-Syn] , *éditeur de texte Syn Text Editor*, http ://syn.sourceforge.net/ (2004)

Table des figures

Liste des tableaux

Table des matières

Index

N

T