Gérard Swinnen

Apprendre à programmer avec

Python

Une version numérique de ce texte peut être téléchargée librement à partir du site :
http://www.ulg.ac.be/cifen/inforef/swi

Quelques pages de cet ouvrage sont adaptées de :
How to think like a computer scientist
de Allen B. Downey, Jeffrey Elkner & Chris Meyers
disponible sur : *http://thinkpython.com*
ou : *http://www.openbookproject.net/thinkCSpy*

Copyright (C) 2000-2008 Gérard Swinnen

L'ouvrage qui suit est distribué suivant les termes de la Licence de Documentation Libre GNU (*GNU Free Documentation License, version 1.1*) de la *Free Software Foundation*. Cela signifie que vous pouvez copier, modifier et redistribuer ces pages tout à fait librement, pour autant que vous respectiez un certain nombre de règles qui sont précisées dans cette licence, dont le texte complet peut être consulté dans l'annexe intitulée « *GNU Free Documentation licence* », page 365.

Pour l'essentiel, sachez que vous ne pouvez pas vous approprier ce texte pour le redistribuer ensuite (modifié ou non) en définissant vous-même d'autres droits de copie. Le document que vous redistribuez, modifié ou non, doit obligatoirement inclure intégralement le texte de la licence citée ci-dessus, le présent avis, la préface qui suit, ainsi que la section *preface* du texte original américain (voir annexes). L'accès à ces notes doit rester libre pour tout le monde. Vous êtes autorisé à demander une contribution financière à ceux à qui vous redistribuez ces notes, mais la somme demandée ne peut concerner que les frais de reproduction. Vous ne pouvez pas redistribuer ces notes en exigeant pour vous-même des droits d'auteur, ni limiter les droits de reproduction des copies que vous distribuez. La diffusion commerciale de ce texte en librairie, sous la forme classique d'un manuel imprimé, est réservée exclusivement à la maison d'édition Eyrolles (Paris).

La couverture

Choisie délibérément hors propos, l'illustration de couverture est un dessin réalisé par l'auteur à la mine de graphite sur papier Canson en 1987, d'après une photographie ancienne. Il représente le yacht de course de 106 tonnes Valdora participant à une régate dans la rade de Cowes en 1923.

Construit vingt ans plus tôt, et d'abord gréé en yawl, Valdora remporta plusieurs trophées avant d'être regréé en ketch en 1912 avec la voilure de 516 m² que l'on voit sur le dessin.

Ce superbe voilier, très estimé par ses équipages pour son bon comportement à la mer, a navigué pendant près d'un demi-siècle.

Grace Hopper, inventeur du compilateur :
« Pour moi, la programmation est plus qu'un art appliqué important. C'est aussi une ambitieuse quête menée dans les tréfonds de la connaissance. »

À Maximilien, Élise, Lucille, Augustin et Alexane.

Préface

En tant que professeur ayant pratiqué l'enseignement de la programmation en parallèle avec d'autres disciplines, nous croyons pouvoir affirmer qu'il s'agit là d'une forme d'apprentissage extrêmement enrichissante pour la formation intellectuelle d'un jeune, et dont la valeur formative est au moins égale, sinon supérieure, à celle de branches plus classiques telles que le latin.

Excellente idée donc, que celle de proposer cet apprentissage dans certaines filières, y compris de l'enseignement secondaire. Comprenons-nous bien : il ne s'agit pas de former trop précocement de futurs programmeurs professionnels. Nous sommes simplement convaincus que l'apprentissage de la programmation a sa place dans la formation générale des jeunes (ou au moins d'une partie d'entre eux), car c'est une extraordinaire école de logique, de rigueur, et même de courage.

À l'origine, le présent ouvrage a été rédigé à l'intention des élèves qui suivent le cours *Programmation et langages* de l'option *Sciences & informatique* au 3e degré de transition de l'enseignement secondaire belge. Il nous a semblé par la suite que ce cours pouvait également très bien convenir à toute personne n'ayant encore jamais programmé, mais souhaitant s'initier à cette discipline en autodidacte.

Nous y proposons une démarche d'apprentissage non linéaire qui est très certainement critiquable. Nous sommes conscients qu'elle apparaîtra un peu chaotique aux yeux de certains puristes, mais nous l'avons voulue ainsi parce que nous sommes convaincus qu'il existe de nombreuses manières d'apprendre (pas seulement la programmation, d'ailleurs), et qu'il faut accepter d'emblée ce fait établi que des individus différents n'assimilent pas les mêmes concepts dans le même ordre. Nous avons donc cherché avant tout à susciter l'intérêt et à ouvrir un maximum de portes, en nous efforçant tout de même de respecter les principes directeurs suivants :

- L'apprentissage que nous visons se veut généraliste : nous souhaitons mettre en évidence les invariants de la programmation et de l'informatique, sans nous laisser entraîner vers une spécialisation quelconque, ni supposer que le lecteur dispose de capacités intellectuelles hors du commun.
- Les outils utilisés au cours de l'apprentissage doivent être modernes et performants, mais il faut aussi que le lecteur puisse se les procurer en toute légalité à très bas prix pour son usage personnel. Notre texte s'adresse en effet en priorité à des étudiants, et toute notre démarche d'apprentissage vise à leur donner la possibilité de mettre en chantier le plus tôt possible des réalisations personnelles qu'il pourront développer et exploiter à leur guise.
- Nous avons pris le parti d'aborder très tôt la programmation d'une interface graphique, avant même d'avoir présenté l'ensemble des structures de données disponibles, parce que cette programmation présente des défis qui apparaissent plus concrets aux yeux d'un programmeur débutant. D'autre part, nous observons que les jeunes qui arrivent aujourd'hui dans nos classes « baignent » déjà dans une culture informatique à base de fenêtres et autres objets graphiques interactifs. S'ils choisissent d'apprendre la programmation, ils sont forcément impatients de créer par eux-mêmes des applications (peut-être très simples) où l'aspect graphique est déjà bien présent. Nous avons donc choisi cette approche un peu inhabituelle afin de permettre au lecteur de se lancer très tôt dans de petits projets personnels attrayants, par lesquels ils puisse se sentir valorisé. En revanche, nous laisserons délibérément de côté les environnements de

programmation sophistiqués qui écrivent automatiquement de nombreuses lignes de code, parce que nous ne voulons pas non plus masquer la complexité sous-jacente.

Certains nous reprocheront que notre démarche n'est pas suffisamment centrée sur l'algorithmique pure et dure. Nous pensons que celle-ci est moins primordiale que par le passé. Il semble en effet que l'apprentissage de la programmation moderne par objets nécessite plutôt une mise en contact aussi précoce que possible de l'apprenant avec des objets et des bibliothèques de classes préexistants. Ainsi il apprend très tôt à raisonner en termes d'interactions entre objets, plutôt qu'en termes de procédures, et cela l'autorise assez vite à tirer profit de concepts avancés, tels que l'héritage et le polymorphisme.

Nous avons par ailleurs accordé une place assez importante à la manipulation de différents types de structures de données, car nous estimons que c'est la réflexion sur les données qui doit rester la colonne vertébrale de tout développement logiciel.

Choix d'un premier langage de programmation

Il existe un très grand nombre de langages de programmation, chacun avec ses avantages et ses inconvénients. Il faut bien en choisir un. Lorsque nous avons commencé à réfléchir à cette question, durant notre préparation d'un curriculum pour la nouvelle option Sciences & Informatique, nous avions personnellement accumulé une assez longue expérience de la programmation sous *Visual Basic* (*Microsoft*) et sous *Clarion* (*Topspeed*). Nous avions également expérimenté quelque peu sous *Delphi* (*Borland*). Il était donc naturel que nous pensions d'abord exploiter l'un ou l'autre de ces langages. Si nous souhaitions les utiliser comme outils de base pour un apprentissage général de la programmation, ces langages présentaient toutefois deux gros inconvénients :

- Ils sont liés à des environnements de programmation (c'est-à-dire des logiciels) propriétaires. Cela signifiait donc, non seulement que l'institution scolaire désireuse de les utiliser devrait acheter une licence de ces logiciels pour chaque poste de travail (ce qui risquait de se révéler assez coûteux), mais surtout que les élèves souhaitant utiliser leurs compétences de programmation ailleurs qu'à l'école seraient implicitement forcés d'en acquérir eux aussi des licences, ce que nous ne pouvions pas accepter.
- Ce sont des langages spécifiquement liés au seul système d'exploitation *Windows*. Ils ne sont pas « portables » sur d'autres systèmes (*Unix*, *MacOS*, *etc.*). Cela ne cadrait pas avec notre projet pédagogique qui ambitionne d'inculquer une formation générale (et donc diversifiée) dans laquelle les invariants de l'informatique seraient autant que possible mis en évidence.

Nous avons alors décidé d'examiner l'offre alternative, c'est-à-dire celle qui est proposée gratuitement dans la mouvance de l'informatique libre[1]. Ce que nous avons trouvé nous a enthousiasmés : non seulement il existe dans le monde de *l'Open Source* des interpréteurs et des compilateurs gratuits pour toute une série de langages, mais surtout ces langages sont modernes, performants, portables (c'est-à-dire utilisables sur différents systèmes d'exploitation tels que *Windows*, *Linux*, *MacOS* ...), et fort bien documentés.

Le langage dominant y est sans conteste *C/C++*. Ce langage s'impose comme une référence absolue, et tout informaticien sérieux doit s'y frotter tôt ou tard. Il est malheureusement très rébarbatif et compliqué, trop proche de la machine. Sa syntaxe est peu lisible et fort contraignante. La mise au

[1] Un logiciel libre (*Free Software*) est avant tout un logiciel dont le code source est accessible à tous (*Open source*). Souvent gratuit (ou presque), copiable et modifiable librement au gré de son acquéreur, il est généralement le produit de la collaboration bénévole de centaines de développeurs enthousiastes dispersés dans le monde entier. Son code source étant "épluché" par de très nombreux spécialistes (étudiants et professeurs universitaires), un logiciel libre se caractérise la plupart du temps par un très haut niveau de qualité technique. Le plus célèbre des logiciels libres est le système d'exploitation ***GNU/Linux***, dont la popularité ne cesse de s'accroître de jour en jour.

point d'un gros logiciel écrit en *C/C++* est longue et pénible. (Les mêmes remarques valent aussi dans une large mesure pour le langage *Java*).

D'autre part, la pratique moderne de ce langage fait abondamment appel à des générateurs d'applications et autres outils d'assistance très élaborés tels *C++Builder*, *Kdevelop*, etc. Ces environnements de programmation peuvent certainement se révéler très efficaces entre les mains de programmeurs expérimentés, mais ils proposent d'emblée beaucoup trop d'outils complexes, et ils présupposent de la part de l'utilisateur des connaissances qu'un débutant ne maîtrise évidemment pas encore. Ce seront donc aux yeux de celui-ci de véritables « usines à gaz » qui risquent de lui masquer les mécanismes de base du langage lui-même. Nous laisserons donc le *C/C++* pour plus tard.

Pour nos débuts dans l'étude de la programmation, il nous semble préférable d'utiliser un langage de plus haut niveau, moins contraignant, à la syntaxe plus lisible. Après avoir successivement examiné et expérimenté quelque peu les langages *Perl* et *Tcl/Tk* , nous avons finalement décidé d'adopter Python, langage très moderne à la popularité grandissante.

Présentation du langage Python

> *Ce texte de Stéfane Fermigier est extrait d'un article paru dans le magazine Programmez! en décembre 1998. Il est également disponible sur http://www.linux-center.org/articles/9812/python.html). Stéfane Fermigier est le co-fondateur de l'AFUL (Association Francophone des Utilisateurs de Linux et des logiciels libres).*

Python est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est développé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles.

Caractéristiques du langage

Détaillons un peu les principales caractéristiques de Python, plus précisément, du langage et de ses deux implantations actuelles:

- Python est **portable**, non seulement sur les différentes variantes d'*Unix*, mais aussi sur les OS propriétaires: *MacOS*, *BeOS*, *NeXTStep*, *MS-DOS* et les différentes variantes de *Windows*. Un nouveau compilateur, baptisé *JPython*, est écrit en Java et génère du *bytecode* Java.
- Python est **gratuit**, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des **scripts** d'une dizaine de lignes qu'à des **projets complexes** de plusieurs dizaines de milliers de lignes.
- La **syntaxe** de Python est **très simple** et, combinée à des **types de données évolués** (listes, dictionnaires,...), conduit à des programmes à la fois très compacts et très lisibles. A fonctionnalités égales, un programme Python (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme C ou C++ (ou même Java) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
- Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur, par un mécanisme de **comptage de références** (proche, mais différent, d'un *garbage collector*).
- Il n'y a **pas de pointeurs** explicites en Python.

- Python est (optionnellement) **multi-threadé**.
- Python est **orienté-objet**. Il supporte **l'héritage multiple** et **la surcharge des opérateurs**. Dans son modèle objets, et en reprenant la terminologie de C++, toutes les méthodes sont virtuelles.
- Python intègre, comme Java ou les versions récentes de C++, un système **d'exceptions**, qui permettent de simplifier considérablement la gestion des erreurs.
- Python est **dynamique** (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions Python), **orthogonal** (un petit nombre de concepts suffit à engendrer des constructions très riches), **réflectif** (il supporte la métaprogrammation, par exemple la capacité pour un objet de se rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution) et **introspectif** (un grand nombre d'outils de développement, comme le *debugger* ou le *profiler*, sont implantés en Python lui-même).
- Comme *Scheme* ou *SmallTalk*, Python est dynamiquement typé. Tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- Python possède actuellement deux implémentations. L'une, **interprétée**, dans laquelle les programmes Python sont compilés en instructions portables, puis exécutés par une machine virtuelle (comme pour Java, avec une différence importante: Java étant statiquement typé, il est beaucoup plus facile d'accélérer l'exécution d'un programme Java que d'un programme Python). L'autre génère directement du *bytecode* Java.
- Python est **extensible** : comme *Tcl* ou *Guile*, on peut facilement l'interfacer avec des bibliothèques C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- La **bibliothèque standard** de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standards (fichiers, *pipes*, signaux, sockets, threads...), protocoles Internet (Web, News, FTP, CGI, HTML...), persistance et bases de données, interfaces graphiques.
- Python est un langage qui **continue à évoluer**, soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.
- Enfin, Python est un langage de choix pour traiter le XML.

Pour le professeur qui souhaite utiliser cet ouvrage comme support de cours

Nous souhaitons avec ces notes ouvrir un maximum de portes. A notre niveau d'études, il nous paraît important de montrer que la programmation d'un ordinateur est un vaste univers de concepts et de méthodes, dans lequel chacun peut trouver son domaine de prédilection. Nous ne pensons pas que tous nos étudiants doivent apprendre exactement les mêmes choses. Nous voudrions plutôt qu'ils arrivent à développer chacun des compétences quelque peu différentes, qui leur permettent de se valoriser à leurs propres yeux ainsi qu'à ceux de leurs condisciples, et également d'apporter leur contribution spécifique lorsqu'on leur proposera de collaborer à des travaux d'envergure.

De toute manière, notre préoccupation primordiale doit être d'arriver à susciter l'intérêt, ce qui est loin d'être acquis d'avance pour un sujet aussi ardu que la programmation d'un ordinateur. Nous ne voulons pas feindre de croire que nos jeunes élèves vont se passionner d'emblée pour la construction de beaux algorithmes. Nous sommes plutôt convaincus qu'un certain intérêt ne pourra durablement s'installer qu'à partir du moment où ils commenceront à réaliser qu'ils sont devenus capables de développer un projet personnel original, dans une certaine autonomie.

Ce sont ces considérations qui nous ont amenés à développer une structure de cours que certains trouveront peut-être un peu chaotique. Le début s'inspire d'un texte américain disponible sous licence libre : « *How to think like a computer scientist* », par Allen Downey, Jeff Elkner and Chris Meyers (Voir : http://greenteapress.com/thinkpython/thinkCSpy/), mais nous l'avons progressivement éclaté pour y insérer toute une série d'éléments concernant la gestion des entrées/sorties, et en particulier l'interface graphique *Tkinter*. Nous souhaiterions en effet que les élèves puissent déjà réaliser l'une ou l'autre petite application graphique dès la fin de leur première année d'études.

Très concrètement, cela signifie que nous pensons pouvoir explorer les huit premiers chapitres de ces notes durant la première année de cours. Cela suppose que l'on aborde d'abord toute une série de concepts importants (types de données, variables, instructions de contrôle du flux, fonctions et boucles) d'une manière assez rapide, sans trop se préoccuper de ce que chaque concept soit parfaitement compris avant de passer au suivant, en essayant plutôt d'inculquer le goût de la recherche personnelle et de l'expérimentation. Il sera souvent plus efficace de réexpliquer les notions et les mécanismes essentiels en situation, dans des contextes variés.

Dans notre esprit, c'est surtout en seconde année que l'on cherchera à structurer les connaissances acquises, en les approfondissant. Les algorithmes seront davantage décortiqués et commentés. Les projets, cahiers des charges et méthodes d'analyse seront discutés en concertation. On exigera la tenue régulière d'un cahier de notes et la rédaction de rapports techniques pour certains travaux.

L'objectif ultime sera pour chaque élève de réaliser un projet de programmation original d'une certaine importance. On s'efforcera donc de boucler l'étude théorique des concepts essentiels suffisamment tôt dans l'année scolaire, afin que chacun puisse disposer du temps nécessaire.

Il faut bien comprendre que les nombreuses informations fournies dans ces notes concernant une série de domaines particuliers (gestion des interfaces graphiques, des communications, des bases de données, etc.) sont matières facultatives. Ce sont seulement une série de suggestions et de repères que nous avons inclus pour aider les étudiants à choisir et à commencer leur projet personnel de fin d'études. Nous ne cherchons en aucune manière à former des spécialistes d'un certain langage ou d'un certain domaine technique : nous voulons simplement donner un petit aperçu des immenses possibilités qui s'offrent à celui qui se donne la peine d'acquérir une compétence de programmeur.

Versions du langage

Python continue à évoluer, mais cette évolution ne vise qu'à améliorer ou perfectionner le produit. Vous n'aurez pas à modifier tous vos programmes afin de les adapter à une nouvelle version qui serait devenue incompatible avec les précédentes. Les exemples de ce livre ont été réalisés les uns après les autres sur une période de temps relativement longue : certains ont été développés sous Python 1.5.2, puis d'autres sous Python 1.6, Python 2.0, Python 2.1, Python 2.2 et enfin Python 2.3.

Tous continuent cependant à fonctionner sans problème sous les versions 2.4 et 2.5 apparues depuis, et ils continueront certainement à fonctionner sans modification majeure sur les versions futures.

Installez donc sur votre système la dernière version disponible, et amusez-vous bien !

Distribution de Python - Bibliographie

Les différentes versions de Python (pour *Windows*, *Unix*, etc.), son ***tutoriel*** original, son ***manuel de référence***, la ***documentation*** des bibliothèques de fonctions, etc. sont disponibles en téléchargement gratuit depuis l'internet, à partir du site web officiel : http://www.python.org

Il existe également de très bons ouvrages imprimés concernant Python. En langue française, vous pourrez très profitablement consulter les manuels ci-après :

- ***Programmation Python***, par Tarek Ziadé, Editions Eyrolles, Paris, 2006, 538 p., ISBN 2-212-11677-2. C'est l'un des premiers ouvrages édités directement en langue française sur le langage Python. Excellent. Une mine de renseignements essentielle si vous voulez acquérir les meilleures pratiques et vous démarquer des débutants.
- ***Au coeur de Python***, volumes 1 et 2, par Wesley J. Chun, traduction de Python core programming, 2d edition (Prentice Hall) par Marie-Cécile Baland, Anne Bohy et Luc Carité, Editions CampusPress, Paris, 2007, respectivement 645 et 385 p., ISBN 978-2-7440-2148-0 et 978-2-7440-2195-4. Cet ouvrage est un ouvrage de référence indispensable, très bien écrit.

D'autres excellents ouvrages en français étaient proposés par la succursale française de la maison d'éditions O'Reilly, laquelle a malheureusement disparu. En langue anglaise, le choix est évidemment beaucoup plus vaste. Nous apprécions personnellement beaucoup ***Python : How to program***, par Deitel, Liperi & Wiedermann, Prentice Hall, Upper Saddle River - NJ 07458, 2002, 1300 p., ISBN 0-13-092361-3 , très complet, très clair, agréable à lire et qui utilise une méthodologie éprouvée, et ***Learn to program using Python***, par Alan Gauld, Addison-Wesley, Reading, MA, 2001, 270 p., ISBN 0-201-70938-4 , qui est un très bon ouvrage pour débutants.

Pour aller plus loin, notamment dans l'utilisation de la bibliothèque graphique Tkinter, on pourra utilement consulter ***Python and Tkinter Programming***, par John E. Grayson, Manning publications co., Greenwich (USA), 2000, 658 p., ISBN 1-884777-81-3 , et surtout l'incontournable ***Programming Python*** (second edition) de Mark Lutz, Editions O'Reilly, 2001, 1255 p., ISBN 0-596-00085-5, qui est une extraordinaire mine de renseignements sur de multiples aspects de la programmation moderne (sur tous systèmes).

Si vous savez déjà bien programmer, et que vous souhaiter progresser encore en utilisant les concepts les plus avancés de l'algorithmique Pythonienne, procurez vous ***Python cookbook***, par Alex Martelli et David Ascher, Editions O'Reilly, 2002, 575 p., ISBN 0-596-00167-3 , dont les recettes sont savoureuses.

Exemples du livre

Le code source des exemples de ce livre peut être téléchargé à partir du site de l'auteur :

http://inforef.be/swi/python.htm , ou encore :

http://main.pythomium.net/download/cours_python.zip

Remerciements

Ce livre est pour une partie le résultat d'un travail personnel, mais pour une autre - bien plus importante - la compilation d'informations et d'idées mises à la disposition de tous par des professeurs et des chercheurs bénévoles. Comme déjà signalé plus haut, l'une de mes sources les plus importantes a été le cours de A.Downey, J.Elkner & C.Meyers : *How to think like a computer scientist*. Merci encore à ces professeurs enthousiastes. J'avoue aussi m'être largement inspiré du tutoriel original écrit par Guido van Rossum lui-même (l'auteur principal de Python), ainsi que d'exemples et de documents divers émanant de la (très active) communauté des utilisateurs de Python. Il ne m'est malheureusement pas possible de préciser davantage les références de tous ces textes, mais je voudrais que leurs auteurs soient assurés de toute ma reconnaissance.

Merci également à tous ceux qui œuvrent au développement de Python, de ses accessoires et de sa documentation, à commencer par Guido van Rossum, bien sûr, mais sans oublier non plus tous les autres (Il sont (mal)heureusement trop nombreux pour que je puisse les citer tous ici).

Merci encore à mes collègues Freddy Klich, Christine Ghiot et David Carrera, professeurs à l'Institut St. Jean-Berchmans de Liège, qui ont accepté de se lancer dans l'aventure de ce nouveau cours avec leurs élèves, et ont également suggéré de nombreuses améliorations. Un merci tout particulier à Christophe Morvan, professeur à l'IUT de Marne-la-Vallée, pour ses avis précieux et ses encouragements. Grand merci aussi à Florence Leroy, mon éditrice chez O'Reilly, qui a corrigé mes incohérences et mes belgicismes avec une compétence sans faille. Merci encore à mes partenaires actuels chez Eyrolles, Muriel Shan Sei Fan et Matthieu Montaudouin, qui ont efficacement pris en charge cette nouvelle édition.

Merci enfin à mon épouse Suzel, pour sa patience et sa compréhension.

1

Penser comme un programmeur

Nous allons introduire dans ce chapitre quelques concepts qu'il vous faut connaître avant de vous lancer dans l'apprentissage de la programmation. Nous avons volontairement limité nos explications afin de ne pas vous encombrer l'esprit. La programmation n'est pas vraiment difficile, mais elle exige de la méthode et une bonne dose de persévérance.

La démarche du programmeur

Le but de ce cours est de vous apprendre à penser et à réfléchir comme un analyste-programmeur. Ce mode de pensée combine des démarches intellectuelles complexes, similaires à celles qu'accomplissent les mathématiciens, les ingénieurs et les scientifiques.

Comme le mathématicien, l'analyste-programmeur utilise des langages formels pour décrire des raisonnements (ou algorithmes). Comme l'ingénieur, il conçoit des dispositifs, il assemble des composants pour réaliser des mécanismes et il évalue leurs performances. Comme le scientifique, il observe le comportement de systèmes complexes, il ébauche des hypothèses explicatives, il teste des prédictions.

> *L'activité essentielle d'un analyste-programmeur consiste à résoudre des problèmes.*

Il s'agit là d'une compétence de haut niveau, qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète.

La programmation d'un ordinateur consiste en effet à « expliquer » en détail à une machine ce qu'elle doit faire, en sachant d'emblée qu'elle ne peut pas véritablement « comprendre » un langage humain, mais seulement effectuer un traitement automatique sur des séquences de caractères.
Un programme n'est rien d'autre qu'une suite d'instructions, encodées en respectant de manière très stricte un ensemble de conventions fixées à l'avance que l'on appelle un langage informatique. La machine est ainsi pourvue d'un mécanisme qui décode ces instructions en associant à chaque « mot » du langage une action précise.

Vous allez donc apprendre à programmer, activité déjà intéressante en elle-même parce qu'elle contribue à développer votre intelligence. Mais vous serez aussi amené à utiliser la programmation pour réaliser des projets concrets, ce qui vous procurera certainement de très grandes satisfactions.

Langage machine, langage de programmation

A strictement parler, un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique du type « tout ou rien » et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, *en format binaire*. Cela est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.), mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

Le seul « langage » que l'ordinateur puisse véritablement « comprendre » est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0 (les "bits") souvent traités par groupes de 8 (les « octets »), 16, 32, ou même 64. Ce « langage machine » est évidemment presqu'incompréhensible pour nous. Pour « parler » à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous.

Ces systèmes de traduction automatique seront établis sur la base de toute une série de conventions, dont il existera évidemment de nombreuses variantes.

Le système de traduction proprement dit s'appellera *interpréteur* ou bien *compilateur*, suivant la méthode utilisée pour effectuer la traduction (voir ci-après). On appellera *langage de programmation* un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des « phrases » que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire).

Suivant son niveau d'abstraction, on pourra dire d'un langage qu'il est « de bas niveau » (ex : *Assembler*) ou « de haut niveau » (ex : *Pascal*, *Perl*, *Smalltalk*, *Clarion*, *Lisp*...). Un langage de bas niveau est constitué d'instructions très élémentaires, très « proches de la machine ». Un langage de haut niveau comporte des instructions plus abstraites ou, plus « puissantes ». Cela signifie que chacune de ces instructions pourra être traduite par l'interpréteur ou le compilateur en un grand nombre d'instructions machine élémentaires.

Le langage que vous allez apprendre en premier est *Python*. Il s'agit d'un langage de haut niveau, dont la traduction en codes binaires est complexe et prend donc toujours un certain temps. Cela pourrait paraître un inconvénient. En fait, les avantages que présentent les langages de haut niveau sont énormes : il est *beaucoup plus facile* d'écrire un programme dans un langage de haut niveau ; l'écriture du programme prend donc beaucoup moins de temps ; la probabilité d'y faire des fautes est nettement plus faible ; la maintenance (c'est-à-dire l'apport de modifications ultérieures) et la recherche des erreurs (les « bugs ») sont grandement facilitées. De plus, un programme écrit dans un langage de haut niveau sera souvent *portable*, c'est-à-dire que l'on pourra le faire fonctionner sans guère de modifications sur des machines ou des systèmes d'exploitation différents. Un programme écrit dans un langage de bas niveau ne peut jamais fonctionner que sur un seul type de machine : pour qu'une autre l'accepte, il faut le réécrire entièrement.

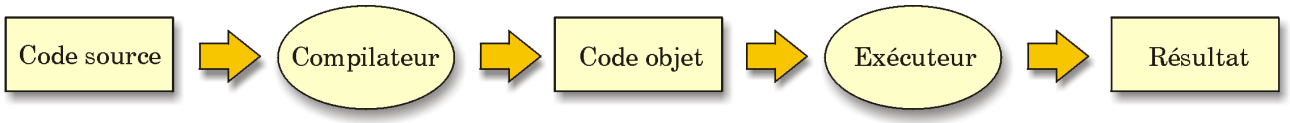
Compilation et interprétation

Le programme tel que nous l'écrivons à l'aide d'un logiciel éditeur (une sorte de traitement de texte spécialisé) sera appelé désormais programme source (ou code source). Comme déjà signalé plus haut, il existe deux techniques principales pour effectuer la traduction d'un tel programme source en code binaire exécutable par la machine : l'interprétation et la compilation.

- Dans la technique appelée *interprétation*, le logiciel interpréteur doit être utilisé chaque fois que l'on veut faire fonctionner le programme. Dans cette technique en effet, chaque ligne du programme source analysé est traduite au fur et à mesure en quelques instructions du langage machine, qui sont ensuite directement exécutées. Aucun programme objet n'est généré.

L'interpréteur lit le code source ... *... et le résultat apparaît sur l'écran.*

- La *compilation* consiste à traduire la totalité du texte source en une fois. Le logiciel compilateur lit toutes les lignes du programme source et produit une nouvelle suite de codes que l'on appelle *programme objet* (ou code objet). Celui-ci peut désormais être exécuté indépendamment du compilateur et être conservé tel quel dans un fichier (« fichier exécutable »).



Le compilateur lit le code source ... *... et produit un code objet (binaire).* *On exécute le code objet ...* *... et le résultat apparaît à l'écran.*

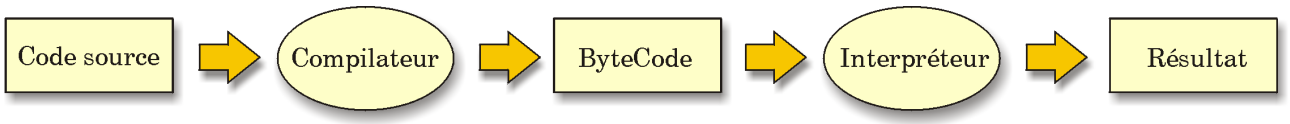
Chacune de ces deux techniques a ses avantages et ses inconvénients :

L'interprétation est idéale lorsque l'on est en phase d'apprentissage du langage, ou en cours d'expérimentation sur un projet. Avec cette technique, on peut en effet tester immédiatement toute modification apportée au programme source, sans passer par une phase de compilation qui demande toujours du temps.

Par contre, lorsqu'un projet comporte des fonctionnalités complexes qui doivent s'exécuter rapidement, la compilation est préférable : il est clair en effet qu'un programme compilé fonctionnera toujours nettement plus vite que son homologue interprété, puisque dans cette technique l'ordinateur n'a plus à (re)traduire chaque instruction en code binaire avant qu'elle puisse être exécutée.

Certains langages modernes tentent de combiner les deux techniques afin de retirer le meilleur de chacune. C'est le cas de Python et aussi de Java. Lorsque vous lui fournissez un programme source, Python commence par le compiler pour produire un code intermédiaire, similaire à un langage machine, que l'on appelle *bytecode*, lequel sera ensuite transmis à un interpréteur pour l'exécution finale. Du point de vue de l'ordinateur, le *bytecode* est très facile à interpréter en langage machine. Cette interprétation sera donc beaucoup plus rapide que celle d'un code source.

Les avantages de cette méthode sont appréciables :



Le compilateur Python lit le code source ... *... et produit un pseudo-code intermédiaire.* *L'interpréteur Python lit le pseudo-code ...* *... et le résultat apparaît à l'écran.*

- Le fait de disposer en permanence d'un interpréteur permet de tester immédiatement n'importe quel petit morceau de programme. On pourra donc vérifier le bon fonctionnement de chaque composant d'une application au fur et à mesure de sa construction.
- L'interprétation du *bytecode* compilé n'est pas aussi rapide que celle d'un véritable code binaire, mais elle est très satisfaisante pour de très nombreux programmes, y compris graphiques.
- Le *bytecode* est portable. Pour qu'un programme Python ou Java puisse s'exécuter sur différentes machines, il suffit de disposer pour chacune d'elles d'un interpréteur adapté.

Tout ceci peut vous paraître un peu compliqué, mais la bonne nouvelle est que tout ceci est pris en charge automatiquement par l'environnement de développement de Python. Il vous suffira d'entrer vos commandes au clavier, de frapper <Enter>, et Python se chargera de les compiler et de les interpréter pour vous.

Mise au point d'un programme & recherche des erreurs (« debug »)

La programmation est une démarche très complexe, et comme c'est le cas dans toute activité humaine, on y commet de nombreuses erreurs. Pour des raisons anecdotiques, les erreurs de programmation s'appellent des « *bugs* » (ou « bogues », en France)[2], et l'ensemble des techniques que l'on met en œuvre pour les détecter et les corriger s'appelle « *debug* » (ou « débogage »).

En fait, il peut exister dans un programme trois types d'erreurs assez différentes, et il convient que vous appreniez à bien les distinguer :

Erreurs de syntaxe

Python ne peut exécuter un programme que si sa syntaxe est parfaitement correcte. Dans le cas contraire, le processus s'arrête et vous obtenez un message d'erreur. Le terme syntaxe se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.

Tout langage comporte sa syntaxe. Dans la langue française, par exemple, une phrase doit toujours commencer par une majuscule et se terminer par un point. ainsi cette phrase comporte deux erreurs de syntaxe

Dans les textes ordinaires, la présence de quelques petites fautes de syntaxe par-ci par-là n'a généralement pas d'importance. Il peut même arriver (en poésie, par exemple), que des fautes de syntaxe soient commises volontairement. Cela n'empêche pas que l'on puisse comprendre le texte.

---

[2] "*bug*" est à l'origine un terme anglais servant à désigner de petits insectes gênants, tels les punaises. Les premiers ordinateurs fonctionnaient à l'aide de "lampes" radios qui nécessitaient des tensions électriques assez élevées. Il est arrivé à plusieurs reprises que des petits insectes s'introduisent dans cette circuiterie complexe et se fassent électrocuter, leurs cadavres calcinés provoquant alors des court-circuits et donc des pannes incompréhensibles.
Le mot français "*bogue*" a été choisi par homonymie approximative. Il désigne la coque épineuse de la châtaigne.

Dans un programme d'ordinateur, par contre, la moindre erreur de syntaxe produit invariablement un arrêt de fonctionnement (un « plantage ») ainsi que l'affichage d'un message d'erreur. Au cours des premières semaines de votre carrière de programmeur, vous passerez certainement pas mal de temps à rechercher vos erreurs de syntaxe. Avec de l'expérience, vous en commettrez beaucoup moins.

Gardez à l'esprit que les mots et les symboles utilisés n'ont aucune signification en eux-mêmes : ce ne sont que des suites de codes destinés à être convertis automatiquement en nombres binaires.
Par conséquent, il vous faudra être très attentifs à respecter scrupuleusement la syntaxe du langage.

Il est heureux que vous fassiez vos débuts en programmation avec un langage interprété tel que Python. La recherche des erreurs y est facile et rapide. Avec les langages compilés (tel C++), il vous faudrait recompiler l'intégralité du programme après chaque modification, aussi minime soit-elle.

Erreurs sémantiques

Le second type d'erreur est l'erreur sémantique ou erreur de logique. S'il existe une erreur de ce type dans un de vos programmes, celui-ci s'exécute parfaitement, en ce sens que vous n'obtenez aucun message d'erreur, mais le résultat n'est pas celui que vous attendiez : vous obtenez autre chose.

En réalité, le programme fait exactement ce que vous lui avez dit de faire. Le problème est que ce que vous lui avez dit de faire ne correspond pas à ce que vous vouliez qu'il fasse. La séquence d'instructions de votre programme ne correspond pas à l'objectif poursuivi. La sémantique (la logique) est incorrecte.

Rechercher des fautes de logique peut être une tâche ardue. Il faut analyser ce qui sort de la machine et tâcher de se représenter une par une les opérations qu'elle a effectuées, à la suite de chaque instruction.

Erreurs à l'exécution

Le troisième type d'erreur est l'erreur en cours d'exécution (*Run-time error*), qui apparaît seulement lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus). Ces erreurs sont également appelées des ***exceptions***, parce qu'elles indiquent généralement que quelque chose d'exceptionnel s'est produit (et qui n'avait pas été prévu). Vous rencontrerez davantage ce type d'erreur lorsque vous programmerez des projets de plus en plus volumineux.

Recherche des erreurs et expérimentation

L'une des compétences les plus importantes à acquérir au cours de votre apprentissage est celle qui consiste à *déboguer* efficacement un programme. Il s'agit d'une activité intellectuelle parfois énervante mais toujours très riche, dans laquelle il faut faire montre de beaucoup de perspicacité.

Ce travail ressemble par bien des aspects à une enquête policière. Vous examinez un ensemble de faits, et vous devez émettre des hypothèses explicatives pour reconstituer les processus et les événements qui ont logiquement entraîné les résultats que vous constatez.

Cette activité s'apparente aussi au travail expérimental en sciences. Vous vous faites une première idée de ce qui ne va pas, vous modifiez votre programme et vous essayez à nouveau. Vous

avez émis une hypothèse, qui vous permet de prédire ce que devra donner la modification. Si la prédiction se vérifie, alors vous avez progressé d'un pas sur la voie d'un programme qui fonctionne. Si la prédiction se révèle fausse, alors il vous faut émettre une nouvelle hypothèse. Comme l'a bien dit Sherlock Holmes : « Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité » (A. Conan Doyle, *Le signe des quatre*).

Pour certaines personnes, « programmer » et « déboguer » signifient exactement la même chose. Ce qu'elles veulent dire par là est que l'activité de programmation consiste en fait à modifier, à corriger sans cesse un même programme, jusqu'à ce qu'il se comporte finalement comme vous le vouliez. L'idée est que la construction d'un programme commence toujours par une ébauche qui fait déjà quelque chose (et qui est donc déjà déboguée), à laquelle on ajoute couche par couche de petites modifications, en corrigeant au fur et à mesure les erreurs, afin d'avoir de toute façon à chaque étape du processus un programme qui fonctionne.

Par exemple, vous savez que Linux est un système d'exploitation (et donc un gros logiciel) qui comporte des milliers de lignes de code. Au départ, cependant, cela a commencé par un petit programme simple que Linus Torvalds avait développé pour tester les particularités du processeur *Intel 80386*. Suivant Larry GreenField (« *The Linux user's guide* », beta version 1) : « L'un des premiers projets de Linus était un programme destiné à convertir une chaîne de caractères AAAA en BBBB. C'est cela qui plus tard finit par devenir Linux ! ».

Ce qui précède ne signifie pas que nous voulions vous pousser à programmer par approximations successives, à partir d'une vague idée. Lorsque vous démarrerez un projet de programmation d'une certaine importance, il faudra au contraire vous efforcer d'établir le mieux possible un *cahier des charges* détaillé, lequel s'appuiera sur un plan solidement construit pour l'application envisagée.

Diverses méthodes existent pour effectuer cette tâche d'analyse, mais leur étude sort du cadre de ces notes. Nous vous présenterons cependant plus loin (voir chapitre 15) quelques idées de base.

Langages naturels et langages formels

Les *langages naturels* sont ceux que les êtres humains utilisent pour communiquer. Ces langages n'ont pas été mis au point délibérément (encore que certaines instances tâchent d'y mettre un peu d'ordre) : ils évoluent naturellement.

Les *langages formels* sont des langages développés en vue d'applications spécifiques. Ainsi par exemple, le système de notation utilisé par les mathématiciens est un langage formel particulièrement efficace pour représenter les relations entre nombres et grandeurs diverses. Les chimistes utilisent un langage formel pour représenter la structure des molécules, etc.

> *Les langages de programmation sont des langages formels qui ont été développés pour décrire des algorithmes et des structures de données.*

Comme on l'a déjà signalé plus haut, les langages formels sont dotés d'une syntaxe qui obéit à des règles très strictes. Par exemple, **3+3=6** est une représentation mathématique correcte, alors que **$3=+6** ne l'est pas. De même, la formule chimique H_2O est correcte, mais non Zq_3G_2

Les règles de syntaxe s'appliquent non seulement aux symboles du langage (par exemple, le symbole chimique **Zq** est illégal parce qu'il ne correspond à aucun élément), mais aussi à la manière de les combiner. Ainsi l'équation mathématique **6+=+/5-** ne contient que des symboles parfaitement autorisés, mais leur arrangement incorrect ne signifie rien du tout.

Lorsque vous lisez une phrase quelconque, vous devez arriver à vous représenter la structure logique de la phrase (même si vous faites cela inconsciemment la plupart du temps). Par exemple, lorsque vous lisez la phrase « la pièce est tombée », vous comprenez que « la pièce » en est le sujet et « est tombée » le verbe. L'analyse vous permet de comprendre la signification, la logique de la phrase (sa sémantique). D'une manière analogue, l'interpréteur Python devra analyser la structure de votre programme source pour en extraire la signification.

Les langages naturels et formels ont donc beaucoup de caractéristiques communes (des symboles, une syntaxe, une sémantique), mais ils présentent aussi des différences très importantes :

Ambiguïté.

Les langages naturels sont pleins d'ambiguïtés, que nous pouvons lever dans la plupart des cas en nous aidant du contexte. Par exemple, nous attribuons tout naturellement une signification différente au mot vaisseau, suivant que nous le trouvons dans une phrase qui traite de circulation sanguine ou de navigation à voiles. Dans un langage formel, il ne peut pas y avoir d'ambiguïté. Chaque instruction possède une seule signification, indépendante du contexte.

Redondance.

Pour compenser toutes ces ambiguïtés et aussi de nombreuses erreurs ou pertes dans la transmission de l'information, les langages naturels emploient beaucoup la redondance (dans nos phrases, nous répétons plusieurs fois la même chose sous des formes différentes, pour être sûrs de bien nous faire comprendre). Les langages formels sont beaucoup plus concis.

Littéralité.

Les langages naturels sont truffés d'images et de métaphores. Si je dis « la pièce est tombée ! » dans un certain contexte, il se peut qu'il ne s'agisse en fait ni d'une véritable pièce, ni de la chute de quoi que ce soit. Dans un langage formel, par contre, les expressions doivent être prises pour ce qu'elles sont, *au pied de la lettre.*

Habitués comme nous le sommes à utiliser des langages naturels, nous avons souvent bien du mal à nous adapter aux règles rigoureuses des langages formels. C'est l'une des difficultés que vous devrez surmonter pour arriver à penser comme un analyste-programmeur efficace.

Pour bien nous faire comprendre, comparons encore différents types de textes :

Un texte poétique :

Les mots y sont utilisés autant pour leur musicalité que pour leur signification, et l'effet recherché est surtout émotionnel. Les métaphores et les ambiguïtés y règnent en maîtres.

Un texte en prose :

La signification littérale des mots y est plus importante, et les phrases sont structurées de manière à lever les ambiguïtés, mais sans y parvenir toujours complètement. Les redondances sont souvent nécessaires.

Un programme d'ordinateur :

La signification du texte est unique et littérale. Elle peut être comprise entièrement par la seule analyse des symboles et de la structure. On peut donc automatiser cette analyse.

Pour conclure, voici quelques suggestions concernant la manière de lire un programme d'ordinateur (ou tout autre texte écrit en langage formel).

Premièrement, gardez à l'esprit que les langages formels sont beaucoup plus denses que les langages naturels, ce qui signifie qu'il faut davantage de temps pour les lire. De plus, la structure y est très importante. Aussi, ce n'est généralement pas une bonne idée que d'essayer de lire un programme d'une traite, du début à la fin. Au lieu de cela, entraînez-vous à analyser le programme dans votre tête, en identifiant les symboles et en interprétant la structure.

Finalement, souvenez-vous que tous les détails ont de l'importance. Il faudra en particulier faire très attention à la *casse* (c'est-à-dire l'emploi des majuscules et des minuscules) et à la *ponctuation*. Toute erreur à ce niveau (même minime en apparence, tel l'oubli d'une virgule, par exemple) peut modifier considérablement la signification du code, et donc le déroulement du programme.

2

Premiers pas

Il est temps de nous mettre au travail. Plus exactement, nous allons prendre le commandement de l'ordinateur, et lui demander de travailler à notre place, en lui donnant l'ordre, par exemple, d'effectuer une addition et d'en afficher le résultat.

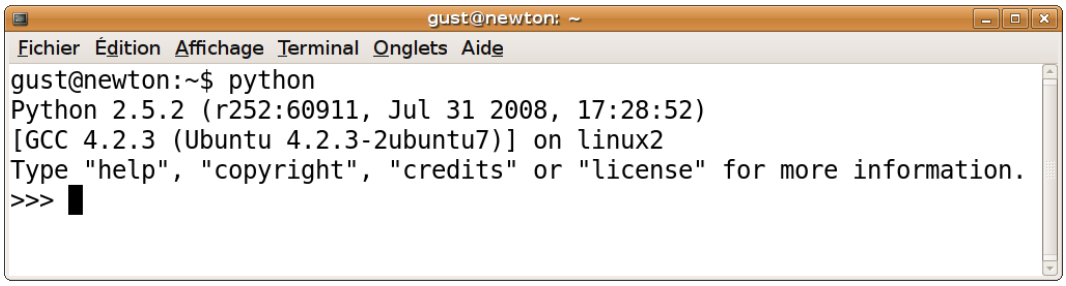
Pour cela, nous allons devoir lui transmettre des « instructions », et également lui indiquer les « données » auxquelles nous voulons appliquer ces instructions.

Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes.
Vous allez d'abord l'utiliser en mode interactif, c'est-à-dire d'une manière telle que vous pourrez dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

L'interpréteur peut être lancé directement depuis la ligne de commande (dans un « shell » *Linux*, ou bien dans une fenêtre *DOS* sous *Windows*) : il suffit d'y taper la commande "python" (en supposant que le logiciel lui-même ait été correctement installé).

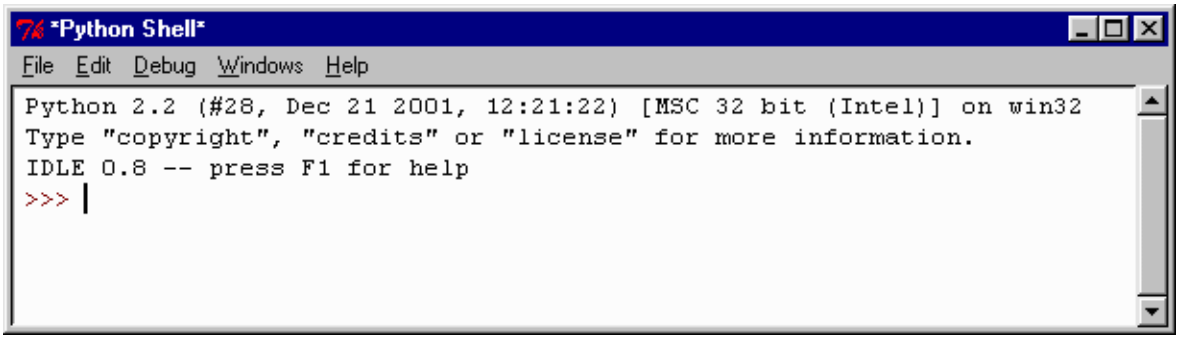
Si vous utilisez une interface graphique telle que *Windows*, *Gnome*, *WindowMaker* ou *KDE*, vous préférerez vraisemblablement travailler dans une « fenêtre de terminal », ou encore dans un environnement de travail spécialisé tel que *IDLE*. Voici par exemple ce qui apparaît dans une fenêtre de terminal *Gnome* (sous *Ubuntu Linux*)[3] :



[3] Sous *Windows*, vous aurez surtout le choix entre l'environnement *IDLE* développé par Guido Van Rossum, auquel nous donnons nous-même la préférence, et *PythonWin*, une interface de développement développée par Mark Hammond. D'autres environnements de travail plus sophistiqués existent aussi, tels l'excellent *Boa Constructor* par exemple (qui fonctionne de façon très similaire à *Delphi*), mais nous estimons qu'ils ne conviennent guère aux débutants. Pour tout renseignement complémentaire, veuillez consulter le site Web de Python.

Sous *Linux*, nous préférons personnellement travailler dans une simple fenêtre de terminal pour lancer l'interpréteur Python ou l'exécution des scripts, et à faire appel à un éditeur de texte ordinaire tel que *Gedit*, *Kate*, ou un peu plus spécialisé comme *SciTE*, *DrPython* ou *Geany* pour l'édition de ces derniers (voir Annexe, page 315).

Avec *IDLE* sous *Windows*, votre environnement de travail ressemblera à celui-ci :



Les trois caractères « supérieur à » constituent le signal d'invite, ou *prompt principal*, lequel vous indique que Python est prêt à exécuter une commande.

Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (Prenez l'habitude d'utiliser votre cahier d'exercices pour noter les résultats qui apparaissent à l'écran) :

```
>>> 5+3

>>> 2 - 9            # les espaces sont optionnels

>>> 7 + 3 * 4        # la hiérarchie des opérations mathématiques
                     # est-elle respectée ?
>>> (7+3)*4

>>> 20 / 3           # surprise !!!
```

Comme vous pouvez le constater, les opérateurs arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement +, -, * et /. Les parenthèses sont fonctionnelles.

Par défaut, la division est cependant une *division entière*, ce qui signifie que si on lui fournit des arguments qui sont des nombres entiers, le résultat de la division est lui-même un entier (tronqué), comme dans le dernier exemple ci-dessus. Si vous voulez qu'un argument soit compris par Python comme étant un nombre réel, il faut le lui faire savoir, en fournissant au moins un point décimal[4].

Essayez par exemple (comparez les résultats avec celui obtenu à l'exercice précédent[5]) :

```
>>> 20.0 / 3

>>> 8./5
```

Si une opération est effectuée avec des arguments de types mélangés (entiers et réels), Python convertit automatiquement les opérandes en réels avant d'effectuer l'opération. Essayez :

```
>>> 4 * 2.5 / 3.3
```

[4] Dans tous les langages de programmation, les conventions mathématiques de base sont celles en vigueur dans les pays anglophones : le séparateur décimal sera donc toujours un point, et non une virgule comme chez nous.
Dans le monde de l'informatique, les nombres réels sont souvent désignés comme des nombres "à virgule flottante", ou encore des nombres "de type ***float***".

[5] Ce comportement par défaut de l'opérateur de division / sous Python sera probablement modifié dans les versions futures du langage. Il est proposé en effet que le comportement par défaut de cet opérateur consistera à l'avenir à effectuer plutôt une division réelle, alors que la division entière ne sera plus obtenue qu'à l'aide du nouvel opérateur // (qui fonctionne déjà).. De ce fait, si vous souhaitez ***effectivement*** obtenir une division entière, il vous est conseillé d'utiliser de préférence ce dernier..

Données et variables

Nous aurons l'occasion de détailler plus loin les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept de grande importance :

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des *données*. Ces données peuvent être très diverses (tout ce qui est *numérisable*, en fait[6]), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à *une suite finie de nombres binaires*.

Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de *variables* de différents types.

Une variable apparaît dans un langage de programmation sous un *nom de variable* à peu près quelconque (voir ci-après), mais pour l'ordinateur il s'agit d'une *référence* désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.

A cet emplacement est stocké une *valeur* bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé. Cela peut être en fait à peu près n'importe quel « objet » susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc.

Pour distinguer les uns des autres ces divers contenus possibles, le langage de programmation fait usage de différents types de variables. (le type *entier*, le type *réel*, le type *chaîne de caractères*, le type *liste*, etc.). Nous allons expliquer tout cela dans les pages suivantes.

Noms de variables et mots réservés

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que *altitude*, *altit* ou *alt* conviennent mieux que x pour exprimer une altitude.

> *Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.*

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que $, #, @, etc. sont interdits, à l'exception du caractère _ (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués).
  Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre[7]). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez

---

[6] *Que peut-on numériser au juste ?* Voilà une question très importante, qu'il vous faudra débattre dans votre cours d'informatique générale.

[7] Les noms commençant par une majuscule ne sont pas interdits, mais l'usage veut qu'on les réserve plutôt aux variables qui désignent des ***classes*** (le concept de classe sera abordé plus loin dans ces notes). Il arrive aussi que l'on écrive

les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatieres*, par exemple.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les 30 « mots réservés » ci-dessous (ils sont utilisés par le langage lui-même) :

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

Affectation (ou assignation)

Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons en définir une et lui *affecter* une valeur. Les termes « affecter une valeur » ou « assigner une valeur » à une variable sont équivalents. Ils désignent l'opération par laquelle on établit un lien entre le nom de la variable et sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe *égale*[8] :

```
>>> n = 7                       # donner à n la valeur 7
>>> msg = "Quoi de neuf ?"      # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159                # assigner sa valeur à la variable pi
```

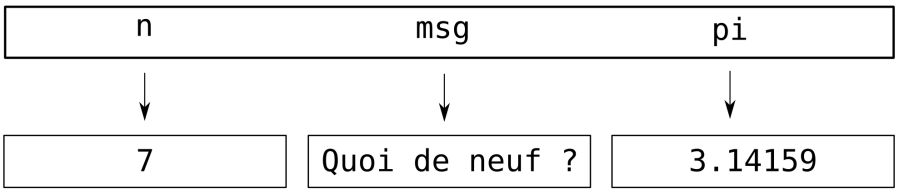
Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

- trois noms de variables, à savoir **n**, **msg** et **pi**
- trois séquences d'octets, où sont encodées le nombre entier **7**, la chaîne de caractères **Quoi de neuf ?** et le nombre réel **3,14159**.

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un **nom de variable** ;
- lui attribuer un **type** bien déterminé (ce point sera explicité à la page suivante) ;
- créer et mémoriser une **valeur** particulière ;
- établir un **lien** (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

On peut mieux se représenter tout cela par un diagramme d'état tel que celui-ci :



entièrement en majuscules certaines variables que l'on souhaite traiter comme des pseudo-constantes (c'est-à-dire des variables que l'on évite de modifier au cours du programme).

[8] Il faut bien comprendre qu'il ne s'agit en aucune façon d'une égalité, et que l'on aurait très bien pu choisir un autre symbolisme, tel que **n ← 7** par exemple, comme on le fait souvent dans certains pseudo-langages servant à décrire des algorithmes, pour bien montrer qu'il s'agit de relier un contenu (la valeur 7) à un contenant (la variable n).

Les trois noms de variables sont des *références*, mémorisées dans une zone particulière de la mémoire que l'on appelle *espace de noms*, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres. Nous aurons l'occasion de préciser ce concept plus loin dans ces pages.

Afficher la valeur d'une variable

A la suite de l'exercice ci-dessus, nous disposons donc des trois variables **n**, **msg** et **pi**.
Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis <*Enter*>. Python répond en affichant la valeur correspondante :

```
>>> n
7
>>> msg
"Quoi de neuf ?"
>>> pi
3.14159
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. A l'intérieur d'un programme, vous utiliserez toujours l'instruction **print** :

```
>>> print msg
Quoi de neuf ?
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. L'instruction **print** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des guillemets (afin de vous rappeler le type de la variable : nous y reviendrons).

Veuillez aussi ignorer pour l'instant les bizarreries qui se produisent si vous modifiez l'exercice précédent en choisissant une chaîne de caractères qui contient des lettres accentuées, comme dans l'exemple ci-dessous :

```
>>> msg = "Mon prénom est Chimène"
>>> msg
'Mon pr\xe9nom est Chim\xe8ne'
```

Les lettres accentuées du français (de même que les lettres d'autres alphabets, tels le grec, l'arabe, le cyrillique, l'hébreu, etc.), ne font pas partie du jeu de caractères standard d'un ordinateur, lequel se limite aux chiffres arabes, aux lettres majuscules et minuscules de l'alphabet latin non accentué, plus quelques signes typographiques usuels (?/+; etc.). Leur utilisation pose donc quelques problèmes particuliers, qui n'ont malheureusement pas toujours été solutionnés de la même manière au cours de l'évolution des systèmes informatiques (Il se peut par exemple que le résultat du test présenté ci-dessus soit déjà différent sur votre propre ordinateur !). Nous examinerons ces questions plus loin (voir p. *****). Pour l'instant nous devons d'abord affiner notre compréhension des variables.

Typage des variables

Sous Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'assigner une valeur à un nom de variable pour que celle-ci soit *automatiquement créée avec le type qui correspond au mieux à la valeur fournie*. Dans l'exercice précédent, par exemple, les variables **n**, **msg** et **pi** ont été créées automatiquement chacune avec un type différent (« nombre entier » pour **n**, « chaîne de caractères » pour **msg**, « nombre à virgule flottante » (ou « *float* », en anglais) pour **pi**).

Ceci constitue une particularité intéressante de Python, qui le rattache à une famille particulière de langages où l'on trouve aussi par exemple *Lisp*, *Scheme*, et quelques autres. On dira à ce sujet que *le typage des variables sous Python est un typage dynamique*, par opposition au *typage statique* qui est de règle par exemple en C++ ou en Java. Dans ces langages, il faut toujours - par des instructions distinctes - d'abord déclarer (définir) le nom et le type des variables, et ensuite seulement leur assigner un contenu, lequel doit bien entendu être compatible avec le type déclaré.

Le typage statique est préférable dans le cas des langages compilés, parce qu'il permet d'optimiser l'opération de compilation (dont le résultat est un code binaire « figé »).

Le typage dynamique quant à lui permet d'écrire plus aisément des constructions logiques de niveau élevé (métaprogrammation, réflexivité), en particulier dans le contexte de la programmation orientée objet (polymorphisme). Il facilite également l'utilisation de structures de données très riches telles que les listes et les dictionnaires.

Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
>>> x = y = 7
>>> x
7
>>> y
7
```

On peut aussi effectuer des *affectations parallèles* à l'aide d'un seul opérateur :

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs **4** et **8,33**.

> *Les francophones que nous sommes avons pour habitude d'utiliser la virgule comme séparateur décimal, alors que les langages de programmation utilisent toujours la convention en vigueur dans les pays de langue anglaise, c'est-à-dire le point décimal. La virgule, quant à elle, est très généralement utilisée pour séparer différents éléments (arguments, etc.) comme on le voit dans notre exemple, pour les variables elles-mêmes ainsi que pour les valeurs qu'on leur attribue.*

2 Exercices

2.1 Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :

```
>>> largeur = 20
>>> hauteur = 5 * 9.3
>>> largeur * hauteur
930
```

2.2 Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c.
Effectuez l'opération a - b/c . Le résultat est-il mathématiquement correct ?
Si ce n'est pas le cas, comment devez-vous procéder pour qu'il le soit ?

Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent, en les combinant avec des *opérateurs* pour former des *expressions*. Exemple :

```
a, b = 7.3, 12
y = 3*a + b/5
```

Dans cet exemple, nous commençons par affecter aux variables **a** et **b** les valeurs **7,3** et **12**.
Comme déjà expliqué précédemment, Python assigne automatiquement le type « réel » à la variable **a**, et le type « entier » à la variable **b**.

La seconde ligne de l'exemple consiste à affecter à une nouvelle variable **y** le résultat d'une expression qui combine les *opérateurs* ***** , **+** et **/** avec les *opérandes* **a**, **b**, **3** et **5**. Les opérateurs sont les symboles spéciaux utilisés pour représenter des opérations mathématiques simples, telles l'addition ou la multiplication. Les opérandes sont les valeurs combinées à l'aide des opérateurs.

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. À cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a dans l'expression. Dans l'exemple ci-dessus, y sera du type réel, parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un réel.

Les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. Il faut leur ajouter l'opérateur ****** pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc. Nous reparlerons de tout cela plus loin.

Signalons au passage la disponibilité de l'opérateur *modulo*, représenté par le symbole **%**.
Cet opérateur fournit *le reste de la division entière* d'un nombre par un autre. Essayez par exemple :

```
>>> 10 % 3       (et prenez note de ce qui se passe !)
>>> 10 % 5
```

Cet opérateur vous sera très utile plus loin, notamment pour tester si un nombre **a** est divisible par un nombre **b**. Il suffira en effet de vérifier que **a % b** donne un résultat égal à zéro.

Exercice :

2.3 Testez les lignes d'instructions suivantes. Décrivez dans votre cahier ce qui se passe :

```
>>>  r , pi = 12, 3.14159
>>>  s = pi * r**2
>>>  print s
>>>  print type(r), type(pi), type(s)
```

Quelle est, à votre avis, l'utilité de la *fonction* **type()** ?
(Note : les *fonctions* seront décrites en détail, plus loin dans ce cours).

Priorité des opérations

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de *règles de priorité*. Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique. Vous pouvez les mémoriser aisément à l'aide d'un « truc » mnémotechnique, l'acronyme *PEMDAS* :

- *P* pour *parenthèses*. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez.
  Ainsi `2*(3-1) = 4`, et `(1+1)**(5-2) = 8`.
- *E* pour *exposants*. Les exposants sont évalués ensuite, avant les autres opérations.
  Ainsi `2**1+1 = 3` (et non 4), et `3*1**10 = 3` (et non 59049 !).
- *M* et *D* pour *multiplication* et *division*, qui ont la même priorité. Elles sont évaluées avant *l'addition A* et *la soustraction S*, lesquelles sont donc effectuées en dernier lieu.
  Ainsi `2*3-1 = 5` (plutôt que 4), et `2/3-1 = -1` (Rappelez-vous aussi, pour ce dernier exemple,que par défaut Python effectue une division entière).
- Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite.
  Ainsi dans l'expression `59*100/60`, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer 5900/60, ce qui donne 98. Si la division était effectuée en premier, le résultat serait 59 (rappelez-vous ici encore qu'il s'agit d'une division entière).

Composition

Jusqu'ici nous avons examiné les différents éléments d'un langage de programmation, à savoir : les *variables*, les *expressions* et les *instructions*, mais sans traiter de la manière dont nous pouvons les combiner les unes avec les autres.

Or l'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de construire des instructions complexes par assemblage de fragments divers. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
>>> print 17 + 3
>>> 20
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
>>> h, m, s = 15, 27, 34
>>> print "nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s
```

Attention, cependant : il y a une limite à ce que vous pouvez combiner ainsi :

Ce que vous placez à la gauche du signe égale dans une expression doit toujours être une variable, et non une expression. Cela provient du fait que le signe égale n'a pas ici la même signification qu'en mathématique : comme nous l'avons déjà signalé, il s'agit d'un symbole d'affectation (nous plaçons un certain contenu dans une variable) et non un symbole d'égalité. Le symbole d'égalité (dans un test conditionnel, par exemple) sera évoqué un peu plus loin.

Ainsi par exemple, l'instruction `m + 1 = b` est tout à fait **illégale**.

Par contre, écrire `a = a + 1` est inacceptable en mathématique, alors que cette forme d'écriture est très fréquente en programmation. L'instruction `a = a + 1` signifie en l'occurrence « augmenter la valeur de la variable a d'une unité » (ou encore : « incrémenter a »).

Nous aurons l'occasion de revenir bientôt sur ce sujet. Mais auparavant, il nous faut encore aborder un autre concept de grande importance.

3

Contrôle du flux d'exécution

Dans notre premier chapitre, nous avons vu que l'activité essentielle d'un analyste-programmeur est la résolution de problèmes. Or, pour résoudre un problème informatique, il faut toujours effectuer une série d'*actions* dans un certain *ordre*. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un *algorithme*.

Le « chemin » suivi par Python à travers un programme est appelé un *flux d'exécution*, et les constructions qui le modifient sont appelées des *instructions de contrôle de flux*.

Les *structures de contrôle* sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois : la *séquence*[9] et la *sélection*, que nous allons décrire dans ce chapitre, et la *répétition* que nous aborderons au chapitre suivant.

Séquence d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, *dans l'ordre où elles ont été écrites à l'intérieur du script*.

Cette affirmation peut vous paraître banale et évidente à première vue. L'expérience montre cependant qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres. Par exemple, dans la séquence d'instructions suivantes :

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
>>> print a, b
```

Vous obtiendrez un résultat contraire si vous intervertissez les 2e et 3e lignes.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une *instruction conditionnelle* comme l'instruction **if** décrite ci-après (nous en rencontrerons d'autres plus loin, notamment à propos des boucles de répétition). Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

Sélection ou exécution conditionnelle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des

[9] Tel qu'il est utilisé ici, le terme de ***séquence*** désigne donc une série d'instructions qui se suivent. Nous préférerons dans la suite de cet ouvrage réserver ce terme à un concept Python précis, lequel englobe les ***chaînes de caractères***, les ***tuples*** et les ***listes*** (voir plus loin).

circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de *tester une certaine condition* et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction **if**. Pour expérimenter son fonctionnement, veuillez entrer dans votre éditeur Python les deux lignes suivantes :

```
>>> a = 150
>>> if (a > 100):
...
```

La première commande affecte la valeur 150 à la variable **a**. Jusqu'ici rien de nouveau.
Lorsque vous finissez d'entrer la seconde ligne, par contre, vous constatez que Python réagit d'une nouvelle manière. En effet, et à moins que vous n'ayez oublié le caractère « : » à la fin de la ligne, vous constatez que le *prompt principal* (**>>>**) est maintenant remplacé par un *prompt secondaire* constitué de trois points[10].

Si votre éditeur ne le fait pas automatiquement, vous devez à présent effectuer une tabulation (ou entrer 4 espaces) avant d'entrer la ligne suivante, de manière à ce que celle-ci soit *indentée* (c'est-à-dire en retrait) par rapport à la précédente. Votre écran devrait se présenter maintenant comme suit :

```
>>> a = 150
>>> if (a > 100):
...     print "a dépasse la centaine"
...
```

Frappez encore une fois <*Enter*>. Le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec **a = 20** en guise de première ligne : cette fois Python n'affiche plus rien du tout.

L'expression que vous avez placée entre parenthèses est ce que nous appellerons désormais une condition. L'instruction **if** permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons *indentée* après le « : » est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici sont optionnelles sous Python. Nous les avons utilisées pour améliorer la lisibilité. Dans d'autres langages, il se peut qu'elles soient obligatoires.

Recommencez encore, en ajoutant deux lignes comme indiqué ci-dessous. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

```
>>> a = 20
>>> if (a > 100):
...     print "a dépasse la centaine"
... else:
...     print "a ne dépasse pas cent"
...
```

Frappez <*Enter*> encore une fois. Le programme s'exécute, et affiche cette fois :

```
a ne dépasse pas cent
```

Comme vous l'aurez certainement déjà compris, l'instruction **else** (« sinon », en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de « else if ») :

[10] Dans certaines versions de l'éditeur Python pour ***Windows***, le prompt secondaire n'apparaît pas.

```
>>> a = 0
>>> if a > 0 :
...     print "a est positif"
... elif a < 0 :
...     print "a est négatif"
... else:
...     print "a est nul"
...
```

Opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les *opérateurs de comparaison* suivants :

```
x == y       # x est égal à y
x != y       # x est différent de y
x > y          # x est plus grand que y
x < y          # x est plus petit que y
x >= y       # x est plus grand que, ou égal à y
x <= y       # x est plus petit que, ou égal à y
```

Exemple :

```
>>> a = 7
>>> if (a % 2 == 0):
...     print "a est pair"
...     print "parce que le reste de sa division par 2 est nul"
... else:
...     print "a est impair"
...
```

Notez bien que l'opérateur de comparaison pour l'égalité de deux valeurs est constitué de deux signes « égale » et non d'un seul[11]. (Le signe « égale » utilisé seul est un opérateur d'affectation, et non un opérateur de comparaison. Vous retrouverez le même symbolisme en *C++* et en *Java*).

Instructions composées – Blocs d'instructions

La construction que vous avez utilisée avec l'instruction **if** est votre premier exemple d'***instruction composée***. Vous en rencontrerez bientôt d'autres. Sous Python, toutes les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête. Exemple :

```
Ligne d'en-tête:
    première instruction du bloc
    ... ...
    ... ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau* (comptez un décalage de 4 caractères, par exemple). Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple du paragraphe précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction if constituent un même bloc logique : ces deux lignes ne sont exécutées - toutes les deux - que si la condition testée avec l'instruction if se révèle vraie, c'est-à-dire si le reste de la division de **a** par 2 est nul.

[11] Rappel : l'opérateur % est l'opérateur *modulo* : il calcule le reste d'une division entière. Ainsi par exemple, **a % 2** fournit le reste de la division de **a** par 2.

Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if embranchement == "vertébrés":                    # 1
    if classe == "mammifères":                      # 2
        if ordre == "carnivores":                   # 3
            if famille == "félins":                 # 4
                print "c'est peut-être un chat"     # 5
        print "c'est en tous cas un mammifère"      # 6
    elif classe == 'oiseaux':                       # 7
        print "c'est peut-être un canari"           # 8
print"la classification des animaux est complexe"   # 9
```

Analysez cet exemple. Ce fragment de programme n'imprime la phrase « c'est peut-être un chat » que dans le cas où les quatre premières conditions testées sont vraies.

Pour que la phrase « c'est en tous cas un mammifère » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 4) se trouve en effet au même niveau d'indentation que l'instruction : `if ordre == "carnivores":` (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 & 2 sont vraies.

Pour que la phrase « c'est peut-être un canari » soit affichée, il faut que la variable `embranchement` contienne « vertébrés », et que la variable `classe` contienne « oiseaux ».

Quant à la phrase de la ligne 9, elle est affichée dans tous les cas, parce qu'elle fait partie du même bloc d'instructions que la ligne 1.

Quelques règles de syntaxe Python

Tout ce qui précède nous amène à faire le point sur quelques règles de syntaxe :

Les limites des instructions et des blocs sont définies par la mise en page

Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de fin de ligne[12] qui joue ce rôle. (Nous verrons plus loin comment outrepasser cette règle pour étendre une instruction complexe sur plusieurs lignes). On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial `#` . Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur.

Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que `begin` et `end`). En *C++* et en *Java*, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus, difficiles à relire pour les pauvres humains que nous sommes. On conseille donc à tous les programmeurs qui utilisent ces langages de se servir *aussi* des sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

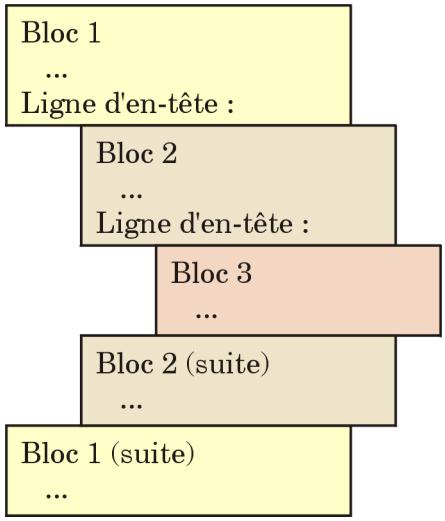
[12] Ce caractère n'apparaît ni à l'écran, ni sur les listings imprimés. Il est cependant bien présent, à un point tel qu'il fait même problème dans certains cas, parce qu'il n'est pas encodé de la même manière par tous les systèmes d'exploitation. Nous en reparlerons plus loin, à l'occasion de notre étude des fichiers texte (page 117).

Avec Python, vous *devez* utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

Instruction composée = En-tête , double point , bloc d'instructions indenté

Nous aurons de nombreuses occasions d'approfondir le concept de « bloc d'instructions » et de faire des exercices à ce sujet, dès le chapitre suivant.

Le schéma ci-contre en résume le principe.



- Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, ...) *se terminant par un double point.*
- Les blocs sont *délimités par l'indentation* : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c'est-à-dire décalées vers la droite d'un même nombre d'espaces). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.
- Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (Il n'est imbriqué dans rien).

> *Vous pouvez aussi indenter à l'aide de tabulations, mais alors vous devrez faire très attention à ne pas utiliser tantôt des espaces, tantôt des tabulations pour indenter les lignes d'un même bloc. En effet, et même si le résultat paraît identique à l'écran, espaces et tabulations sont des codes binaires distincts : Python considérera donc que ces lignes indentées différemment font partie de blocs différents. Il peut en résulter des erreurs difficiles à déboguer.*
> *En conséquence, la plupart des programmeurs préfèrent se passer des tabulations. Si vous utilisez un éditeur "intelligent", vous pouvez escamoter le problème en activant l'option "Remplacer les tabulations par des espaces".*

Les espaces et les commentaires sont normalement ignorés

A part ceux qui servent à l'indentation, en début de ligne, les espaces placés à l'intérieur des instructions et des expressions sont presque toujours ignorés (sauf s'ils font partie d'une chaîne de caractères). Il en va de même pour les commentaires : ceux-ci commencent toujours par un caractère dièse (#) et s'étendent jusqu'à la fin de la ligne courante.

4

Instructions répétitives.

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction while.

Ré-affectation

Nous ne l'avions pas encore signalé explicitement : il est permis de ré-affecter une nouvelle valeur à une même variable, autant de fois qu'on le souhaite.

L'effet d'une ré-affectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
>>> altitude = 320
>>> print altitude
320
>>> altitude = 375
>>> print altitude
375
```

Ceci nous amène à attirer une nouvelle fois votre attention sur le fait que le symbole égale utilisé sous Python pour réaliser une affectation ne doit en aucun cas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il est tentant d'interpréter l'instruction `altitude = 320` comme une affirmation d'égalité, mais ce n'en n'est pas une !

- Premièrement, l'égalité est *commutative*, alors que l'affectation ne l'est pas. Ainsi, en mathématique, les écritures `a = 7` et `7 = a` sont équivalentes, alors qu'une instruction de programmation telle que `375 = altitude` serait illégale.
- Deuxièmement, l'égalité est *permanente*, alors que l'affectation peut être remplacée comme nous venons de le voir. Lorsqu'en mathématique, nous affirmons une égalité telle que a = b au début d'un raisonnement, alors a continue à être égal à b durant tout le développement qui suit.
  En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre. Exemple :

```
>>> a = 5
>>> b = a         # a et b contiennent des valeurs égales
>>> b = 2         # a et b sont maintenant différentes
```

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément :

```
>>> a, b, c, d = 3, 4, 5, 7
```

Cette fonctionnalité de Python est bien plus intéressante encore qu'elle n'en a l'air à première vue.
Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **c**. (Actuellement, **a** contient la valeur **3**, et **c** la valeur **5**. Nous voudrions que ce soit l'inverse). Comment faire ?

4 Exercice

4.1 Écrivez les lignes d'instructions nécessaires pour obtenir ce résultat.

A la suite de l'exercice proposé ci-dessus, vous aurez certainement trouvé une méthode, et un professeur vous demanderait certainement de la commenter en classe. Comme il s'agit d'une opération courante, les langages de programmation proposent souvent des raccourcis pour l'effectuer (par exemple des instructions spécialisées, telle l'instruction SWAP du langage *Basic*). Sous Python, *l'affectation parallèle* permet de programmer l'échange d'une manière particulièrement élégante :

```
>>> a, b = b, a
```

(On pourrait bien entendu échanger d'autres variables en même temps, dans la même instruction).

Répétitions en boucle - l'instruction while

L'une des choses que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle construite à partir de l'instruction **while**.

Veuillez donc entrer les commandes ci-dessous :

```
>>> a = 0
>>> while (a < 7):        # (n'oubliez pas le double point !)
...     a = a + 1         # (n'oubliez pas l'indentation !)
...     print a
```

Frappez encore une fois <*Enter*>.

Que se passe-t-il ?

Avant de lire les commentaires de la page suivante, prenez le temps d'ouvrir votre cahier et d'y noter cette série de commandes. Décrivez aussi le résultat obtenu, et essayez de l'expliquer de la manière la plus détaillée possible.

Commentaires

Le mot **while** signifie « tant que » en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut *répéter continuellement le bloc d'instructions qui suit, tant que* le contenu de la variable a reste inférieur à 7.

Comme l'instruction if abordée au chapitre précédent, l'instruction **while** amorce une *instruction composée*. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces).

Nous avons ainsi construit notre première *boucle de programmation*, laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici comment cela fonctionne :

- Avec l'instruction **while**, Python commence par évaluer la validité de la *condition* fournie entre parenthèses (Celles-ci sont optionnelles. Nous ne les avons utilisées que pour clarifier notre explication).

- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine[13].
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant *le corps de la boucle*, c'est-à-dire :
  - l'instruction **a = a + 1** qui incrémente d'une unité le contenu de la variable **a**
    (ce qui signifie que l'on affecte à la variable a une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
  - l'instruction **print** qui affiche la valeur courante de la variable **a**
- lorsque ces deux instructions ont été exécutées, nous avons assisté à une première *itération*, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction **while**. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite.
  Dans notre exemple, si la condition **a < 7** est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.

Remarques

- La variable évaluée dans la condition doit exister au préalable (Il faut qu'on lui ait déjà affecté au moins une valeur)
- Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner). Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par while, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer.
  Exemple de boucle sans fin (à éviter !) :

```
>>> n = 3
>>> while n < 5:
...     print "hello !"
```

Élaboration de tables

Recommencez à présent le premier exercice, mais avec la petite modification ci-dessous :

```
>>> a = 0
>>> while a < 12:
...     a = a +1
...     print a , a**2 , a**3
```

Vous devriez obtenir la liste des carrés et des cubes des nombres de 1 à 12.
Notez au passage que l'instruction **print** permet d'afficher plusieurs expressions l'une à la suite de l'autre sur la même ligne : il suffit de les séparer par des virgules. Python insère automatiquement un espace entre les éléments affichés.

Construction d'une suite mathématique

Le petit programme ci-dessous permet d'afficher les dix premiers termes d'une suite appelée « *Suite de Fibonacci* ». Il s'agit d'une suite de nombres, dont chaque terme est égal à la somme des

[13] ... du moins dans cet exemple. Nous verrons un peu plus loin qu'en fait l'exécution continue avec la première instruction qui suit le bloc indenté, et qui fait partie du même bloc que l'instruction while elle-même.

deux termes qui le précèdent. Analysez ce programme (qui utilise judicieusement l'affectation parallèle). Décrivez le mieux possible le rôle de chacune des instructions.

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
...     print b,
...     a, b, c = b, a+b, c+1
```

Lorsque vous lancez l'exécution de ce programme, vous obtenez :

```
1 2 3 5 8 13 21 34 55 89
```

Les termes de la suite de Fibonacci sont affichés sur la même ligne. Vous obtenez ce résultat grâce à la virgule placée à la fin de la ligne qui contient l'instruction **print**. Si vous supprimez cette virgule, les nombres seront affichés l'un en-dessous de l'autre.

Dans vos programmes futurs, vous serez très souvent amenés à mettre au point des boucles de répétition comme celle que nous analysons ici. Il s'agit d'une question essentielle, que vous devez apprendre à maîtriser parfaitement. Soyez sûr que vous y arriverez progressivement, à force d'exercices.

Lorsque vous examinez un problème de cette nature, vous devez considérer les lignes d'instruction, bien entendu, mais surtout décortiquer *les états successifs des différentes variables* impliquées dans la boucle. Cela n'est pas toujours facile, loin de là. Pour vous aider à y voir plus clair, prenez la peine de dessiner sur papier une table d'états similaire à celle que nous reproduisons ci-dessous pour notre programme « suite de Fibonacci » :

Variables	a	b	c
Valeurs initiales	1	1	1
Valeurs prises successivement, au cours des itérations	1 2 3 5 ...	2 3 5 8 ...	2 3 4 5 ...
Expression de remplacement	b	a+b	c+1

Dans une telle table, on effectue en quelque sorte « à la main » le travail de l'ordinateur, en indiquant ligne par ligne les valeurs que prendront chacune des variables au fur et à mesure des itérations successives. On commence par inscrire en haut du tableau les noms des variables concernées. Sur la ligne suivante, les valeurs initiales de ces variables (valeurs qu'elles possèdent avant le démarrage de la boucle). Enfin, tout en bas du tableau, les expressions utilisées dans la boucle pour modifier l'état de chaque variable à chaque itération.

On remplit alors quelques lignes correspondant aux premières itérations. Pour établir les valeurs d'une ligne, il suffit d'appliquer à celles de la ligne précédente, l'expression de remplacement qui se trouve en bas de chaque colonne. On vérifie ainsi que l'on obtient bien la suite recherchée. Si ce n'est pas le cas, il faut essayer d'autres expressions de remplacement.

Exercices :

4.2 Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.

4.3 Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous :

```
1 euro(s) = 1.65 dollar(s)
2 euro(s) = 3.30 dollar(s)
4 euro(s) = 6.60 dollar(s)
8 euro(s) = 13.20 dollar(s)
```

etc. (S'arrêter à 16384 euros)

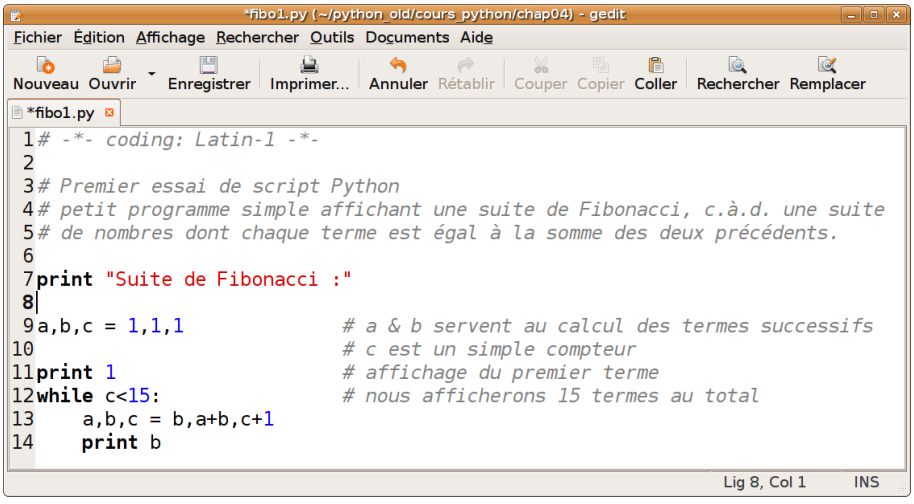
4.4 Écrivez un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.

Premiers scripts, ou comment conserver nos programmes

Jusqu'à présent, vous avez toujours utilisé Python *en mode interactif* (c'est-à-dire que vous avez à chaque fois entré les commandes directement dans l'interpréteur, sans les sauvegarder au préalable dans un fichier). Cela vous a permis d'apprendre très rapidement les bases du langage, par expérimentation directe. Cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions que vous avez écrites disparaissent irrémédiablement dès que vous fermez l'interpréteur. Avant de poursuivre plus avant votre étude, il est donc temps que vous appreniez à sauvegarder vos programmes dans des fichiers, sur disque dur ou clef USB, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, vous allez désormais rédiger vos séquences d'instructions dans un *éditeur de textes* quelconque (par exemple *Kate*, *Gedit*, *Geany*, *Scite* ... sous *Linux*, *Edit* sous *MS-DOS*, *Wordpad* sous *Windows*, ou encore l'éditeur incorporé dans une interface de développement telle que *IDLE* ou *PythonWin*). Ainsi vous écrirez *un script*, que vous pourrez ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur[14].

La figure ci-dessous illustre l'utilisation de l'éditeur *Gedit* sous *Gnome* (*Ubuntu Linux*) :



Par la suite, lorsque vous voudrez tester l'exécution de votre programme, il vous suffira de lancer l'interpréteur Python en lui fournissant (comme argument) le nom du fichier qui contient le

[14] Il serait parfaitement possible d'utiliser un système de traitement de textes, à la condition d'effectuer la sauvegarde sous un format "texte pur" (sans balises de mise en page). Il est cependant préférable d'utiliser un véritable éditeur "intelligent" tel que *Geany*, *DrPython*, *Scite* ou *IDLE*, muni d'une fonction de coloration syntaxique pour Python, qui vous aide à éviter les fautes de syntaxe. Avec *IDLE*, suivez le menu : File → New window (ou frappez CTRL-N) pour ouvrir une nouvelle fenêtre dans laquelle vous écrirez votre script. Pour l'exécuter, il vous suffira (après sauvegarde), de suivre le menu : Edit → Run script (ou de frapper CTRL-F5).

script. Par exemple, si vous avez placé un script dans un fichier nommé « MonScript », il suffira d'entrer la commande suivante dans une fenêtre de terminal pour que ce script s'exécute :

```
python MonScript
```

Pour faire mieux encore, veillez à donner au fichier un nom qui se termine par l'extension **.py**

Si vous respectez cette convention, vous pourrez (sous *Windows*, *MacOs*, ou la plupart des gestionnaires de fichiers graphiques en usage sous *Linux*) lancer l'exécution du script, simplement en cliquant sur son nom ou sur l'icône correspondante dans le gestionnaire de fichiers (c'est-à-dire l'explorateur, sous *Windows*, ou bien *Nautilus*, *Konqueror*, ... sous *Linux*).

Ces gestionnaires graphiques « savent » en effet qu'il doivent lancer l'interpréteur Python chaque fois que leur utilisateur essaye d'ouvrir un fichier dont le nom se termine par .py. (Cela suppose bien entendu qu'ils aient été correctement configurés). La même convention permet en outre aux éditeurs « intelligents » de reconnaître automatiquement les scripts Python et d'adapter leur coloration syntaxique en conséquence.

Un script Python contiendra des séquences d'instructions identiques à celles que vous avez expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues plus tard par vous-même ou par d'autres, *il vous est très fortement recommandé d'expliciter vos scripts le mieux possible, en y incorporant de nombreux commentaires*. La principale difficulté de la programmation consiste en effet à mettre au point des algorithmes corrects. Afin que ces algorithmes puissent être vérifiés, corrigés, modifiés, etc. dans de bonnes conditions, il est essentiel que leur auteur les décrive le plus complètement et le plus clairement possible. Et le meilleur emplacement pour cette description est le corps même du script (ainsi elle ne peut pas s'égarer).

> *Un bon programmeur veille toujours à insérer un grand nombre de commentaires dans ses scripts. En procédant ainsi, non seulement il facilite la compréhension de ses algorithmes pour d'autres lecteurs éventuels, mais encore il se force lui-même à avoir les idées plus claires.*

On peut insérer des commentaires quelconques à peu près n'importe où dans un script. Il suffit de les faire précéder d'un caractère *#*. Lorsqu'il rencontre ce caractère, l'interpréteur Python ignore tout ce qui suit, jusqu'à la fin de la ligne courante.

Comprenez bien qu'il est important d'inclure des commentaires *au fur et à mesure de l'avancement de votre travail de programmation*. N'attendez pas que votre script soit terminé pour les ajouter « après coup ». Vous vous rendrez progressivement compte qu'un programmeur passe énormément de temps à relire son propre code (pour le modifier, y rechercher des erreurs, etc). Cette relecture sera grandement facilitée si le code comporte de nombreuses explications et remarques.

Ouvrez donc un éditeur de texte, et rédigez le script ci-dessous :

```
# -*- coding: Latin-1 -*-

# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

a, b, c = 1, 1, 1           # a & b servent au calcul des termes successifs
                            # c est un simple compteur
print 1                     # affichage du premier terme
while c<15:                 # nous afficherons 15 termes au total
    a, b, c = b, a+b, c+1
    print b
```

La première ligne, un peu étrange, doit être reproduite telle quelle. Ne vous préoccupez pas de son rôle pour l'instant, il sera expliqué un peu plus loin dans cette page.

Afin de vous montrer tout de suite le bon exemple, nous commençons ce script par trois lignes de commentaires, qui contiennent une courte description de la fonctionnalité du programme. Prenez l'habitude de faire de même dans vos propres scripts.

Les lignes de code elle-mêmes sont documentées. Si vous procédez comme nous l'avons fait, c'est-à-dire en insérant des commentaires à la droite des instructions correspondantes, veillez à les écarter suffisamment de celles-ci, afin de ne pas gêner leur lisibilité.

Lorsque vous aurez bien vérifié votre texte, sauvegardez-le et exécutez-le.

> *Bien que ce ne soit pas indispensable, nous vous recommandons une fois encore de choisir pour vos scripts des noms de fichiers se terminant par l'extension .py Cela aide beaucoup à les identifier comme tels dans un répertoire. Les gestionnaires graphiques de fichiers (explorateur Windows, Konqueror) se servent d'ailleurs de cette extension pour leur associer une icône spécifique.* ***Evitez cependant de choisir des noms qui risqueraient d'être déjà attribués à des modules python existants : des noms tels que math.py ou Tkinter.py, par exemple, sont à proscrire !***

Si vous travaillez en mode texte sous *Linux*, ou dans une fenêtre *MSDOS*, vous pouvez exécuter votre script à l'aide de la commande **python** suivie du nom du script. Si vous travaillez en mode graphique sous *Linux*, vous pouvez ouvrir une *fenêtre de terminal* et faire la même chose :

```
python monScript.py
```

Dans l'explorateur *Windows*, vous pouvez lancer l'exécution de votre script en effectuant un simple clic de souris sur l'icône correspondante (en principe, si *IDLE* a été installé).

Si vous travaillez avec *IDLE*, vous pouvez lancer l'exécution du script en cours d'édition, directement à l'aide de la combinaison de touches <**Ctrl**-**F5**>. Dans d'autres environnements de travail spécifiquement dédiés à Python, tels que *DrPython*, *Geany* ou *Scite*, vous trouverez également des icônes et/ou des raccourcis clavier pour lancer l'exécution. Consultez votre professeur ou votre gourou local concernant les autres possibilités de lancement éventuelles sur différents systèmes d'exploitation.

Remarque concernant les caractères accentués

A partir de la version 2.3, il est vivement recommandé aux francophones d'inclure l'un des pseudo-commentaires suivants au début de tous leurs scripts Python (obligatoirement à la 1e ou à la 2e ligne) :

```
# -*- coding:Latin-1 -*-
```

Ou bien :

```
# -*- coding:Utf-8 -*-
```

Ces pseudo-commentaires indiquent à Python que vous utiliserez dans votre script :

- soit le jeu de caractères accentués *ASCII étendu* correspondant aux principales langues de l'Europe occidentale (Français, Allemand, Portugais, etc.), encodé sur un seul octet suivant la norme *ISO-8859-1*, laquelle est souvent désignée aussi par l'étiquette *Latin-1* ;
- soit un jeu de caractères beaucoup plus étendu suivant la nouvelle norme *Unicode*, laquelle a été mise au point pour standardiser la représentation numérique de tous les caractères spécifiques des différentes langues mondiales. Il existe plusieurs représentations ou encodages de

cette norme, et nous devrons approfondir cette question un peu plus loin, mais pour l'instant il vous suffit de savoir que l'encodage le plus répandu sur les ordinateurs récents est *Utf-8*. Dans ce système, les caractères standard (*ASCII*) sont encore encodés sur un seul octet, ce qui assure une certaine compatibilité avec le codage *Latin-1*, mais les autres caractères (parmi lesquels nos caractères accentués) peuvent être encodés sur 2, 3 ou même parfois 4 octets.

Python peut traiter tout à fait correctement les caractères de ces deux systèmes, mais vous devez lui signaler lequel vous utilisez, à l'aide d'un pseudo-commentaires en début de script.

Si vous ne le faites pas, vos scripts ne pourront pas s'exécuter, ou alors ils fourniront des résultats étranges et incorrects. Supprimez par exemple la première ligne du script précédent, puis lancez à nouveau son exécution. Avec une version récente de Python, vous obtiendrez un message d'erreur similaire à celui-ci :

```
SyntaxError: Non-ASCII character '\xe0' in file fibo2.py on line 2, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Si votre système d'exploitation est configuré de telle manière que les frappes clavier génèrent des codes Utf-8, configurez votre éditeur de textes pour qu'il utilise lui aussi ce codage, et placez le second des pseudo-commentaires indiqués ci-dessus au début de chacun de vos scripts.

Si votre système d'exploitation fonctionne suivant la norme ancienne (*ISO-8859-1*), vous devrez utiliser plutôt le premier pseudo-commentaire.

Si vous ignorez le mode d'encodage par défaut de votre système d'exploitation, vous pouvez vous servir de Python pour le découvrir. Lancez votre interpréteur python, puis entrez à la ligne de commande une simple instruction d'assignation telle que `prenom = "Gérard"` , par laquelle vous affectez à la variable **prenom** une chaîne de caractères contenant au moins une lettre accentuée. Entrez ensuite le nom de cette variable, suivi de *<Enter>*. (Rappelez-vous qu'à la ligne de commande, vous pouvez obtenir ainsi de l'interpréteur qu'il vous indique le contenu détaillé d'une variable quelconque.

Si vous obtenez un résultat tel que :

```
>>> prenom = "Gérard"
>>> prenom
'G\xc3\xa9rard'
>>>
```

cela signifie que votre système d'exploitation utilise l'encodage **Utf-8**. Python vous indique ainsi, en effet, que le second caractère de la chaîne « Gérard » est un caractère *non-ASCII* encodé sur deux octets, représentés ici en format hexadécimal, respectivement par **\xc3** et **\xa9**.

Par contre, si vous obtenez le résultat suivant :

```
>>> prenom = "Gérard"
>>> prenom
'G\xe9rard'
>>>
```

cela signifie que votre système d'exploitation utilise toujours la norme ancienne **Latin-1**. Python vous indique en effet que la même lettre accentuée est encodée cette fois à l'aide d'un seul octet : **\xe9**.

Nous devrons approfondir encore cette question des différents encodages, lorsque nous étudierons un peu plus en détail le traitement des chaînes de caractères (Voir le chapitre suivant, page 50).

Exercices :

4.5 Ecrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.

4.6 Ecrivez un programme qui convertisse un nombre entier de secondes fourni au départ, en un nombre d'années, de mois, de jours, de minutes et de secondes.
(Utilisez l'opérateur modulo : %).

4.7 Ecrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3.
Exemple : 7 14 21 * 28 35 42 * 49

4.8 Ecrivez un programme qui calcule les 50 premiers termes de la table de multiplication par 13, mais n'affiche que ceux qui sont des multiples de 7.

4.9 Ecrivez un programme qui affiche la suite de symboles suivante :

```
*
**
***
****
*****
******
*******
```

5

Principaux types de données

Dans le chapitre 2, nous avons déjà manipulé des données de différents types : des nombres entiers ou réels, et des chaînes de caractères. Il est temps à présent d'examiner d'un peu plus près ces types de données, et également de vous en faire découvrir d'autres.

Les données numériques

Dans les exercices réalisés jusqu'à présent, nous avons déjà utilisé des données de deux types : les nombres *entiers* ordinaires et les nombres *réels* (aussi appelés nombres *à virgule flottante*). Tâchons de mettre en évidence les caractéristiques (et les limites) de ces concepts :

Les types « integer » et « long »

Supposons que nous voulions modifier légèrement notre précédent exercice sur la suite de Fibonacci, de manière à obtenir l'affichage d'un plus grand nombre de termes. A priori, il suffit de modifier la condition de bouclage, dans la deuxième ligne. Avec `while c<49:` , nous devrions obtenir quarante-huit termes. Modifions donc légèrement l'exercice, de manière à afficher aussi le type de la variable principale :

```
>>> a, b, c = 1, 1, 1
>>> while c<49:
        print c, " : ", b, type(b)
        a, b, c = b, a+b, c+1
...
...
...  (affichage des 43 premiers termes)
...
44  :  1134903170 <type 'int'>
45  :  1836311903 <type 'int'>
46  :  2971215073 <type 'long'>
47  :  4807526976 <type 'long'>
48  :  7778742049 <type 'long'>
```

Que pouvons-nous constater ?

Si nous n'avions pas utilisé la fonction **type()**, qui nous permet de vérifier à chaque itération le type de la variable **b**, nous n'aurions rien remarqué du tout : la suite des nombres de Fibonacci s'affiche sans problème (et nous pourrions encore l'allonger de nombreux termes supplémentaires).

Il semble donc que Python soit capable de traiter des nombres entiers de taille illimitée.

L'exercice que nous venons de réaliser indique cependant qu'il se passe « quelque chose » lorsque ces nombres deviennent très grands. Au début du programme, les variables **a**, **b** et **c** sont définies implicitement comme étant du type *integer*. C'est ce qui se passe toujours avec Python lorsqu'on affecte une valeur entière à une variable, à condition que cette valeur ne soit pas trop grande. Dans la mémoire de l'ordinateur, ce type de donnée est en effet encodé sous la forme d'un bloc de

4 octets (ou 32 bits). Or la gamme de valeurs décimales qu'il est possible d'encoder sur 4 octets seulement s'étend de -2147483648 à + 2147483647 (Voir cours d'informatique générale).

Les calculs effectués avec ce type de variable sont toujours très rapides, parce que le processeur de l'ordinateur est capable de traiter directement par lui-même de tels nombres entiers à 32 bits. En revanche, lorsqu'il est question de traiter des nombres entiers plus grands, ou encore des nombres réels (nombres « à virgule flottante »), les logiciels que sont les interpréteurs et compilateurs doivent effectuer un gros travail de codage/décodage, afin de ne présenter en définitive au processeur que des opérations binaires sur des nombres entiers de 32 bits au maximum.

Vous savez déjà que le type des variables Python est défini de manière dynamique.

Puisqu'il s'agit du type le plus performant (aussi bien en termes de vitesse de calcul qu'en termes d'occupation de place dans la mémoire), Python utilise le type **integer** par défaut, chaque fois que cela est possible, c'est-à-dire tant que les valeurs traitées sont des entiers compris entre les limites déjà mentionnées plus haut (environ 2 milliards, en positif ou en négatif).

Lorsque les valeurs traitées sont des nombres entiers se situant au-delà de ces limites, leur encodage dans la mémoire de l'ordinateur devient plus complexe. Les variables auxquelles on affecte de tels nombres sont alors automatiquement définies comme appartenant au type « entier long » (lequel est désigné par **long** dans la terminologie Python).

Ce type **long** permet l'encodage de valeurs entières avec une précision quasi infinie : une valeur définie sous cette forme peut en effet posséder un nombre de chiffres significatifs quelconque, ce nombre n'étant limité *que par la taille de la mémoire disponible sur l'ordinateur utilisé !*

Exemple :

```
>>> a, b, c = 3, 2, 1
>>> while c < 15:
        print c, ": ", b
        a, b, c = b, a*b, c+1

1 :  2
2 :  6
3 :  12
4 :  72
5 :  864
6 :  62208
7 :  53747712
8 :  3343537668096
9 :  179707499645975396352
10 :  6008587943056673222701554251857 92
11 :  10797883156496691381438492294473845785924307043903078 4
12 :  648800305446607527907368373691049776950010342842280428918276494561862 34
582611607420928
13 :  700566989011183200292376413995762169216245450579726979173836923132717 54
883621235064434673400268965204696103008832506249008437424702378475 52
14 :  454528076456265799856362940482493512051682398707229461514016556556583 98
642227616335815123825782460196980206141536747116094173550514227947953005917 00
969504226930790382476340558291752968319462245039335017547760330040127583682 56
>>>
```

Dans l'exemple ci-dessus, la valeur des nombres affichés augmente très rapidement, car chacun d'eux est égal au produit des deux termes précédents.

Au départ, les variables a, **b** et **c** sont du type *integer*, puisqu'on leur affecte des petites valeurs numériques entières : **3**, **2** et **1**. A partir de la 8e itération, cependant, les variables **b** et **a** sont automatiquement converties l'une après l'autre dans le type *long* : le résultat de la multiplication des termes 6 et 7 est en effet déjà bien supérieur à la limite des 2 milliards évoquée plus haut.

La progression continue avec des nombres de plus en plus gigantesques, mais la vitesse de calcul diminue. Les nombres mémorisés sous le type long occupent une place variable dans la mémoire de l'ordinateur, en fonction de leur taille.

Le type « float »

Vous avez déjà rencontré précédemment cet autre type de donnée numérique : le type « nombre réel », ou « nombre à virgule flottante », désigné en anglais par l'expression *floating point number*, et que pour cette raison on appellera type **float** sous Python.

Ce type autorise les calculs sur de très grands ou très petits nombres (données scientifiques, par exemple), avec un degré de précision constant.

Pour qu'une donnée numérique soit considérée par Python comme étant du type **float**, il suffit qu'elle contienne dans sa formulation un élément tel qu'un point décimal ou un exposant de 10.

Par exemple, les données :

```
      3.14      10.      .001      1e100      3.14e-10
```

sont automatiquement interprétées par Python comme étant du type ***float***.

Essayons donc ce type de données dans un nouveau petit programme (inspiré du précédent) :

```
>>> a, b, c = 1., 2., 1            # => a et b seront du type 'float'
>>> while c <18:
...     a, b, c = b, b*a, c+1
...     print b

2.0
4.0
8.0
32.0
256.0
8192.0
2097152.0
17179869184.0
3.6028797019e+16
6.18970019643e+26
2.23007451985e+43
1.38034926936e+70
3.07828173409e+113
4.24910394253e+183
1.30799390526e+297
         Inf
         Inf
```

Comme vous l'aurez certainement bien compris, nous affichons cette fois encore une série dont les termes augmentent extrêmement vite, chacun d'eux étant égal au produit des deux précédents. Au huitième terme, nous dépassons déjà largement la capacité d'un *integer*. Au neuvième terme, Python passe automatiquement à la notation scientifique (« e+n » signifie en fait : « fois dix à l'exposant n »). Après le quinzième terme, nous assistons à nouveau à un dépassement de capacité (sans message d'erreur) : les nombres vraiment trop grands sont tout simplement notés « inf » (pour « infini »).

Le type *float* utilisé dans notre exemple permet de manipuler des nombres (positifs ou négatifs) compris entre 10^{-308} et 10^{308} avec une précision de 12 chiffres significatifs. Ces nombres sont encodés d'une manière particulière sur 8 octets (64 bits) dans la mémoire de la machine : une partie du code correspond aux 12 chiffres significatifs, et une autre à l'ordre de grandeur (exposant de 10).

Exercices :

5.1 Écrivez un programme qui convertisse en radians un angle fourni au départ en degrés, minutes, secondes.

5.2 Écrivez un programme qui convertisse en degrés, minutes, secondes un angle fourni au départ en radians.

5.3 Écrivez un programme qui convertisse en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse.
La formule de conversion est : .

5.4 Écrivez un programme qui calcule les intérêts accumulés chaque année pendant 20 ans, par capitalisation d'une somme de 100 euros placée en banque au taux fixe de 4,3 %

5.5 Une légende de l'Inde ancienne raconte que le jeu d'échecs a été inventé par un vieux sage, que son roi voulut remercier en lui affirmant qu'il lui accorderait n'importe quel cadeau en récompense. Le vieux sage demanda qu'on lui fournisse simplement un peu de riz pour ses vieux jours, et plus précisément un nombre de grains de riz suffisant pour que l'on puisse en déposer 1 seul sur la première case du jeu qu'il venait d'inventer, deux sur la suivante, quatre sur la troisième, et ainsi de suite jusqu'à la 64^{e} case.
Écrivez un programme Python qui affiche le nombre de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières :
- le nombre exact de grains (nombre entier)
- le nombre de grains en notation scientifique (nombre réel)

Les données alphanumériques

Jusqu'à présent nous n'avons manipulé que des nombres. Mais un programme d'ordinateur peut également traiter des caractères alphabétiques, des mots, des phrases, ou des suites de symboles quelconques. Dans la plupart des langages de programmation, il existe pour cet usage des structures de données particulières que l'on appelle « chaînes de caractères ».

Sous Python, il existe deux structures de données distinctes pour traiter les chaînes de caractères : le type **string** et le type **unicode**. Le type *string* est le plus fondamental, et c'est celui que nous utiliserons le plus souvent dans ce cours. Le type *unicode* est plus élaboré : il a été inventé afin de simplifier le traitement de tous les caractères d'écriture utilisés dans le monde entier, les accents, les symboles mathématiques, etc. Si votre système d'exploitation est encore configuré de manière à utiliser l'encodage Latin-1 par défaut (Voir le chapitre précédent, page 44), vous pouvez probablement plus ou moins ignorer le type *unicode*, tout au moins à vos débuts. Si vous utilisez un ordinateur récent, par contre, il y a gros à parier que votre système d'exploitation utilise la nouvelle norme Utf-8 par défaut (ce qui est par ailleurs une excellente chose), et dans ce cas vous devrez le plus tôt possible apprendre à distinguer les deux types, à les traiter correctement, à convertir les données d'un type à l'autre, et inversément.

Le type « string »

Une donnée de type *string* peut se définir en première approximation comme une suite quelconque de caractères. Dans un script python, on peut délimiter une telle suite de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). Exemples :

```
>>> phrase1 = 'les oeufs durs.'
>>> phrase2 = '"Oui", répondit-il,'
>>> phrase3 = "j'aime bien"
>>> print phrase2, phrase3, phrase1
"Oui", répondit-il, j'aime bien les oeufs durs.
```

Les 3 variables **phrase1**, **phrase2**, **phrase3** sont donc des variables de type *string*.

Remarquez l'utilisation des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes, ou l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets. Remarquez aussi encore une fois que l'instruction **print** insère un espace entre les éléments affichés.

Le caractère spécial « \ » (*antislash*) permet quelques subtilités complémentaires :

- En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).
- A l'intérieur d'une chaîne de caractères, l'*antislash* permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Exemples :

```
>>> txt3 = '"N\'est-ce pas ?" répondit-elle.'
>>> print txt3
"N'est-ce pas ?" répondit-elle.

>>> Salut = "Ceci est une chaîne plutôt longue\n contenant plusieurs lignes \
... de texte (Ceci fonctionne\n de la même façon en C/C++.\n\
...     Notez que les blancs en début\n de ligne sont significatifs.\n"
>>> print Salut
Ceci est une chaîne plutôt longue
 contenant plusieurs lignes de texte (Ceci fonctionne
 de la même façon en C/C++.
    Notez que les blancs en début
 de ligne sont significatifs.
```

Remarques :

- La séquence **\n** dans une chaîne provoque un saut à la ligne.
- La séquence **\'** permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes. De la même manière, la séquence **\"** permet d'insérer des guillemets dans une chaîne délimitée elle-même par des guillemets.
- Rappelons encore ici que la casse est significative dans les noms de variables (Il faut respecter scrupuleusement le choix initial de majuscules ou minuscules).

« Triple quotes » :

Pour insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage de l'*antislash*, ou pour faire accepter l'*antislash* lui-même dans la chaîne, on peut encore délimiter la chaîne à l'aide de *triples guillemets* ou de *triples apostrophes* :

```
>>> a1 = """
... Usage: trucmuche[OPTIONS]
... { -h
...   -H hôte
... }"""

>>> print a1

Usage: trucmuche[OPTIONS]
{ -h
  -H hôte
}
```

Accès aux caractères individuels d'une chaîne

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des *données composites*. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. En fonction des circonstances, nous souhaiterons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères distincts. Un langage de programmation tel que Python doit donc être pourvu de mécanismes qui permettent d'accéder séparément à chacun des caractères d'une chaîne. Comme vous allez le voir, cela n'est pas bien compliqué.

Python considère qu'une chaîne de caractères est un objet de la catégorie des *séquences*, lesquelles sont des *collections ordonnées d'éléments*. Cela signifie simplement que les caractères d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un *index*.

Pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne, et on lui accole entre deux crochets l'index numérique qui correspond à la position du caractère dans la chaîne.

Attention, cependant : comme vous aurez l'occasion de le vérifier par ailleurs, les données informatiques sont presque toujours numérotées *à partir de zéro* (et non à partir de un). C'est le cas pour les caractères d'une chaîne.

Exemple :

```
>>> ch = "Christine"
>>> print ch[0], ch[3], ch[5]
C i t
```

Limitations du type « string »

Veuillez donc recommencer l'exercice de l'exemple ci-dessus, mais en utilisant cette fois un ou deux caractères « non-ASCII », tels que lettres accentuées, cédilles, etc. Les choses se compliquent quelque peu, suivant que votre système d'exploitation utilise l'encodage par défaut *Latin-1* ou *Utf-8* :

Si votre ordinateur utilise l'encodage *Latin-1*, tout semble se passer comme prévu :

```
>>> ch ="Noël en Décembre"
>>> >>> print ch[1],ch[2],ch[3],ch[4],ch[8],ch[9],ch[10],ch[11],ch[12]
o ë l   D é c e m
```

Par contre, si votre utilisateur utilise l'encodage *Utf-8*, les résultats deviennent bizarres :

```
>>> ch ="Noël en Décembre"
>>> print ch[1],ch[2],ch[3],ch[4],ch[8],ch[9],ch[10],ch[11],ch[12]
o ? ? l   D ? ? c
```

Dans la mémoire d'un ordinateur utilisant l'encodage *Latin-1*, chaque caractère alphanumérique est représenté par un seul octet. Sachant qu'un octet peut représenter 256 valeurs différentes, vous comprenez aisément qu'il est possible d'encoder sur un seul octet, non seulement l'ensemble des caractères standards *ASCII* (caractères non accentués et chiffres, plus quelques autres symboles typographiques courants, encodés avec les valeurs d'octet de 0 à 127), mais aussi un certain nombre de caractères plus particuliers, malheureusement pas toujours les mêmes d'une région du monde à l'autre (encodés avec les valeurs d'octet de 128 à 255). Dans le cas de la norme *ISO-8859-1* ou *Latin-1* qui nous intéresse ici, ces caractères supplémentaires sont les principaux caractères accentués et symboles divers utilisés en Europe occidentale. Cette norme permet donc

l'encodage correct de textes courants ordinaires dans notre langue, mais elle interdit d'y incorporer par exemple du grec, de l'arabe, du russe, du japonais, etc., ainsi qu'un grand nombre de symboles mathématiques ou techniques.

L'encodage *Latin-1* n'est donc en définitive qu'une médiocre amélioration de l'encodage standard *ASCII*, vaguement adapté à un groupe de langues particulières (français, espagnol, allemand, ...) grâce à l'appoint d'une petite centaine de caractères, mais aux possibilités tout de même très limitées. En fait, le seul intérêt résiduel de cette norme ancienne est sa simplicité. Suivant cette convention, en effet, les chaînes de caractères ne sont rien d'autre que des séquences d'octets, et leur traitement informatique reste par conséquent assez simple.

À l'heure actuelle, nous ne pouvons cependant plus nous satisfaire de cette simplicité trompeuse. Les ordinateurs ont envahi tous les secteurs d'activité, et le monde est devenu un village. Il nous faut donc intégrer l'idée que les diverses suites de symboles ingurgités par nos ordinateurs (c'est-à-dire non seulement les textes qui mélangeront différentes langues, mais aussi les équations mathématiques, chimiques, etc.) devront tôt ou tard accepter la coexistence de caractères extrêmement variés.

Étant donnée l'impossibilité évidente de représenter plus de 256 caractères différents à l'aide d'un seul octet, il faut donc élaborer de nouveaux concepts, et établir de nouvelles normes de codage.

Comme cela a déjà été signalé plus haut, la norme *Utf-8* qu'utilisent les ordinateurs récents permet d'encoder une multitude de caractères et de symboles de toute sorte. Afin d'assurer une certaine compatibilité avec les textes encodés aux normes anciennes, les caractères standards continuent à y être encodés sur un seul octet, exactement comme en *ASCII*, mais tous les autres caractères « exotiques », comme par exemple nos lettres accentuées, y sont encodés sur deux octets ou davantage.

La conséquence la plus importante qui découle de cette convention est que nous allons devoir désormais établir une distinction nette entre les concepts de « chaîne de caractères » et de « séquence d'octets ». Si nous travaillons sur un ordinateur moderne utilisant l'encodage *Utf-8* par défaut, cela apparaît assez clairement, comme dans notre dernier exemple (reproduit une fois encore ci-dessous) :

```
>>> ch ="Noël en Décembre"
>>> print ch[1],ch[2],ch[3],ch[4],ch[8],ch[9],ch[10],ch[11],ch[12]
o ▯ ▯ l   D ▯ ▯ c
```

Pour comprendre ce qui se passe dans cet exemple, il suffit d'admettre en effet qu'après l'affectation de la première ligne, la variable **ch** contient, non pas vraiment une chaîne de caractères, *mais plutôt une séquence d'octets*. Lorsque nous demandons à Python d'afficher séparément les éléments n°1, n°2, n°3, etc. de cette séquence, nous n'obtenons des caractères que pour les octets qui peuvent effectivement représenter des caractères en *Utf-8* (c'est-à-dire seulement les octets de valeur inférieure à 128, lesquels peuvent toujours représenter des caractères *ASCII* standards). Pour les autres, par contre, Python déclare forfait, parce qu'en application de la norme *Utf-8*, les octets de valeur supérieure à 127 ne peuvent pas être interprétés séparément comme des caractères : ils doivent pour cela être évalués par paires (ou même par triplets ou par quadruplets dans certains cas).

La conclusion de tout ceci est que le type de données **string** :

- doit être compris comme une séquence d'octets, et non de caractères alphanumériques ;
- n'est adapté au traitement détaillé de chaînes de caractères, qu'à la condition que celles-ci soient encodées en ASCII ou l'une de ses variantes « étendues » telles que Latin-1, ce qui sera de moins en moins la norme à l'avenir.

Afin de s'affranchir de ces limitations, Python s'est doté (à partir de sa version 1.6) d'un nouveau type de données appelé **unicode**. Celui-ci autorise dorénavant le traitement détaillé des chaînes de caractères, en faisant abstraction de la manière dont ceux-ci sont encodés dans la mémoire de l'ordinateur.

Nous étudierons le type *unicode* en détail au chapitre 10 (voir page 123). En attendant, nous allons provisoirement considérer que vous n'utilisez, dans les exercices nécessitant un traitement détaillé des chaînes de caractères (c'est-à-dire les exercices où il est question d'accéder aux caractères individuels de la chaîne en question), que les lettres non accentuées du jeu standard *ASCII*. Rien ne vous empêche d'utiliser aussi des lettres accentuées, notamment si votre poste de travail utilise toujours la norme *Latin-1*. Mais si vous le faites, attendez-vous à obtenir dans certains cas des résultats étranges.

Opérations élémentaires sur les chaînes

Python intègre de nombreuses *fonctions* qui permettent d'effectuer divers traitements sur les chaînes de caractères (conversions majuscules/minuscules, découpage en chaînes plus petites, recherche de mots, etc.). Une fois de plus, cependant, nous devons vous demander de patienter : ces questions ne seront développées qu'à partir du chapitre 10 (voir page 123).

Pour l'instant, nous pouvons nous contenter de savoir qu'il est possible d'accéder individuellement à chacun des caractères d'une chaîne, comme cela a été expliqué à la rubrique précédente. Sachons en outre que l'on peut aussi :

- assembler plusieurs petites chaînes pour en construire de plus grandes. Cette opération s'appelle *concaténation* et on la réalise sous Python à l'aide de l'opérateur `+` (Cet opérateur réalise donc l'opération d'addition lorsqu'on l'applique à des nombres, et l'opération de concaténation lorsqu'on l'applique à des chaînes de caractères. Exemple :

```
a = 'Petit poisson'
b = ' deviendra grand'
c = a + b
print c
petit poisson deviendra grand
```

- déterminer la longueur (c'est-à-dire le nombre de caractères) d'une chaîne, en faisant appel à la fonction intégrée **len()** :

```
>>> ch ='Georges'
>>> print len(ch)
7
```

Attention : comme expliqué à la rubrique précédente, les données de type string doivent être comprises comme des séquences d'octets, et non de véritables chaînes de caractères. Si vous testez ainsi des chaînes contenant des caractères accentués, et que votre poste de travail utilise l'encodage Utf-8, attendez-vous donc de nouveau à des résultats étranges :

```
>>> ch ='René'
>>> print len(ch)
5
```

Vous comprenez ce qui se passe : si elle est encodée en *Utf-8*, la chaîne de 4 caractères « René » compte en fait 5 octets, à cause de la lettre accentuée **'é'** (encodée sur deux octets).

- Convertir en nombre véritable une chaîne de caractères qui représente un nombre. Exemple :

```
>>> ch = '8647'
>>> print ch + 45
  →    *** erreur *** : on ne peut pas additionner une chaîne et un nombre
>>> n = int(ch)
>>> print n + 65
8712                    # OK : on peut additionner 2 nombres
```

Dans cet exemple, la fonction intégrée **int()** convertit la chaîne en nombre entier. Il serait également possible de convertir une chaîne en nombre réel, à l'aide de la fonction intégrée **float()**.

Exercices :

5.6 Écrivez un script qui détermine si une chaîne contient ou non le caractère « e ».

5.7 Écrivez un script qui compte le nombre d'occurrences du caractère « e » dans une chaîne.

5.8 Écrivez un script qui recopie une chaîne (dans une nouvelle variable), en insérant des astérisques entre les caractères.
Ainsi par exemple, « **gaston** » devra devenir « **g*a*s*t*o*n** »

5.9 Écrivez un script qui recopie une chaîne (dans une nouvelle variable) en l'inversant.
Ainsi par exemple, « **zorglub** » deviendra « **bulgroz** ».

5.10 En partant de l'exercice précédent, écrivez un script qui détermine si une chaîne de caractères donnée est un *palindrome* (c'est-à-dire une chaîne qui peut se lire indifféremment dans les deux sens), comme par exemple « radar » ou « s.o.s ».

Les listes (première approche)

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de *données composites*. On appelle ainsi les structures de données qui sont utilisées pour regrouper de manière structurée des ensembles de valeurs. Vous apprendrez progressivement à utiliser plusieurs autres types de données composites, parmi lesquelles les *listes*, les *tuples* et les *dictionnaires*.[15] Nous n'allons cependant aborder ici que le premier de ces trois types, et ce de façon assez sommaire. Il s'agit là en effet d'un sujet fort vaste, sur lequel nous devrons revenir à plusieurs reprises.

Sous Python, on peut définir une liste comme *une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets*. Exemple :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print jour
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
```

Dans cet exemple, la valeur de la variable **jour** est une liste.

Comme on peut le constater dans le même exemple, les éléments individuels qui constituent une liste peuvent être de types variés. Dans cet exemple, en effet, les trois premiers éléments sont des chaînes de caractères, le quatrième élément est un entier, le cinquième un réel, etc. (Nous verrons

[15] Vous pourrez même créer vos propres types de données composites, lorsque vous aurez assimilé le concept de ***classe*** (voir page 159).

plus loin qu'un élément d'une liste peut lui-même être une liste !). A cet égard, le concept de liste est donc assez différent du concept de « tableau » (*array*) ou de « variable indicée » que l'on rencontre dans d'autres langages de programmation.

Remarquons aussi que comme les chaînes de caractères, les listes sont des *séquences*, c'est-à-dire des *collections ordonnées d'objets*. Les divers éléments qui constituent une liste sont en effet toujours disposés dans le même ordre, et l'on peut donc accéder à chacun d'entre eux individuellement si l'on connaît son *index* dans la liste. Comme c'était déjà le cas pour les caractères dans une chaîne, il faut cependant retenir que la numérotation de ces index commence *à partir de zéro*, et non à partir de un.

Exemples :

```
>>> jour = ['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> print jour[2]
mercredi
>>> print jour[4]
20.357
```

A la différence de ce qui se passe pour les chaînes, qui constituent un type de données *non-modifiables* (nous aurons plus loin diverses occasions de revenir là-dessus), il est possible de changer les éléments individuels d'une liste :

```
>>> print jour
['lundi', 'mardi', 'mercredi', 1800, 20.357, 'jeudi', 'vendredi']
>>> jour[3] = jour[3] +47
>>> print jour
['lundi', 'mardi', 'mercredi', 1847, 20.357, 'jeudi', 'vendredi']
```

On peut donc remplacer certains éléments d'une liste par d'autres, comme ci-dessous :

```
>>> jour[3] = 'Juillet'
>>> print jour
['lundi', 'mardi', 'mercredi', 'Juillet', 20.357, 'jeudi', 'vendredi']
```

La *fonction intégrée* **len()** , que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste :

```
>>> len(jour)
7
```

Une autre *fonction intégrée* permet de supprimer d'une liste un élément quelconque (à partir de son index). Il s'agit de la fonction **del()** [16] :

```
>>> del(jour[4])
>>> print jour
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi']
```

Il est également tout à fait possible d'ajouter un élément à une liste, mais pour ce faire, il faut considérer que la liste est un *objet*, dont on va utiliser l'une des *méthodes*. Les concepts informatiques *d'objet* et de *méthode* ne seront expliqués qu'un peu plus loin dans ces notes, mais nous pouvons dès à présent montrer « comment ça marche » dans le cas particulier d'une liste :

```
>>> jour.append('samedi')
>>> print jour
['lundi', 'mardi', 'mercredi', 'juillet', 'jeudi', 'vendredi', 'samedi']
>>>
```

[16] Il existe en fait tout un ensemble de techniques qui permettent de découper une liste en tranches, d'y insérer des groupes d'éléments, d'en enlever d'autres, etc., en utilisant une syntaxe particulière où n'interviennent que les index.
Cet ensemble de techniques (qui peuvent aussi s'appliquer aux chaînes de caractères) porte le nom générique de *slicing* (tranchage). On le met en oeuvre en plaçant plusieurs indices au lieu d'un seul entre les crochets que l'on accole au nom de la variable. Ainsi jour[1:3] désigne le sous-ensemble ['mardi', 'mercredi'].
Ces techniques un peu particulières sont décrites plus loin (voir pages 123 et suivantes).

Dans la première ligne de l'exemple ci-dessus, nous avons appliqué la *méthode* **append()** à *l'objet* **jour**, avec *l'argument* 'samedi'. Si l'on se rappelle que le mot « append » signifie « ajouter » en anglais, on peut comprendre que la méthode **append()** est une sorte de *fonction* qui est en quelque manière attachée ou intégrée aux objets du type « liste ». L'argument que l'on utilise avec cette fonction est bien entendu l'élément que l'on veut ajouter à la fin de la liste.

Nous verrons plus loin qu'il existe ainsi toute une série de ces *méthodes* (c'est-à-dire des fonctions intégrées, ou plutôt « encapsulées » dans les objets de type « liste »). Notons simplement au passage que l'on applique une méthode à un objet *en reliant les deux à l'aide d'un point*. (D'abord le nom de la variable qui référence l'objet, puis le point, puis le nom de la méthode, cette dernière toujours accompagnée d'une paire de parenthèses).

Comme les chaînes de caractères, les listes seront approfondies plus loin dans ces notes (voir page 140). Nous en savons cependant assez pour commencer à les utiliser dans nos programmes. Veuillez par exemple analyser le petit script ci-dessous et commenter son fonctionnement :

```
jour = ['dimanche','lundi','mardi','mercredi','jeudi','vendredi','samedi']
a, b = 0, 0
while a<25:
     a = a + 1
     b = a % 7
     print a, jour[b]
```

La 5e ligne de cet exemple fait usage de l'opérateur « *modulo* » déjà rencontré précédemment et qui peut rendre de grands services en programmation. On le représente par % dans de nombreux langages (dont Python). Quelle est l'opération effectuée par cet opérateur ?

Exercices :

5.11 Soient les listes suivantes :

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin',
            'Juillet', 'Août', 'Septembre', 'Octobre', 'Novembre',
'Décembre']
```

Écrivez un petit programme qui crée une nouvelle liste **t3**. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant :

```
['Janvier',31,'Février',28,'Mars',31, etc...].
```

5.12 Écrivez un programme qui affiche « proprement » tous les éléments d'une liste. Si on l'appliquait par exemple à la liste **t2** de l'exercice ci-dessus, on devrait obtenir :

```
Janvier Février Mars Avril Mai Juin Juillet Août Septembre Octobre
Novembre Décembre
```

5.13 Écrivez un programme qui recherche le plus grand élément présent dans une liste donnée. Par exemple, si on l'appliquait à la liste `[32, 5, 12, 8, 3, 75, 2, 15]`, ce programme devrait afficher :

```
le plus grand élément de cette liste a la valeur 75.
```

5.14 Écrivez un programme qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres *pairs* de la liste initiale, et l'autre les nombres *impairs*. Par exemple, si

la liste initiale est celle de l'exercice précédent, le programme devra construire une liste **pairs** qui contiendra `[32, 12, 8, 2]`, et une liste **impairs** qui contiendra `[5, 3, 75, 15]`. Astuce : pensez à utiliser l'opérateur **modulo** (%) déjà cité précédemment.

5.15 Écrivez un programme qui analyse un par un tous les éléments d'une liste de mots (par exemple : `['Jean', 'Maximilien', 'Brigitte', 'Sonia', 'Jean-Pierre', 'Sandra']` pour générer deux nouvelles listes. L'une contiendra les mots comportant moins de 6 caractères, l'autre les mots comportant 6 caractères ou davantage.

6

Fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de *fonction*[17]. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés eux-mêmes en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la ré-écrire à chaque fois.

Interaction avec l'utilisateur : la fonction input()

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc.). Dans un script en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction intégrée **input()**. Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <*Enter*>. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une valeur correspondant à ce que l'utilisateur a entré. Cette valeur peut alors être assignée à une variable quelconque.

On peut invoquer la fonction **input()** en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
print 'Veuillez entrer un nombre positif quelconque : ',
nn = input()
print 'Le carré de', nn, 'vaut', nn**2
```

ou encore :

```
prenom = input('Entrez votre prénom (entre guillemets) : ')
print 'Bonjour,', prenom
```

Remarques importantes :

- La fonction **input()** renvoie une valeur dont le type correspond à ce que l'utilisateur a entré. Dans notre exemple, la variable **nn** contiendra donc un entier, une chaîne de caractères, un réel, etc. suivant ce que l'utilisateur aura décidé. Si l'utilisateur souhaite entrer une chaîne de caractères, il doit l'entrer comme telle, c'est-à-dire *incluse entre des apostrophes ou des guillemets*. Nous verrons plus loin qu'un bon script doit toujours vérifier si le type ainsi entré correspond bien à ce que l'on attend pour la suite du programme.
- Pour cette raison, il sera souvent préférable d'utiliser dans vos scripts la fonction similaire **raw_input()**, laquelle renvoie toujours une chaîne de caractères. Vous pouvez ensuite convertir cette chaîne en nombre à l'aide de **int()** ou de **float()**. Exemple :

[17] Sous Python, le terme de "fonction" est utilisé indifféremment pour désigner à la fois de véritables fonctions mais également des **procédures**. Nous indiquerons plus loin la distinction entre ces deux concepts proches.

```
>>> a = raw_input('Entrez une donnée : ')
Entrez une donnée : 52.37
>>> type(a)
<type 'str'>
>>> b = float(a)   # conversion en valeur numérique
>>> type(b)
<type 'float'>
```

Importer un module de fonctions

Vous avez déjà rencontré des *fonctions intégrées* au langage lui-même, comme la fonction **len()**, par exemple, qui permet de connaître la longueur d'une chaîne de caractères. Il va de soi cependant qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité : vous apprendrez d'ailleurs très bientôt comment en créer vous-même de nouvelles. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des *modules*.

Les modules sont donc des fichiers qui regroupent des ensembles de fonctions. Vous verrez plus loin comme il est commode de découper un programme important en plusieurs fichiers de taille modeste pour en faciliter la maintenance. Une application Python typique sera alors constituée d'un programme principal accompagné de un ou plusieurs modules contenant chacun les définitions d'un certain nombre de fonctions accessoires.

Il existe un grand nombre de modules pré-programmés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées que l'on appelle des *bibliothèques*.

Le module *math*, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que *sinus*, *cosinus*, *tangente*, *racine carrée*, etc. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from math import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant *toutes* les fonctions (c'est là la signification du symbole « joker » *) du module *math*, lequel contient une bibliothèque de fonctions mathématiques pré-programmées.

Dans le corps du script lui-même, vous écrirez par exemple :
`racine = sqrt(nombre)` pour assigner à la variable `racine` la racine carrée de `nombre`,
`sinusx = sin(angle)` pour assigner à la variable `sinusx` le sinus de `angle` (en radians !), etc.

Exemple :

```
# Démo : utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6      # (la bibliothèque math inclut aussi la définition de pi)
print 'racine carrée de', nombre, '=', sqrt(nombre)
print 'sinus de', angle, 'radians', '=', sin(angle)
```

L'exécution de ce script provoque l'affichage suivant :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

Ce court exemple illustre déjà fort bien quelques caractéristiques importantes des fonctions :

- une fonction apparaît sous la forme d'*un nom quelconque associé à des parenthèses*
  exemple : **sqrt()**
- dans les parenthèses, on *transmet* à la fonction un ou plusieurs *arguments*
  exemple : **sqrt(121)**
- la fonction fournit une *valeur de retour* (on dira aussi qu'elle « renvoie » une valeur)
  exemple : **11.0**

Nous allons développer tout ceci dans les pages suivantes. Veuillez noter au passage que les fonctions mathématiques utilisées ici ne représentent qu'un tout premier exemple. Un simple coup d'œil dans la documentation des bibliothèques Python vous permettra de constater que de très nombreuses fonctions sont d'ores et déjà disponibles pour réaliser une multitude de tâches, y compris des algorithmes mathématiques très complexes (Python est couramment utilisé dans les universités pour la résolution de problèmes scientifiques de haut niveau). Il est donc hors de question de fournir ici une liste détaillée. Une telle liste est aisément accessible dans le système d'aide de Python :

Documentation HTML → *Python documentation* → *Modules index* → *math*

Au chapitre suivant, nous apprendrons comment créer nous-mêmes de nouvelles fonctions.

Exercices :

> *(Note : Dans tous ces exercices, utilisez la fonction* ***raw_input()*** *pour l'entrée des données)*

6.1 Écrivez un programme qui convertisse en mètres par seconde et en km/h une vitesse fournie par l'utilisateur en miles/heure. (*Rappel : 1 mile = 1609 mètres*)

6.2 Écrivez un programme qui calcule le périmètre et l'aire d'un triangle quelconque dont l'utilisateur fournit les 3 côtés.
(Rappel : l'aire d'un triangle quelconque se calcule à l'aide de la formule :

$$S=\sqrt{d\cdot(d-a)\cdot(d-b)\cdot(d-c)}$$

dans laquelle d désigne la longueur du demi-périmètre, et a, b, c celles des trois côtés).

6.3 Écrivez un programme qui calcule la période d'un pendule simple de longueur donnée.

La formule qui permet de calculer la période d'un pendule simple est $T=2\pi\sqrt{\frac{l}{g}}$,

l représentant la longueur du pendule et ***g*** la valeur de l'accélération de la pesanteur au lieu d'expérience.

6.4 Écrivez un programme qui permette d'encoder des valeurs dans une liste. Ce programme devrait fonctionner en boucle, l'utilisateur étant invité à entrer sans cesse de nouvelles valeurs, jusqu'à ce qu'il décide de terminer en frappant *<Enter>* en guise d'entrée. Le programme se terminerait alors par l'affichage de la liste. Exemple de fonctionnement :

```
Veuillez entrer une valeur : 25
Veuillez entrer une valeur : 18
Veuillez entrer une valeur : 6284
Veuillez entrer une valeur :
[25, 18, 6284]
```

Un peu de détente avec le module turtle

Comme nous venons de le voir, l'une des grandes qualités de Python est qu'il est extrêmement facile de lui ajouter de nombreuses fonctionnalités par importation de divers *modules*.

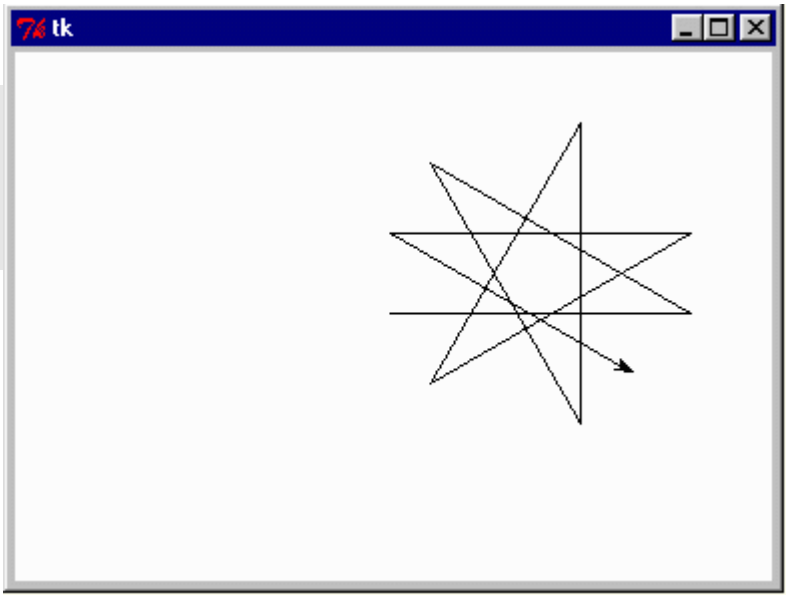
Pour illustrer cela, et nous amuser un peu avec d'autres objets que des nombres, nous allons explorer un module Python qui permet de réaliser des « graphiques tortue », c'est-à-dire des dessins géométriques correspondant à la piste laissée derrière elle par une petite « tortue » virtuelle, dont nous contrôlons les déplacements sur l'écran de l'ordinateur à l'aide d'instructions simples.

Activer cette tortue est un vrai jeu d'enfant. Plutôt que de vous donner de longues explications, nous vous invitons à essayer tout de suite :

```
>>> from turtle import *
>>> forward(120)
>>> left(90)
>>> color('red')
>>> forward(80)
```

L'exercice est évidemment plus riche si l'on utilise des boucles :

```
>>> reset()
>>> a = 0
>>> while a <12:
        a = a +1
        forward(150)
        left(150)
```



Attention cependant : avant de lancer un tel script, assurez-vous toujours qu'il ne comporte pas de boucle sans fin (voir page 39), car si c'est le cas vous risquez de ne plus pouvoir reprendre le contrôle des opérations (en particulier sous *Windows*).

Amusez-vous à écrire des scripts qui réalisent des dessins suivant un modèle imposé à l'avance. Les principales fonctions mises à votre disposition dans le module *turtle* sont les suivantes :

reset() On efface tout et on recommence
goto(x, y) Aller à l'endroit de coordonnées x, y
forward(distance) Avancer d'une distance donnée
backward(distance) Reculer
up() Relever le crayon (pour pouvoir avancer sans dessiner)
down() Abaisser le crayon(pour recommencer à dessiner)
color(couleur) *couleur* peut être une chaîne prédéfinie ('red', 'blue', etc.)
left(angle) Tourner à gauche d'un angle donné (exprimé en degrés)
right(angle) Tourner à droite
width(épaisseur) Choisir l'épaisseur du tracé
fill(1) Remplir un contour fermé à l'aide de la couleur sélectionnée
write(texte) *texte* doit être une chaîne de caractères

Véracité/fausseté d'une expression

Lorsqu'un programme contient des instructions telles que **while** ou **if**, l'ordinateur qui exécute ce programme doit évaluer la véracité d'une condition, c'est-à-dire déterminer si une expression est *vraie* ou *fausse*. Par exemple, une boucle initiée par `while c<20:` s'exécutera aussi longtemps que la condition `c<20` restera *vraie*.

Mais comment un ordinateur peut-il déterminer si quelque chose est *vrai* ou *faux* ?

En fait - et vous le savez déjà - un ordinateur ne manipule strictement que des nombres. Tout ce qu'un ordinateur doit traiter doit d'abord toujours être converti en valeur numérique. Cela s'applique aussi à la notion de vrai/faux. En Python, tout comme en *C*, en *Basic* et en de nombreux autres langages de programmation, on considère que toute valeur numérique autre que zéro est « vraie ». Seule la valeur zéro est « fausse ». Exemple :

```
a = input('Entrez une valeur quelconque')
if a:
    print "vrai"
else:
    print "faux"
```

Le petit script ci-dessus n'affiche « faux » que si vous entrez la valeur 0. Pour toute autre valeur numérique, vous obtiendrez « vrai ».

Si vous entrez une chaîne de caractères ou une liste, vous obtiendrez encore « vrai ». Seules les chaînes ou les listes *vides* seront considérées comme « fausses ».

Tout ce qui précède signifie donc qu'une expression à évaluer, telle par exemple la condition `a > 5` , est d'abord convertie par l'ordinateur en une valeur numérique. (Généralement **1** si l'expression est vraie, et **zéro** si l'expression est fausse). Exemple :

```
a = input('entrez une valeur numérique : ')
b = (a < 5)
print 'la valeur de b est', b, ':'
if b:
    print "la condition b est vraie"
else:
    print "la condition b est fausse"
```

Le script ci-dessus vous renvoie une valeur b = 1 (condition vraie) si vous avez entré un nombre plus petit que 5.

Ces explications ne sont qu'une première information à propos d'un système de représentation des opérations logiques que l'on appelle *algèbre de Boole*. Vous apprendrez plus loin que l'on peut appliquer aux nombres binaires des opérateurs tels que **and**, **or**, **not**, etc. qui permettent d'effectuer à l'aide de ces nombres des traitements logiques complexes.

Révision

Dans ce qui suit, nous n'allons pas apprendre de nouveaux concepts mais simplement utiliser tout ce que nous connaissons déjà pour réaliser de vrais petits programmes.

Contrôle du flux - Utilisation d'une liste simple

Commençons par un petit retour sur les branchements conditionnels (il s'agit peut-être là du groupe d'instructions le plus important, dans n'importe quel langage !) :

```
# Utilisation d'une liste et de branchements conditionnels
```

```
print "Ce script recherche le plus grand de trois nombres"
print 'Veuillez entrer trois nombres séparés par des virgules : '
# Note : la fonction list() convertit en liste la séquence de données qu'on

# lui fournit en argument. L'instruction ci-dessous convertira donc les
# données fournies par l'utilisateur en une liste  nn :

nn = list(input())
max, index = nn[0], 'premier'
if nn[1] > max:      # ne pas omettre le double point !
    max = nn[1]
    index = 'second'
if nn[2] > max:
    max = nn[2]
    index = 'troisième'
print "Le plus grand de ces nombres est", max
print "Ce nombre est le", index, "de votre liste."
```

Note : Dans cet exercice, vous retrouvez à nouveau le concept de « bloc d'instructions », déjà abondamment commenté aux chapitres 3 et 4, et que vous devez absolument assimiler. Pour rappel, les blocs d'instructions sont délimités par *l'indentation*. Après la première instruction **if**, par exemple, il y a deux lignes indentées définissant un bloc d'instructions. Ces instructions ne seront exécutées que si la condition `nn[1] > max` est vraie.

La ligne suivante, par contre (celle qui contient la deuxième instruction **if**) n'est pas indentée. Cette ligne se situe donc au même niveau que celles qui définissent le corps principal du programme. L'instruction contenue dans cette ligne est donc toujours exécutée, alors que les deux suivantes (qui constituent encore un autre bloc) ne sont exécutées que si la condition `nn[2] > max` est vraie.

En suivant la même logique, on voit que les instructions des deux dernières lignes font partie du bloc principal et sont donc toujours exécutées.

Boucle while - Instructions imbriquées

Continuons dans cette voie en imbriquant d'autres structures :

```
  1#  # Instructions composées <while> - <if> - <elif> - <else>
  2#
  3#  print 'Choisissez un nombre de 1 à 3 (ou zéro pour terminer) ',
  4#  a = input()
  5#  while a != 0:           # l'opérateur != signifie "différent de"
  6#      if a == 1:
  7#          print "Vous avez choisi un :"
  8#          print "le premier, l'unique, l'unité ..."
  9#      elif a == 2:
 10#          print "Vous préférez le deux :"
 11#          print "la paire, le couple, le duo ..."
 12#      elif a == 3:
 13#          print "Vous optez pour le plus grand des trois :"
 14#          print "le trio, la trinité, le triplet ..."
 15#      else :
 16#          print "Un nombre entre UN et TROIS, s.v.p."
 17#      print 'Choisissez un nombre de 1 à 3 (ou zéro pour terminer) ',
 18#      a = input()
 19#  print "Vous avez entré zéro :"
 20#  print "L'exercice est donc terminé."
```

Nous retrouvons ici une boucle **while**, associée à un groupe d'instructions **if**, **elif** et **else**. Notez bien cette fois encore comment la structure logique du programme est créée à l'aide des indentations (... et n'oubliez pas le caractère « : » à la fin de chaque ligne d'en-tête !)

L'instruction **while** est utilisée ici pour relancer le questionnement après chaque réponse de l'utilisateur (du moins jusqu'à ce que celui-ci décide de « quitter » en entrant une valeur nulle : rappelons à ce sujet que l'opérateur de comparaison **!=** signifie « est différent de »).

Dans le corps de la boucle, nous trouvons le groupe d'instructions **if**, **elif** et **else** (de la ligne 6 à la ligne 16), qui aiguille le flux du programme vers les différentes réponses, ensuite une instruction **print** et une instruction **input()** (lignes 17 & 18) qui seront exécutées dans tous les cas de figure : notez bien leur niveau d'indentation, qui est le même que celui du bloc **if**, **elif** et **else**. Après ces instructions, le programme boucle et l'exécution reprend à l'instruction **while** (ligne 5).

Les deux dernières instructions **print** (lignes 19 & 20) ne sont exécutées qu'à la sortie de la boucle.

Exercices

6.5 Que fait le programme ci-dessous, dans les quatre cas où l'on aurait défini au préalable que la variable **a** vaut 1, 2, 3 ou 15 ?

```
if a !=2:
    print 'perdu'
elif a ==3:
    print 'un instant, s.v.p.'
else :
    print 'gagné'
```

6.6 Que font ces programmes ?

```
a)   a = 5
     b = 2
     if (a==5) & (b<2):
         print '"&" signifie "et"; on peut aussi utiliser\
                le mot "and"'
b)   a, b = 2, 4
     if (a==4) or (b!=4):
         print 'gagné'
     elif (a==4) or (b==4):
         print 'presque gagné'
c)   a = 1
     if not a:
         print 'gagné'
     elif a:
         print 'perdu'
```

6.7 Reprendre le programme c) avec a = 0 au lieu de a = 1.
Que se passe-t-il ? Conclure !

6.8 Écrire un programme qui, étant données deux bornes entières a et b, additionne les nombres multiples de 3 et de 5 compris entre ces bornes.
Prendre par exemple a = 0, b = 32 → le résultat devrait être alors 0 + 15 + 30 = 45.
Modifier légèrement ce programme pour qu'il additionne les nombres multiples de 3 ou de 5 compris entre les bornes a et b. Avec les bornes 0 et 32, le résultat devrait donc être : 0 + 3 + 5 + 6 + 9 + 10 + 12 + 15 + 18 + 20 + 21 + 24 + 25 + 27 + 30 = 225.

6.9 Déterminer si une année (dont le millésime est introduit par l'utilisateur) est bissextile ou non. (Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400).

6.10 Demander à l'utilisateur son nom et son sexe (M ou F). En fonction de ces données, afficher « Cher Monsieur » ou « Chère Mademoiselle » suivi du nom de la personne.

6.11 Demander à l'utilisateur d'entrer trois longueurs a, b, c. A l'aide de ces trois longueurs, déterminer s'il est possible de construire un triangle. Déterminer ensuite si ce triangle est rectangle, isocèle, équilatéral ou quelconque. Attention : un triangle rectangle peut être isocèle.

6.12 Demander à l'utilisateur qu'il entre un nombre. Afficher ensuite : soit la racine carrée de ce nombre, soit un message indiquant que la racine carrée de ce nombre ne peut être calculée.

6.13 Convertir une note scolaire N quelconque, entrée par l'utilisateur sous forme de points (par exemple 27 sur 85), en une note standardisée suivant le code suivant :

```
    Note            Appréciation
N >= 80 %               A
80 % > N >= 60 %        B
60 % > N >= 50 %        C
50 % > N >= 40 %        D
N < 40 %                E
```

6.14 Soit la liste suivante :

```
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', Maximilien',
 'Alexandre-Benoît', 'Louise']
```

Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant.

6.15 Écrire une boucle de programme qui demande à l'utilisateur d'entrer des notes d'élèves. La boucle se terminera seulement si l'utilisateur entre une valeur négative. Avec les notes ainsi entrées, construire progressivement une liste. Après chaque entrée d'une nouvelle note (et donc à chaque itération de la boucle), afficher le nombre de notes entrées, la note la plus élevée, la note la plus basse, la moyenne de toutes les notes.

6.16 Écrivez un script qui affiche la valeur de la force de gravitation s'exerçant entre deux masses de 10000 kg , pour des distances qui augmentent suivant une progression géométrique de raison 2, à partir de 5 cm (0,05 mètre).

La force de gravitation est régie par la formule $F = 6{,}67\,10^{-11} \cdot \frac{m \cdot m'}{d^2}$

Exemple d'affichage :

```
d = .05 m :  la force vaut  2.668 N
d = .1 m  :  la force vaut  0.667 N
d = .2 m  :  la force vaut  0.167 N
d = .4 m  :  la force vaut  0.0417 N
```

etc.

7

Fonctions originales

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de fonctions originales.

Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (Ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes[18] pour que ceux-ci restent clairs.

D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

Les *fonctions*[19] et les *classes d'objets* sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la *définition de fonctions* sous Python. Les *objets* et les *classes* seront examinés plus loin.

Nous avons déjà rencontré diverses fonctions pré-programmées. Voyons à présent comment en définir nous-mêmes de nouvelles.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage[20], et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis). Comme c'est le cas pour les noms de variables, il vous

[18] On appelle algorithme la séquence détaillée de toutes les opérations à effectuer pour résoudre un problème.

[19] Il existe aussi dans d'autres langages des **routines** (parfois appelés sous-programmes) et des **procédures**.
Il n'existe pas de **routines** en Python. Quant au terme de **fonction**, il désigne à la fois les fonctions au sens strict (qui fournissent une valeur en retour), et les procédures (qui n'en fournissent pas).

[20] La liste complète des mots réservés Python se trouve page 24.

est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux *classes* que nous étudierons plus loin).

- Comme les instructions **if** et **while** que vous connaissez déjà, l'instruction **def** est une *instruction composée*. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier *d'indenter*.
- La *liste de paramètres* spécifie quelles informations il faudra fournir en guise *d'arguments* lorsque l'on voudra utiliser cette fonction (Les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments).
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un *appel de fonction* est constitué du nom de la fonction suivi de parenthèses.
  Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut (voir plus loin).

Fonction simple sans paramètres

Pour notre première approche concrète des fonctions, nous allons travailler à nouveau en mode interactif. Le mode interactif de Python est en effet idéal pour effectuer des petits tests comme ceux qui suivent. C'est une facilité que n'offrent pas tous les langages de programmation !

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print n * 7,
...         n = n +1
...
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses[21], le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom. Ainsi :

```
>>> table7()
```

provoque l'affichage de :

```
7 14 21 28 35 42 49 56 63 70
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
>>> def table7triple():
...     print 'La table par 7 en triple exemplaire :'
...     table7()
...     table7()
...     table7()
...
```

Utilisons cette nouvelle fonction, en entrant la commande :

[21] Un nom de fonction doit toujours être accompagné de parenthèses, même si la fonction n'utilise aucun paramètre. Il en résulte une convention d'écriture qui stipule que dans un texte quelconque traitant de programmation d'ordinateur, un nom de fonction soit toujours accompagné d'une paire de parenthèses vides. Nous respecterons cette convention dans la suite de ce texte.

```
>>> table7triple()
```

l'affichage résultant devrait être :

```
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Au stade où nous sommes, vous ne voyez peut-être pas encore très bien l'utilité de tout cela, mais vous pouvez déjà noter deux propriétés intéressantes :

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent. Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

Fonction avec paramètre

Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un ***argument***. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments. La fonction **sin(a)**, par exemple, calcule le sinus de l'angle **a**. La fonction **sin()** utilise donc la valeur numérique de **a** comme argument pour effectuer son travail.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un ***paramètre***. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

Voici ce que cela donne dans le cas qui nous intéresse :

```
>>> def table(base):
...     n = 1
...     while n <11 :
...        print n * base,
...        n = n +1
```

La fonction **table()** telle que définie ci-dessus utilise le paramètre **base** pour calculer les dix premiers termes de la table de multiplication correspondante.

Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument. Exemples :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130

>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Dans ces exemples, la valeur que nous indiquons entre parenthèses lors de l'appel de la fonction (et qui est donc un argument) est automatiquement affectée au paramètre **base**. Dans le corps de la fonction, **base** joue le même rôle que n'importe quelle autre variable. Lorsque nous entrons la commande `table(9)`, nous signifions à la machine que nous voulons exécuter la fonction **table()** en affectant la valeur `9` à la variable **base**.

Utilisation d'une variable comme argument

Dans les 2 exemples qui précèdent, l'argument que nous avons utilisé en appelant la fonction **table()** était à chaque fois une constante (la valeur 13, puis la valeur 9). Cela n'est nullement obligatoire. *L'argument que nous utilisons dans l'appel d'une fonction peut être une variable* lui aussi, comme dans l'exemple ci-dessous. Analysez bien cet exemple, essayez-le concrètement, et décrivez le mieux possible dans votre cahier d'exercices ce que vous obtenez, en expliquant avec vos propres mots ce qui se passe. Cet exemple devrait vous donner un premier aperçu de l'utilité des fonctions pour accomplir simplement des tâches complexes :

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...
```

Remarque importante :

Dans l'exemple ci-dessus, l'argument que nous passons à la fonction **table()** est le contenu de la variable `a` . À l'intérieur de la fonction, cet argument est affecté au paramètre **base**, qui est une tout autre variable. Notez donc bien dès à présent que :

> *Le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction.*

Ces noms peuvent être identiques si vous le voulez, mais vous devez bien comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent contenir une valeur identique).

7 ***Exercice*** :

7.1 Importez le module *turtle* pour pouvoir effectuer des dessins simples.

Vous allez dessiner une série de triangles équilatéraux de différentes couleurs.
Pour ce faire, définissez d'abord une fonction **triangle()** capable de dessiner un triangle d'une couleur bien déterminée (ce qui signifie donc que la définition de votre fonction doit comporter un paramètre pour recevoir le nom de cette couleur)

Utilisez ensuite cette fonction pour reproduire ce même triangle en différents endroits, en changeant de couleur à chaque fois.

Fonction avec plusieurs paramètres

La fonction **table()** est certainement intéressante, mais elle n'affiche toujours que les dix premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche

d'autres. Qu'à cela ne tienne. Nous allons l'améliorer en lui ajoutant des paramètres supplémentaires, dans une nouvelle version que nous appellerons cette fois **tableMulti()** :

```
>>> def tableMulti(base, debut, fin):
...     print 'Fragment de la table de multiplication par', base, ':'
...     n = debut
...     while n <= fin :
...         print n, 'x', base, '=', n * base
...         n = n +1
```

Cette nouvelle fonction utilise donc trois paramètres : la base de la table comme dans l'exemple précédent, l'indice du premier terme à afficher, l'indice du dernier terme à afficher.

Essayons cette fonction en entrant par exemple :

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Notes :

- Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.
- Lors de l'appel de la fonction, les arguments utilisés doivent être fournis dans le même ordre que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.
- A titre d'exercice, essayez la séquence d'instructions suivantes et décrivez dans votre cahier d'exercices le résultat obtenu :

```
>>> t, d, f = 11, 5, 10
>>> while t<21:
...     tableMulti(t,d,f)
...     t, d, f = t +1, d +3, f +5
...
```

Variables locales, variables globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des *variables locales* à la fonction. C'est par exemple le cas des variables **base**, **debut**, **fin** et **n** dans l'exercice précédent.

Chaque fois que la fonction **tableMulti()** est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel *espace de noms*[22]. Les contenus des variables **base**, **debut**, **fin** et **n** sont stockés dans cet espace de noms qui est *inaccessible depuis l'extérieur de la fonction*. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable **base** juste après avoir effectué l'exercice ci-dessus, nous obtenons un message d'erreur :

```
>>> print base
Traceback (innermost last):
```

[22] Ce concept d'*espace de noms* sera approfondi progressivement. Vous apprendrez également plus loin que les fonctions sont en fait des *objets* dont on crée à chaque fois une nouvelle *instance* lorsqu'on les appelle.

```
  File "<pyshell#8>", line 1, in ?
    print base
NameError: base
```

La machine nous signale clairement que le symbole **base** lui est inconnu, alors qu'il était correctement imprimé par la fonction **tableMulti()** elle-même. L'espace de noms qui contient le symbole **base** est strictement réservé au fonctionnement interne de **tableMulti()**, et il est automatiquement détruit dès que la fonction a terminé son travail.

Les variables définies à l'extérieur d'une fonction sont des *variables globales*. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction *ne peut pas le modifier*. Exemple :

```
>>> def mask():
...     p = 20
...     print p, q
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print p, q
15 38
```

Analysons attentivement cet exemple :

Nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). A l'intérieur de cette fonction, une variable **p** est définie, avec **20** comme valeur initiale. Cette variable **p** qui est définie à l'intérieur d'une fonction sera donc une *variable locale*.

Une fois terminée la définition de la fonction, nous revenons au niveau principal pour y définir les deux variables **p** et **q** auxquelles nous attribuons les contenus **15** et **38**. Ces deux variables définies au niveau principal seront donc des *variables globales*.

Ainsi le même nom de variable **p** a été utilisé ici à deux reprises, *pour définir deux variables différentes* : l'une est globale et l'autre est locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction (où elles pourraient entrer en compétition), ce sont les variables définies localement qui ont la priorité.

On constate en effet que lorsque la fonction **mask()** est lancée, la variable globale **q** y est accessible, puisqu'elle est imprimée correctement. Pour **p**, par contre, c'est la valeur attribuée localement qui est affichée.

On pourrait croire d'abord que la fonction **mask()** a simplement modifié le contenu de la variable globale **p** (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction **mask()**, la variable globale **p** conserve sa valeur initiale.

Tout ceci peut vous paraître compliqué au premier abord. Vous comprendrez cependant très vite combien il est utile que des variables soient ainsi définies comme étant locales, c'est-à-dire en quelque sorte confinées à l'intérieur d'une fonction. Cela signifie en effet que vous pourrez toujours utiliser quantités de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs.

Cet état de choses peut toutefois être modifié si vous le souhaitez. Il peut se faire par exemple que vous ayez à définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction **global**. Cette instruction permet d'indiquer - à l'intérieur de la définition d'une fonction - quelles sont les variables à traiter globalement.

Dans l'exemple ci-dessous, la variable a utilisée à l'intérieur de la fonction **monter()** est non seulement accessible, mais également modifiable, parce qu'elle est signalée explicitement comme

étant une variable qu'il faut traiter globalement. Par comparaison, essayez le même exercice en supprimant l'instruction **global** : la variable **a** n'est plus incrémentée à chaque appel de la fonction.

```
>>> def monter():
...     global a
...     a = a+1
...     print a
...
>>> a = 15
>>> monter()
16
>>> monter()
17
>>>
```

« Vraies » fonctions et procédures

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des *procédures*[23]. Une « vraie » fonction (au sens strict) doit en effet *renvoyer une valeur* lorsqu'elle se termine. Une « vraie » fonction peut s'utiliser à la droite du signe égale dans des expressions telles que `y = sin(a)`. On comprend aisément que dans cette expression, la fonction **sin()** renvoie une valeur (le sinus de l'argument **a**) qui est directement affectée à la variable **y**.

Commençons par un exemple extrêmement simple :

```
>>> def cube(w):
...    return w*w*w
...
```

L'instruction **return** définit ce que doit être la valeur renvoyée par la fonction. En l'occurrence, il s'agit du cube de l'argument qui a été transmis lors de l'appel de la fonction. Exemple :

```
>>> b = cube(9)
>>> print b
729
```

A titre d'exemple un peu plus élaboré, nous allons maintenant modifier quelque peu la fonction **table()** sur laquelle nous avons déjà pas mal travaillé, afin qu'elle renvoie elle aussi une valeur. Cette valeur sera en l'occurrence une liste (la liste des dix premiers termes de la table de multiplication choisie). Voilà donc une occasion de reparler des listes. Dans la foulée, nous en profiterons pour apprendre encore un nouveau concept :

```
>>> def table(base):
...    result = []            # result est d'abord une liste vide
...    n = 1
...    while n < 11:
...       b = n * base
...       result.append(b)    # ajout d'un terme à la liste
...       n = n +1            # (voir explications ci-dessous)
...    return result
...
```

Pour tester cette fonction, nous pouvons entrer par exemple :

```
>>> ta9 = table(9)
```

Ainsi nous affectons à la variable **ta9** les dix premiers termes de la table de multiplication par 9, sous la forme d'une liste :

[23] Dans certains langages de programmation, les fonctions et les procédures sont définies à l'aide d'instructions différentes. Python utilise la même instruction **def** pour définir les unes et les autres.

```
>>> print ta9
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print ta9[0]
9
>>> print ta9[3]
36
>>> print ta9[2:5]
[27, 36, 45]
>>>
```

(Rappel : le premier élément d'une liste correspond à l'indice 0)

Notes :

- Comme nous l'avons vu dans l'exemple précédent, l'instruction **return** définit ce que doit être la valeur « renvoyée » par la fonction. En l'occurrence, il s'agit ici du contenu de la variable **result**, c'est-à-dire la liste des nombres générés par la fonction[24].
- L'instruction **result.append(b)** est notre second exemple de l'utilisation d'un concept important sur lequel nous reviendrons encore abondamment par la suite : dans cette instruction, nous appliquons la méthode **append()** à l'objet **result**.

  Nous préciserons petit à petit ce qu'il faut entendre par *objet* en programmation. Pour l'instant, admettons simplement que ce terme très général s'applique notamment aux *listes* de Python.
  Une *méthode* n'est en fait rien d'autre qu'une fonction (que vous pouvez d'ailleurs reconnaître comme telle à la présence des parenthèses), mais *une fonction qui est associée à un objet*.
  Elle fait partie de la définition de cet objet, ou plus précisément de la *classe* particulière à laquelle cet objet appartient (nous étudierons ce concept de classe plus tard).

  Mettre en œuvre une méthode associée à un objet consiste en quelque sorte à « faire fonctionner » cet objet d'une manière particulière. Par exemple, on met en œuvre la méthode **methode4()** d'un objet **objet3**, à l'aide d'une instruction du type : **objet3.methode4()** , c'est-à-dire le nom de l'objet, puis le nom de la méthode, reliés l'un à l'autre par un point. Ce point joue un rôle essentiel : on peut le considérer comme un véritable *opérateur*.

  Dans notre exemple, nous appliquons donc la méthode **append()** à l'objet **result**. Sous Python, les *listes* constituent un type particulier d'objets, auxquels on peut effectivement appliquer toute une série de méthodes. En l'occurrence, la méthode **append()** est donc une fonction spécifique des listes, qui sert à leur ajouter un élément par la fin. L'élément à ajouter est transmis entre les parenthèses, comme tout argument qui se respecte.

  Remarque :
  Nous aurions obtenu un résultat similaire si nous avions utilisé à la place de cette instruction une expression telle que « `result = result + [b]` ». Cette façon de procéder est cependant moins rationnelle et beaucoup moins efficace, car elle consiste à redéfinir à chaque itération de la boucle une nouvelle liste **result**, dans laquelle la totalité de la liste précédente est à chaque fois recopiée avec ajout d'un élément supplémentaire.
  Lorsque l'on utilise la méthode **append()**, par contre, l'ordinateur procède bel et bien à une modification de la liste existante (sans la recopier dans une nouvelle variable). Cette technique est donc préférable, car elle mobilise moins lourdement les ressources de l'ordinateur et elle est plus rapide (surtout lorsqu'il s'agit de traiter des listes volumineuses).

[24] **return** peut également être utilisé sans aucun argument, à l'intérieur d'une fonction, pour provoquer sa fermeture immédiate. La valeur retournée dans ce cas est l'objet **None** (objet particulier, correspondant à "rien").

- Il n'est pas du tout indispensable que la valeur renvoyée par une fonction soit affectée à une variable (comme nous l'avons fait jusqu'ici dans nos exemples par souci de clarté).
  Ainsi, nous aurions pu tester les fonction **cube()** et **table()** en entrant les commandes :
  ```
  >>> print cube(9)
  >>> print table(9)
  >>> print table(9)[3]
  ```
  ou encore plus simplement encore :
  ```
  >>> cube(9)
  ```
  ... etc.

Utilisation des fonctions dans un script

Pour cette première approche des fonctions, nous n'avons utilisé jusqu'ici que le mode interactif de l'interpréteur Python.

Il est bien évident que les fonctions peuvent aussi s'utiliser dans des scripts. Veuillez donc essayer vous-même le petit programme ci-dessous, lequel calcule le volume d'une sphère à l'aide de la formule que vous connaissez certainement : $V = \frac{4}{3}\pi R^3$

```
def cube(n):
   return n**3

def volumeSphere(r):
   return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print 'Le volume de cette sphère vaut', volumeSphere(r)
```

Notes :

A bien y regarder, ce programme comporte trois parties : les deux fonctions **cube()** et **volumeSphere()**, et ensuite le corps principal du programme.

Dans le corps principal du programme, il y a un appel de la fonction **volumeSphere()**.

A l'intérieur de la fonction **volumeSphere()**, il y a un appel de la fonction **cube()**.

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : *d'abord la définition des fonctions, et ensuite le corps principal du programme.* Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source.

> *Dans un script, la définition des fonctions doit donc précéder leur utilisation.*

Pour vous en convaincre, intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début), et prenez note du type de message d'erreur qui est affiché lorsque vous essayez d'exécuter le script ainsi modifié.

En fait, le corps principal d'un programme Python constitue lui-même une entité un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé **__main__** (le mot « main » signifie « principal », en anglais. Il est encadré par des caractères « souligné » en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première instruction de cette entité **__main__**, où qu'elle puisse se trouver dans le listing. Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jusqu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu.

Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction avant l'entité **__main__**, et celle-ci sera donc placée en général à la fin du script.

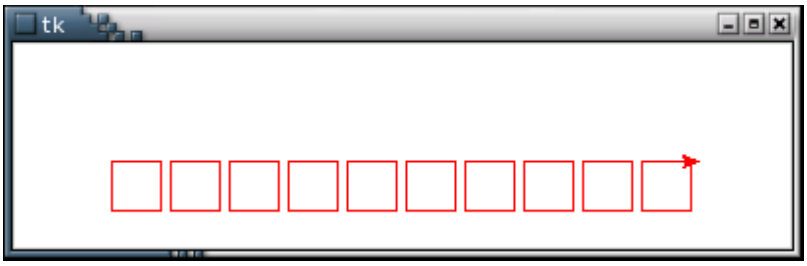
Dans notre exemple, l'entité **__main__** appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première à la dernière ligne, mais plutôt en suivant un cheminement analogue à ce qui se passe lors de l'exécution de ce script. Cela signifie donc concrètement que vous devrez souvent analyser un script en commençant par ses dernières lignes !

Modules de fonctions

Afin que vous puissiez mieux comprendre encore la distinction entre la définition d'une fonction et son utilisation au sein d'un programme, nous vous suggérons de placer fréquemment vos définitions de fonctions dans un module Python, et le programme qui les utilise dans un autre.

Exemple :

On souhaite réaliser la série de dessins ci-dessous, à l'aide du module **turtle** :



Écrivez les lignes de code suivantes, et sauvegardez-les dans un fichier auquel vous donnerez le nom ***dessins_tortue.py*** :

```
from turtle import *

def carre(taille, couleur):
    "fonction qui dessine un carré de taille et de couleur déterminées"
    color(couleur)
    c =0
    while c <4:
        forward(taille)
        right(90)
        c = c +1
```

Vous pouvez remarquer que la définition de la fonction **carre()** commence par une chaîne de caractères. Cette chaîne ne joue aucun rôle fonctionnel dans le script : elle est traitée par Python *comme un simple commentaire*, mais qui est mémorisé à part dans un système de documentation interne automatique, lequel pourra ensuite être exploité par certains utilitaires et éditeurs "intelligents".

Si vous programmez dans l'environnement IDLE, par exemple, vous verrez apparaître cette chaîne documentaire dans une "bulle d'aide", chaque fois que vous ferez appel aux fonctions ainsi documentées.

En fait, Python place cette chaîne dans une variable spéciale dont le nom est **__doc__** (le mot "doc" entouré de deux paires de caractères "souligné"), et qui est associée à l'objet fonction comme étant l'un de ses attributs (Vous en apprendrez davantage au sujet de ces attributs lorsque nous aborderons les classes d'objets. Cf. page 162).

Ainsi, vous pouvez vous-même retrouver la chaîne de documentation d'une fonction quelconque en affichant le contenu de cette variable. Exemple :

```
>>> def essai():
...     "Cette fonction est bien documentée mais ne fait presque rien."
...     print "rien à signaler"

>>> essai()
rien à signaler

>>> print essai.__doc__
Cette fonction est bien documentée mais ne fait presque rien.
```

Prenez donc la peine d'incorporer une telle chaîne explicative dans toutes vos définitions de fonctions futures : il s'agit là d'une pratique hautement recommandable.

Le fichier que vous aurez créé ainsi est dorénavant un véritable *module de fonctions* Python, au même titre que les modules *turtle* ou *math* que vous connaissez déjà. Vous pouvez donc l'utiliser dans n'importe quel autre script, comme celui-ci, par exemple, qui effectuera le travail demandé :

```
from dessins_tortue import *

up()                    # relever le crayon
goto(-150, 50)          # reculer en haut à gauche

# dessiner dix carrés rouges, alignés :
i = 0
while i < 10:
    down()              # abaisser le crayon
    carre(25, 'red')    # tracer un carré
    up()                # relever le crayon
    forward(30)         # avancer + loin
    i = i +1

a = input()             # attendre
```

> ***Attention :*** *vous pouvez à priori nommer vos modules de fonctions comme bon vous semble. Sachez cependant qu'il vous sera impossible d'importer un module si son nom est l'un des 29 mots réservés Python signalés à la page 24, car le nom du module importé deviendrait une variable dans votre script, et les mots réservés ne peuvent pas être utilisés comme noms de variables. Rappelons aussi qu'il vous faut éviter de donner à vos modules - et à tous vos scripts en général - le même nom que celui d'un module Python préexistant, sinon vous devez vous attendre à des conflits. Par exemple, si vous donnez le nom* ***turtle.py*** *à un exercice dans lequel vous avez placé une instruction d'importation du module* ***turtle****, c'est l'exercice lui-même que vous allez importer !*

Résumé : Structure d'un programme Python type

```
###################################
# Programme Python type           #
# auteur : G.Swinnen, Liège, 2003 #
# licence : GPL                   #
###################################


###################################
# Importation de fonctions externes :

from math import sqrt


#################################
# Définition locale de fonctions :


def occurrences(car, ch):
    "nombre de caractères <car> \
     dans la chaîne <ch>"

    nc = 0

    i = 0

    while i < len(ch):

        if ch[i] == car:
            nc = nc + 1

        i = i + 1

    return nc


#############################
# Corps principal du programme :

print "Veuillez entrer un nombre : "
nbr = input()

print "Veuillez entrer une phrase : ",
phr = raw_input()
print "Entrez le caractère à compter : ",
cch = raw_input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print "La racine carrée du cube",
print "du nombre fourni vaut",
print rc
print "La phrase contient",
print no, "caractères", cch
```

Un programme Python contient en général les blocs suivants, dans l'ordre :
- Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales)
- Les définitions locales de fonctions et/ou de classes
- Le corps principal du programme.

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes. (Vous pouvez vous-même définir de tels modules).

La définition d'une fonction comporte souvent une liste de paramètres : ce sont toujours des variables, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit en principe inclure les 4 éléments suivants :
- l'initialisation d'une variable 'compteur' ;
- l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;
- le bloc d'instructions à répéter ;
- une instruction d'incrémentation du compteur.

La fonction 'renvoie' toujours une valeur bien déterminée au programme appelant. (Si l'instruction return n'est pas utilisée, ou si elle est utilisée sans argument,la fonction renvoie un objet vide : <None>)

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'arguments, lesquels peuvent être des valeurs, des variables, ou même des expressions.

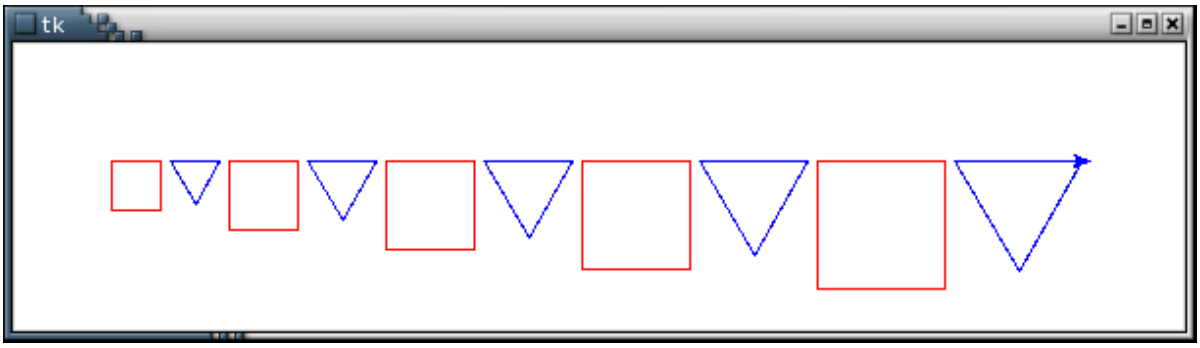
7.2 Définissez une fonction **ligneCar(n, ca)** qui renvoie une chaîne de **n** caractères **ca**.

7.3 Définissez une fonction **surfCercle(R)**. Cette fonction doit renvoyer la surface (l'aire) d'un cercle dont on lui a fourni le rayon **R** en argument. Par exemple, l'exécution de l'instruction :
`print surfCercle(2.5)` doit donner le résultat `19.635`

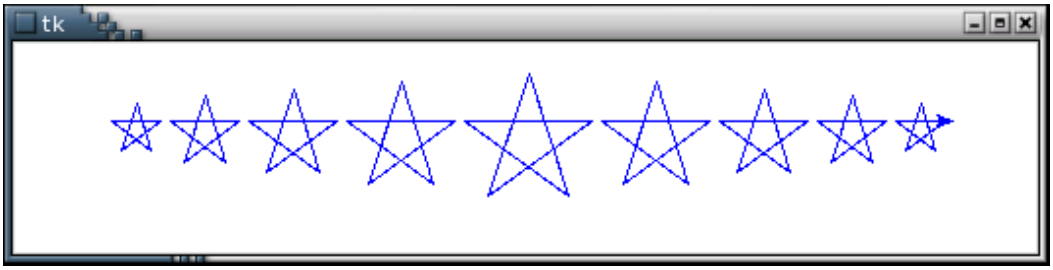
7.4 Définissez une fonction **volBoite(x1,x2,x3)** qui renvoie le volume d'une boîte parallélipipédique dont on fournit les trois dimensions **x1, x2, x3** en arguments.
Par exemple, l'exécution de l'instruction :
`print volBoite(5.2, 7.7, 3.3)` doit donner le résultat : `132.13`

7.5 Définissez une fonction **maximum(n1,n2,n3)** qui renvoie le plus grand de 3 nombres **n1, n2, n3** fournis en arguments. Par exemple, l'exécution de l'instruction :
`print maximum(2,5,4)` doit donner le résultat : `5`

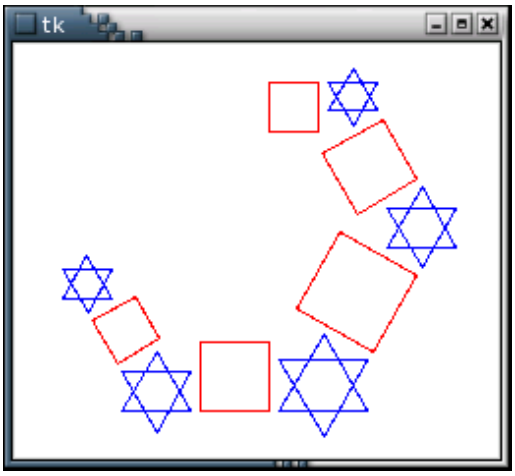
7.6 Complétez le module de fonctions graphiques **dessins_tortue.py** décrit à la page 76.
Commencez par ajouter un paramètre **angle** à la fonction **carre()**, de manière à ce que les carrés puissent être tracés dans différentes orientations.
Définissez ensuite une fonction **triangle(taille, couleur, angle)** capable de dessiner un triangle équilatéral d'une taille, d'une couleur et d'une orientation bien déterminées.
Testez votre module à l'aide d'un programme qui fera appel à ces fonctions à plusieurs reprises, avec des arguments variés pour dessiner une série de carrés et de triangles :

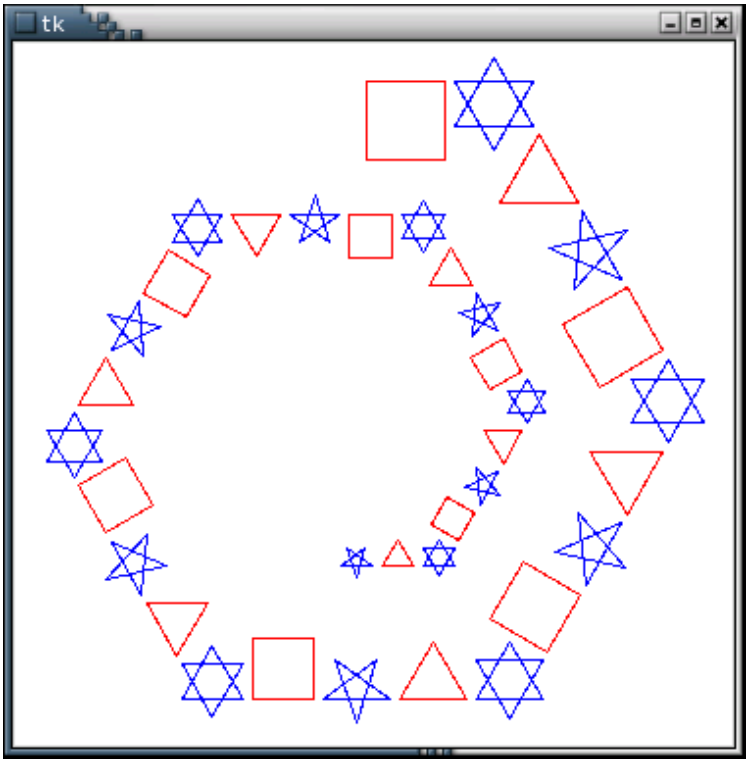


7.7 Ajoutez au module de l'exercice précédent une fonction **etoile5()** spécialisée dans le dessin d'étoiles à 5 branches. Dans votre programme principal, insérez une boucle qui dessine une rangée horizontale de de 9 petites étoiles de tailles variées :



7.8 Ajoutez au module de l'exercice précédent une fonction **etoile6()** capable de dessiner une étoile à 6 branches, elle-même constituée de deux triangles équilatéraux imbriqués. Cette nouvelle fonction devra faire appel à la fonction **triangle()** définie précédemment.
Votre programme principal dessinera également une série de ces étoiles :





7.9 Définissez une fonction **compteCar(ca,ch)** qui renvoie le nombre de fois que l'on rencontre le caractère **ca** dans la chaîne de caractères **ch**. Par exemple, l'exécution de l'instruction :
`print compteCar('e','Cette phrase est un exemple')` doit donner le résultat : `7`

7.10 Définissez une fonction **indexMax(liste)** qui renvoie l'index de l'élément ayant la valeur la plus élevée dans la liste transmise en argument. Exemple d'utilisation :

```
serie = [5, 8, 2, 1, 9, 3, 6, 7]
print indexMax(serie)
4
```

7.11 Définissez une fonction **nomMois(n)** qui renvoie le nom du n-ième mois de l'année.
Par exemple, l'exécution de l'instruction :
`print nomMois(4)` doit donner le résultat : `Avril`

7.12 Définissez une fonction **inverse(ch)** qui permette d'inverser les l'ordre des caractères d'une chaîne quelconque. (La chaîne inversée sera renvoyée au programme appelant).

7.13 Définissez une fonction **compteMots(ph)** qui renvoie le nombre de mots contenus dans la phrase **ph** (On considère comme mots les ensembles de caractères inclus entre des espaces).

Typage des paramètres

Vous avez appris que le *typage* des variables sous Python est un *typage dynamique*, ce qui signifie que le type d'une variable est défini au moment où on lui affecte une valeur. Ce mécanisme fonctionne aussi pour les paramètres d'une fonction. Le type d'un paramètre devient automatiquement le même que celui de l'argument qui a été transmis à la fonction. Exemple :

```
>>> def afficher3fois(arg):
...     print arg, arg, arg
...

>>> afficher3fois(5)
5 5 5

>>> afficher3fois('zut')
zut zut zut

>>> afficher3fois([5, 7])
[5, 7] [5, 7] [5, 7]

>>> afficher3fois(6**2)
36 36 36
```

Dans cet exemple, vous pouvez constater que la même fonction **afficher3fois()** accepte dans tous les cas l'argument qu'on lui transmet, que cet argument soit un nombre, une chaîne de caractères, une liste, *ou même une expression*. Dans ce dernier cas, Python commence par évaluer l'expression, et c'est le résultat de cette évaluation qui est transmis comme argument à la fonction.

Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction *qui peut être appelée avec une partie seulement des arguments attendus*. Exemples :

```
>>> def politesse(nom, vedette ='Monsieur'):
...     print "Veuillez agréer ,", vedette, nom, ", mes salutations
distinguées."
...

>>> politesse('Dupont')
Veuillez agréer , Monsieur Dupont , mes salutations distinguées.

>>> politesse('Durand', 'Mademoiselle')
Veuillez agréer , Mademoiselle Durand , mes salutations distinguées.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

Vous pouvez définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, cependant, *les paramètres sans valeur par défaut doivent précéder les autres* dans la liste. Par exemple, la définition ci-dessous est incorrecte :

```
>>> def politesse(vedette ='Monsieur', nom):
```

Autre exemple :

```
>>> def question(annonce, essais =4, please ='Oui ou non, s.v.p.!'):
...     while essais >0:
...         reponse = raw_input(annonce)
...         if reponse in ('o', 'oui','O','Oui','OUI'):
...             return 1
```

```
...         if reponse in ('n','non','N','Non','NON'):
...             return 0
...         print please
...         essais = essais-1
...
>>>
```

Cette fonction peut être appelée de différentes façons, telles par exemple :

```
rep = question('Voulez-vous vraiment terminer ? ')
```

ou bien :

```
rep = question('Faut-il effacer ce fichier ? ', 3)
```

ou même encore :

```
rep = question('Avez-vous compris ? ', 2, 'Répondez par oui ou par non !')
```

(Prenez la peine d'essayer et de décortiquer cet exemple)

Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis *exactement dans le même ordre* que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, sous la forme déjà décrite ci-dessus, on peut faire appel à la fonction en fournissant les arguments correspondants *dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants*. Exemple :

```
>>> def oiseau(voltage=100, etat='allumé', action='danser la java'):
...     print 'Ce perroquet ne pourra pas', action
...     print 'si vous le branchez sur', voltage, 'volts !'
...     print "L'auteur de ceci est complètement", etat
...

>>> oiseau(etat='givré', voltage=250, action='vous approuver')
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
L'auteur de ceci est complètement givré

>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
L'auteur de ceci est complètement allumé
```

Exercices :

7.14 Modifiez la fonction **volBoite(x1,x2,x3)** que vous avez définie dans un exercice précédent, de manière à ce qu'elle puisse être appelée avec trois, deux, un seul, ou même aucun argument. Utilisez pour ceux ci des valeurs par défaut égales à) 10.
Par exemple :

`print volBoite()`	doit donner le résultat : **1000**
`print volBoite(5.2)`	doit donner le résultat : **520.0**
`print volBoite(5.2, 3)`	doit donner le résultat : **156.0**

7.15 Modifiez la fonction **volBoite(x1,x2,x3)** ci-dessus de manière à ce qu'elle puisse être appelée avec un, deux, ou trois arguments. Si un seul est utilisé, la boîte est considérée comme cubique

(l'argument étant l'arête de ce cube). Si deux sont utilisés, la boîte est considérée comme un prisme à base carrée. (Dans ce cas le premier argument est le côté du carré, et le second la hauteur du prisme). Si trois arguments sont utilisés, la boîte est considérée comme un parallélépipède. Par exemple :

```
print volBoite()          doit donner le résultat : -1 (indication d'une erreur).
print volBoite(5.2)       doit donner le résultat : 140.608
print volBoite(5.2, 3)          doit donner le résultat : 81.12
print volBoite(5.2, 3, 7.4)   doit donner le résultat : 115.44
```

7.16 Définissez une fonction **changeCar(ch,ca1,ca2,debut,fin)** qui remplace tous les caractères **ca1** par des caractères **ca2** dans la chaîne de caractères **ch**, à partir de l'indice **debut** et jusqu'à l'indice **fin**, ces deux derniers arguments pouvant être omis (et dans ce cas la chaîne est traitée d'une extrémité à l'autre). Exemples de la fonctionnalité attendue :

```
>>> phrase = 'Ceci est une toute petite phrase.'
>>> print changeCar(phrase, ' ', '*')
Ceci*est*une*toute*petite*phrase.
>>> print changeCar(phrase, ' ', '*', 8, 12)
Ceci est*une*toute petite phrase.
>>> print changeCar(phrase, ' ', '*', 12)
Ceci est une*toute*petite*phrase.
>>> print changeCar(phrase, ' ', '*', fin = 12)
Ceci*est*une*toute petite phrase.
```

7.17 Définissez une fonction **eleMax(liste,debut,fin)** qui renvoie l'élément ayant la plus grande valeur dans la liste transmise. Les deux arguments **debut** et **fin** indiqueront les indices entre lesquels doit s'exercer la recherche, et chacun d'eux pourra être omis (comme dans l'exercice précédent). Exemples de la fonctionnalité attendue :

```
>>> serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
>>> print eleMax(serie)
9
>>> print eleMax(serie, 2, 5)
7
>>> print eleMax(serie, 2)
8
>>> print eleMax(serie, fin =3, debut =1)
6
```

8

Utilisation de fenêtres et de graphismes

Jusqu'à présent, nous avons utilisé Python exclusivement « en mode texte ». Nous avons procédé ainsi parce qu'il nous fallait absolument d'abord dégager un certain nombre de concepts élémentaires ainsi que la structure de base du langage, avant d'envisager des expériences impliquant des objets informatiques plus élaborés (fenêtres, images, sons, etc.). Nous pouvons à présent nous permettre une petite incursion dans le vaste domaine des interfaces graphiques, mais ce ne sera qu'un premier amuse-gueule : il nous reste en effet encore bien des choses fondamentales à apprendre, et pour nombre d'entre elles l'approche textuelle reste la plus abordable.

Interfaces graphiques (GUI)

Si vous ne le saviez pas encore, apprenez dès à présent que le domaine des interfaces graphiques (ou *GUI : Graphical User Interfaces*) est extrêmement complexe. Chaque système d'exploitation peut en effet proposer plusieurs « bibliothèques » de fonctions graphiques de base, auxquelles viennent fréquemment s'ajouter de nombreux compléments, plus ou moins spécifiques de langages de programmation particuliers. Tous ces composants sont généralement présentés comme des *classes d'objets*, dont il vous faudra étudier les *attributs* et les *méthodes*.

Avec Python, la bibliothèque graphique la plus utilisée jusqu'à présent est la bibliothèque *Tkinter*, qui est une adaptation de la bibliothèque *Tk* développée à l'origine pour le langage *Tcl*. Plusieurs autres bibliothèques graphiques fort intéressantes ont été proposées pour Python : *wxPython*, *pyQT*, *pyGTK*, etc. Il existe également des possibilités d'utiliser les bibliothèques de *widgets Java* et les *MFC* de *Windows*.
Dans le cadre de ces notes, nous nous limiterons cependant à *Tkinter*, dont il existe fort heureusement des versions similaires (et gratuites) pour les plates-formes *Linux*, *Windows* et *MacOS*.

Premiers pas avec Tkinter

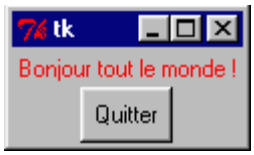
Pour la suite des explications, nous supposerons bien évidemment que le module *Tkinter* a déjà été installé sur votre système. Pour pouvoir en utiliser les fonctionnalités dans un script Python, il faut que l'une des premières lignes de ce script contienne l'instruction d'importation :

```
from Tkinter import *
```

Comme toujours sous Python, il n'est même pas nécessaire d'écrire un script. Vous pouvez faire un grand nombre d'expériences directement à la ligne de commande, en ayant simplement lancé

Python en mode interactif. Dans l'exemple qui suit, nous allons créer une fenêtre très simple, et y ajouter deux *widgets*[25] typiques : un bout de texte (ou *label*) et un bouton (ou *button*).

```
>>> from Tkinter import *
>>> fen1 = Tk()
>>> tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')
>>> tex1.pack()
>>> bou1 = Button(fen1, text='Quitter', command = fen1.destroy)
>>> bou1.pack()
>>> fen1.mainloop()
```



> ***Note :*** *Suivant la version de Python utilisée, vous verrez déjà apparaître la fenêtre d'application immédiatement après avoir entré la deuxième commande de cet exemple, ou bien seulement après la septième*[26]*.*

Examinons à présent plus en détail chacune des lignes de commandes exécutées :

1. Comme cela a déjà été expliqué précédemment, il est aisé de construire différents modules Python, qui contiendront des scripts, des définitions de fonctions, des classes d'objets, etc. On peut alors importer tout ou partie de ces modules dans n'importe quel programme, ou même dans l'interpréteur fonctionnant en mode interactif (c'est-à-dire directement à la ligne de commande). C'est ce que nous faisons à la première ligne de notre exemple : `from Tkinter import *` consiste à importer toutes les classes contenues dans le module *Tkinter*.

   Nous devrons de plus en plus souvent parler de ces *classes*. En programmation, on appelle ainsi des *générateurs d'objets*, lesquels sont eux-mêmes des morceaux de programmes réutilisables. Nous n'allons pas essayer de vous fournir dès à présent une définition définitive et précise de ce que sont les objets et les classes, mais plutôt vous proposer d'en utiliser directement quelques-un(e)s. Nous affinerons notre compréhension petit à petit par la suite.

2. A la deuxième ligne de notre exemple : `fen1 = Tk()`, nous utilisons l'une des classes du module *Tkinter*, la classe **Tk()**, et nous en créons une *instance* (autre terme désignant un objet spécifique), à savoir la fenêtre **fen1**.

   Ce processus d'*instanciation d'un objet à partir d'une classe* est une opération fondamentale dans les techniques actuelles de programmation. Celles-ci font en effet de plus en plus souvent appel à une méthodologie que l'on appelle *programmation orientée objet* (ou OOP : *Object Oriented Programming*).

   La **classe** est en quelque sorte un modèle général (ou un moule) à partir duquel on demande à la machine de construire un objet informatique particulier. La classe contient toute une série de définitions et d'options diverses, dont nous n'utilisons qu'une partie dans l'objet que nous créons à partir d'elle. Ainsi la classe **Tk()** , qui est l'une des classes les plus fondamentales de la bibliothèque *Tkinter*, contient tout ce qu'il faut pour engendrer différents types de fenêtres d'application, de tailles ou de couleurs diverses, avec ou sans barre de menus, etc.

---

25 "*widget*" est le résultat de la contraction de l'expression "*window gadget*". Dans certains environnements de programmation, on appellera cela plutôt un "contrôle" ou un "composant graphique". Ce terme désigne en fait toute entité susceptible d'être placée dans une fenêtre d'application, comme par exemple un bouton, une case à cocher, une image, etc., et parfois aussi la fenêtre elle-même.

26 Si vous effectuez cet exercice sous *Windows*, nous vous conseillons d'utiliser de préférence une version standard de Python dans une fenêtre DOS ou dans IDLE plutôt que *PythonWin*. Vous pourrez mieux observer ce qui se passe après l'entrée de chaque commande.

Nous nous en servons ici pour créer notre objet graphique de base, à savoir la fenêtre qui contiendra tout le reste. Dans les parenthèses de **Tk()**, nous pourrions préciser différentes options, mais nous laisserons cela pour un peu plus tard.

L'instruction d'*instanciation* ressemble à une simple affectation de variable. Comprenons bien cependant qu'il se passe ici deux choses à la fois :

- *la création d'un nouvel objet*, (lequel peut être fort complexe dans certains cas, et par conséquent occuper un espace mémoire considérable) ;
- *l'affectation d'une variable,* qui va désormais servir de *référence* pour manipuler l'objet[27].

3. A la troisième ligne : `tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')`, nous créons un autre objet (un *widget*), cette fois à partir de la classe **Label()**.
Comme son nom l'indique, cette classe définit toutes sortes d'*étiquettes* (ou de *libellés*). En fait, il s'agit tout simplement de fragments de texte quelconques, utilisables pour afficher des informations et des messages divers à l'intérieur d'une fenêtre.

   Nous efforçant d'apprendre au passage la manière correcte d'exprimer les choses, nous dirons que nous créons ici l'objet **tex1** par *instanciation* de la classe **Label()**.

   Remarquons que nous faisons appel à une classe, de la même manière que nous faisons appel à une fonction : c'est-à-dire en fournissant un certain nombre d'arguments dans des parenthèses. Nous verrons plus loin qu'une classe est en fait une sorte de « conteneur » dans lequel sont regroupées des fonctions et des données.

   Quels arguments avons-nous donc fournis pour cette instanciation ?

   - Le premier argument transmis (**fen1**), indique que le nouveau *widget* que nous sommes en train de créer sera contenu *dans un autre widget préexistant*, que nous désignons donc ici comme son « maître » : l'objet **fen1** est le *widget maître* de l'objet **tex1**. (On pourra dire aussi que l'objet **tex1** est un *widget esclave* de l'objet **fen1**).
   - Les deux arguments suivants servent à préciser la forme exacte que doit prendre notre *widget*. Ce sont en effet deux options de création, chacune fournie sous la forme d'une chaîne de caractères : d'abord le texte de l'étiquette, ensuite sa couleur d'avant-plan (ou *foreground*, en abrégé **fg**). Ainsi le texte que nous voulons afficher est bien défini, et il doit apparaître coloré en rouge.
     Nous pourrions encore préciser bien d'autres caractéristiques : la police à utiliser, ou la couleur d'arrière-plan, par exemple. Toutes ces caractéristiques ont cependant une valeur par défaut dans les définitions internes de la classe **Label()**. Nous ne devons indiquer des options que pour les caractéristiques que nous souhaitons différentes du modèle standard.

4. A la quatrième ligne de notre exemple : `tex1.pack()` , nous activons une *méthode* associée à l'objet **tex1** : la méthode **pack()**. Nous avons déjà rencontré ce terme de méthode (à propos des listes, notamment). Une méthode est une fonction intégrée à un objet (on dira aussi qu'elle est *encapsulée* dans l'objet). Nous apprendrons bientôt qu'un ***objet*** informatique est en fait un élément de programme contenant toujours :

   - un certain nombre de *données* (numériques ou autres), contenues dans des variables de types divers : on les appelle les *attributs* (ou les *propriétés*) de l'objet.
   - un certain nombre de *procédures* ou de *fonctions* (qui sont donc des algorithmes) : on les appelle les *méthodes* de l'objet.

---

[27] Cette concision du langage est une conséquence du typage dynamique des variables en vigueur sous Python. D'autres langages utilisent une instruction particulière (telle que **new**) pour instancier un nouvel objet. Exemple :
maVoiture = new Cadillac (instanciation d'un objet de classe **Cadillac**, référencé dans la variable **maVoiture**)

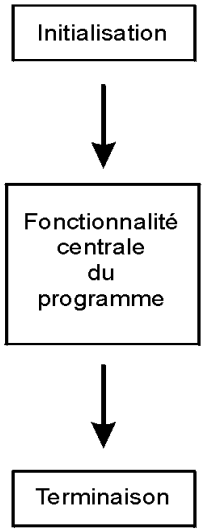
La méthode **pack()** fait partie d'un ensemble de méthodes qui sont applicables non seulement aux widgets de la classe **Label()**, mais aussi à la plupart des autres widgets *Tkinter*, et qui agissent sur leur disposition géométrique dans la fenêtre. Comme vous pouvez le constater par vous-même si vous entrez les commandes de notre exemple une par une, la méthode **pack()** réduit automatiquement la taille de la fenêtre « maître » afin qu'elle soit juste assez grande pour contenir les widgets « esclaves » définis au préalable.

5. A la cinquième ligne : `bou1 = Button(fen1, text='Quitter', command = fen1.destroy)`, nous créons notre second widget « esclave » : un bouton.
   Comme nous l'avons fait pour le widget précédent, nous appelons la classe **Button()** en fournissant entre parenthèses un certain nombre d'arguments. Étant donné qu'il s'agit cette fois d'un objet interactif, nous devons préciser avec l'option **command** ce qui devra se passer lorsque l'utilisateur effectuera un clic sur le bouton. Dans ce cas précis, nous actionnerons la méthode **destroy** associée à l'objet **fen1**, ce qui devrait provoquer l'effacement de la fenêtre.
6. La sixième ligne utilise la méthode **pack()** pour adapter la géométrie de la fenêtre au nouvel objet que nous venons d'y intégrer.
7. La septième ligne : `fen1.mainloop()` est très importante, parce que c'est elle qui provoque le démarrage du *réceptionnaire d'événements* associé à la fenêtre. Cette instruction est nécessaire pour que votre application soit « à l'affût » des clics de souris, des pressions exercées sur les touches du clavier, etc. C'est donc cette instruction qui « la met en marche », en quelque sorte.

   Comme son nom l'indique (*mainloop*), il s'agit d'une méthode de l'objet **fen1**, qui active une *boucle* de programme, laquelle « tournera » en permanence en tâche de fond, dans l'attente de messages émis par le système d'exploitation de l'ordinateur. Celui-ci interroge en effet sans cesse son environnement, notamment au niveau des périphériques d'entrée (souris, clavier, etc.). Lorsqu'un événement quelconque est détecté, divers *messages* décrivant cet événement sont expédiés aux programmes qui souhaitent en être avertis. Voyons cela un peu plus en détail.

Programmes pilotés par des événements

Vous venez d'expérimenter votre premier programme utilisant une interface graphique. Ce type de programme est structuré d'une manière différente des scripts « textuels » étudiés auparavant.

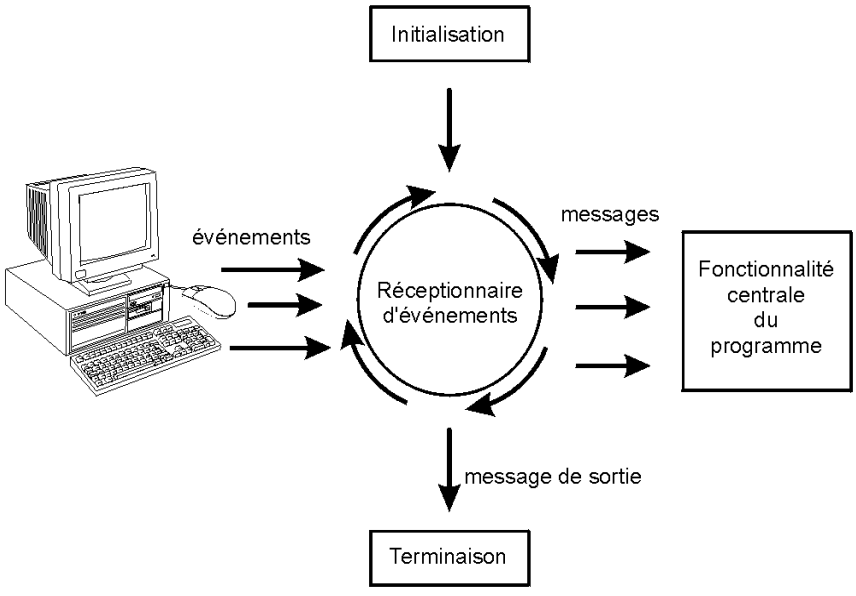


Tous les programmes d'ordinateur comportent *grosso-modo* trois phases principales : *une phase d'initialisation*, laquelle contient les instructions qui préparent le travail à effectuer (appel des modules externes nécessaires, ouverture de fichiers, connexion à un serveur de bases de données ou à l'internet, etc.), *une phase centrale* où l'on trouve la véritable fonctionnalité du programme (c'est-à-dire tout ce qu'il est censé faire : afficher des données à l'écran, effectuer des calculs, modifier le contenu d'un fichier, imprimer, etc.), et enfin *une phase de terminaison* qui sert à clôturer « proprement » les opérations (c'est-à-dire fermer les fichiers restés ouverts, couper les connexions externes, etc.)

Dans un programme « en mode texte », ces trois phases sont simplement organisées suivant un schéma *linéaire* comme dans l'illustration ci-contre. En conséquence, ces programmes se caractérisent par une *interactivité très limitée* avec l'utilisateur. Celui-ci ne dispose pratiquement d'aucune liberté : il lui est demandé de temps à autre d'entrer des données au clavier, mais toujours dans un ordre prédéterminé correspondant à la séquence d'instructions du programme.

Dans le cas d'un programme qui utilise une interface graphique, par contre, l'organisation interne est différente. On dit d'un tel programme qu'il est *piloté par les événements*. Après sa phase d'initialisation, un programme de ce type se met en quelque sorte « en attente », et passe la main à un autre logiciel, lequel est plus ou moins intimement intégré au système d'exploitation de l'ordinateur et « tourne » en permanence.

Ce *réceptionnaire d'événements* scrute sans cesse tous les périphériques (clavier, souris, horloge, modem, etc.) et réagit immédiatement lorsqu'un événement y est détecté.
Un tel événement peut être une action quelconque de l'utilisateur : déplacement de la souris, appui sur une touche, etc., mais aussi un événement externe ou un automatisme (top d'horloge, par ex.)



Lorsqu'il détecte un événement, le réceptionnaire envoie un message spécifique au programme[28], lequel doit être conçu pour réagir en conséquence.

La phase d'initialisation d'un programme utilisant une interface graphique comporte un ensemble d'instructions qui mettent en place les divers composants interactifs de cette interface (fenêtres, boutons, cases à cocher, etc.). D'autres instructions définissent les messages d'événements qui devront être pris en charge : on peut en effet décider que le programme ne réagira qu'à certains événements en ignorant tous les autres.

Alors que dans un programme « textuel », la phase centrale est constituée d'une suite d'instructions qui décrivent à l'avance l'ordre dans lequel la machine devra exécuter ses différentes tâches (même s'il est prévu des cheminements différents en réponse à certaines conditions rencontrées en cours de route), on ne trouve dans la phase centrale d'un programme avec interface graphique qu'un ensemble de fonctions indépendantes. Chacune de ces fonctions est appelée spécifiquement lorsqu'un événement particulier est détecté par le système d'exploitation : elle effectue alors le travail que l'on attend du programme en réponse à cet événement, et rien d'autre[29].

Il est important de bien comprendre ici que pendant tout ce temps, le réceptionnaire continue à « tourner » et à guetter l'apparition d'autres événements éventuels.

S'il arrive d'autres événements, il peut donc se faire qu'une deuxième fonction (ou une 3e, une 4e, ...) soit activée et commence à effectuer son travail « en parallèle » avec la première qui n'a pas encore terminé le sien[30]. Les systèmes d'exploitation et les langages modernes permettent en effet ce parallélisme que l'on appelle aussi *multitâche*.

Au chapitre précédent de ces notes, nous vous avons déjà fait remarquer que la structure du texte d'un programme n'indique pas directement l'ordre dans lequel les instructions seront finalement exécutées. Cette remarque s'applique encore bien davantage dans le cas d'un programme avec interface graphique, puisque l'ordre dans lequel les fonctions sont appelées n'est plus inscrit nulle part dans le programme. Ce sont les événements qui pilotent !

Tout ceci doit vous paraître un peu compliqué. Nous allons l'illustrer dans quelques exemples.

---

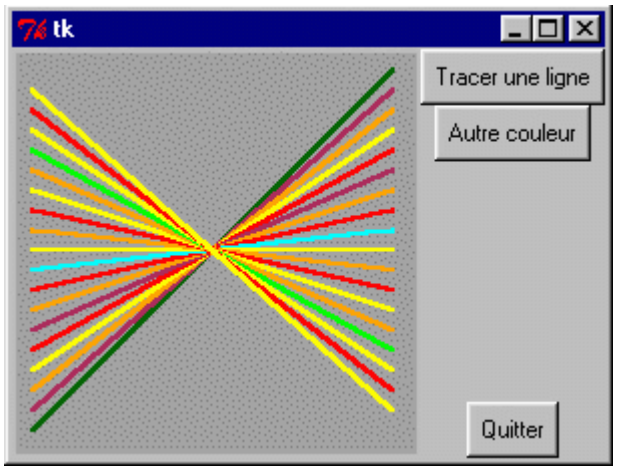
[28] Ces messages sont souvent notés WM (Window messages) dans un environnement graphique constitué de fenêtres (avec de nombreuses zones réactives : boutons, cases à cocher, menus déroulants, etc.). Dans la description des algorithmes, il arrive fréquemment aussi qu'on confonde ces messages avec les événements eux-mêmes.

[29] Au sens strict, une telle fonction qui ne devra renvoyer aucune valeur est donc plutôt une ***procédure*** (cfr. page 73).

[30] En particulier, la même fonction peut être appelée plusieurs fois en réponse à l'occurrence de quelques événements identiques, la même tâche étant alors effectuée en plusieurs exemplaires concurrents. Nous verrons plus loin qu'il peut en résulter des "effets de bords" gênants.

Exemple graphique : tracé de lignes dans un canevas

Le script décrit ci-dessous crée une fenêtre comportant trois boutons et un *canevas*. Suivant la terminologie de *Tkinter*, un *canevas* est une surface rectangulaire délimitée, dans laquelle on peut installer ensuite divers dessins et images à l'aide de méthodes spécifiques[31].



Lorsque l'on actionne le bouton « Tracer une ligne », une nouvelle ligne colorée apparaît sur le canevas, avec à chaque fois une inclinaison différente de la précédente.

Si l'on actionne le bouton « Autre couleur », une nouvelle couleur est tirée au hasard dans une série limitée. Cette couleur est celle qui s'appliquera aux tracés suivants.

Le bouton « Quitter » sert bien évidemment à terminer l'application en refermant la fenêtre.

```
# Petit exercice utilisant la bibliothèque graphique Tkinter

from Tkinter import *
from random import randrange


# --- définition des fonctions gestionnaires d'événements : ---
def drawline():
    "Tracé d'une ligne dans le canevas can1"
    global x1, y1, x2, y2, coul
    can1.create_line(x1,y1,x2,y2,width=2,fill=coul)

    # modification des coordonnées pour la ligne suivante :
    y2, y1 = y2+10, y1-10

def changecolor():
    "Changement aléatoire de la couleur du tracé"
    global coul
    pal=['purple','cyan','maroon','green','red','blue','orange','yellow']
    c = randrange(8)         # => génère un nombre aléatoire de 0 à 7
    coul = pal[c]

#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1, x2, y2 = 10, 190, 190, 10      # coordonnées de la ligne
coul = 'dark green'                    # couleur de la ligne

# Création du widget principal ("maître") :
fen1 = Tk()
# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=200,width=200)
can1.pack(side=LEFT)
bou1 = Button(fen1,text='Quitter',command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1,text='Tracer une ligne',command=drawline)
bou2.pack()
bou3 = Button(fen1,text='Autre couleur',command=changecolor)
bou3.pack()

fen1.mainloop()    # démarrage du réceptionnaire d'événements
```

[31] Ces dessins pourront éventuellement être animés dans une phase ultérieure (voir plus loin)

```
fen1.destroy()  # destruction (fermeture) de la fenêtre
```

Conformément à ce que nous avons expliqué dans le texte des pages précédentes, la fonctionnalité de ce programme est essentiellement assurée par les deux fonctions **drawline()** et **changecolor()**, qui seront activées par des événements, ceux-ci étant eux-mêmes définis dans la phase d'initialisation.

Dans cette phase d'initialisation, on commence par importer l'intégralité du module *Tkinter* ainsi qu'une fonction du module *random* qui permet de tirer des nombres au hasard. On crée ensuite les différents widgets par instanciation à partir des classes **Tk()**, **Canvas()** et **Button()**. Remarquons au passage que la même classe **Button()** sert à instancier plusieurs boutons, qui sont des objets similaires pour l'essentiel, mais néanmoins individualisés grâce aux options de création, et qui pourront fonctionner indépendamment l'un de l'autre.

L'initialisation se termine avec l'instruction **fen1.mainloop()** qui démarre le réceptionnaire d'événements. Les instructions qui suivent ne seront exécutées qu'à la sortie de cette boucle, sortie elle-même déclenchée par la méthode **fen1.quit()** (voir ci-après).

L'option **command** utilisée dans l'instruction d'instanciation des boutons permet de désigner la fonction qui devra être appelée lorsqu'un événement « *clic gauche de la souris sur le widget* » se produira. Il s'agit en fait d'un raccourci pour cet événement particulier, qui vous est proposé par *Tkinter* pour votre facilité parce que cet événement est celui que l'on associe naturellement à un widget de type bouton. Nous verrons plus loin qu'il existe d'autres techniques plus générales pour associer n'importe quel type d'événement à n'importe quel widget.

Les fonctions de ce script peuvent modifier les valeurs de certaines variables qui ont été définies au niveau principal du programme. Cela est rendu possible grâce à l'instruction **global** utilisée dans la définition de ces fonctions. Nous nous permettrons de procéder ainsi pendant quelque temps encore (ne serait-ce que pour vous habituer à distinguer les comportements des variables locales et globales), mais comme vous le comprendrez plus loin, *cette pratique n'est pas vraiment recommandable,* surtout lorsqu'il s'agit d'écrire de grands programmes. Nous apprendrons une meilleure technique lorsque nous aborderons l'étude des classes (à partir de la page 159).

Dans notre fonction **changecolor()**, une couleur est choisie au hasard dans une liste. Nous utilisons pour ce faire la fonction **randrange()** importée du module *random*. Appelée avec un argument **N**, cette fonction renvoie un nombre entier, tiré au hasard entre zéro et N-1.

La commande liée au bouton « Quitter » appelle la méthode **quit()** de la fenêtre **fen1**. Cette méthode sert à fermer (quitter) le réceptionnaire d'événements (**mainloop**) associé à cette fenêtre. Lorsque cette méthode est activée, l'exécution du programme se poursuit avec les instructions qui suivent l'appel de **mainloop**. Dans notre exemple, cela consiste donc à effacer (**destroy**) la fenêtre.

8 *Exercices*

8.1 Comment faut-il modifier le programme pour ne plus avoir que des lignes de couleur *cyan, maroon et green* ?

8.2 Comment modifier le programme pour que toutes les lignes tracées soient horizontales et parallèles ?

8.3 Agrandissez le canevas de manière à lui donner une largeur de 500 unités et une hauteur de 650. Modifiez également la taille des lignes, afin que leurs extrémités se confondent avec les bords du canevas.

8.4 Ajoutez une fonction « **drawline2** » qui tracera deux lignes rouges en croix au centre du canevas : l'une horizontale et l'autre verticale. Ajoutez également un bouton portant l'indication « viseur ». Un clic sur ce bouton devra provoquer l'affichage de la croix.

8.5 Reprenez le programme initial. Remplacez la méthode « **create_line** » par « **create_rectangle** ». Que se passe-t-il ?
De la même façon, essayez aussi « **create_arc** », « **create_oval** », et « **create_polygon** ».
Pour chacune de ces méthodes, notez ce qu'indiquent les coordonnées fournies en paramètres.
(Remarque : pour le polygone, il est nécessaire de modifier légèrement le programme !)

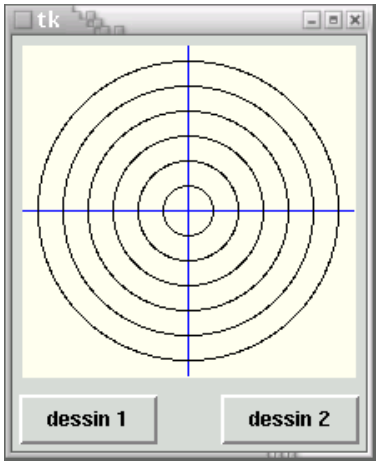
8.6 - Supprimez la ligne « global x1, y1, x2, y2 » dans la fonction « drawline » du programme original. Que se passe-t-il ? Pourquoi ?
- Si vous placez plutôt « x1, y1, x2, y2 » entre les parenthèses, dans la ligne de définition de la fonction « drawline », de manière à transmettre ces variables à la fonction en tant que paramètres, le programme fonctionne-t-il encore ? (N'oubliez pas de modifier aussi la ligne du programme qui fait appel à cette fonction !)
- Si vous définissez « x1, y1, x2, y2 = 10, 390, 390, 10 » à la place de « global x1, y1, ... », que se passe-t-il ? Pourquoi ? Quelle conclusion pouvez-vous tirer de tout cela ?

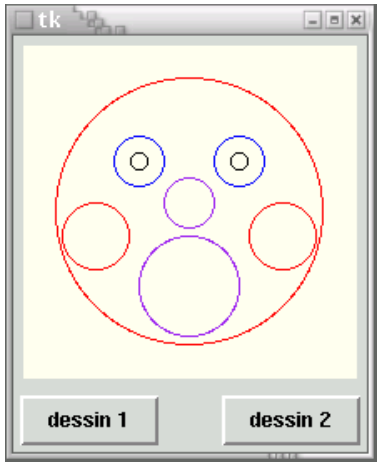
8.7 a) Créez un court programme qui dessinera les 5 anneaux olympiques dans un rectangle de fond blanc (white). Un boutton « Quitter » doit permettre de fermer la fenêtre.
b) Modifiez le programme ci-dessus en y ajoutant 5 boutons. Chacun de ces boutons provoquera le tracé de chacun des 5 anneaux

8.8 Dans votre cahier de notes, établissez un tableau à deux colonnes. Vous y noterez à gauche les définitions des classes d'objets déjà rencontrées (avec leur liste de paramètres), et à droite les méthodes associées à ces classes (également avec leurs paramètres). Laissez de la place pour compléter ultérieurement.

Exemple graphique : deux dessins alternés

Cet autre exemple vous montrera comment vous pouvez exploiter les connaissances que vous avez acquises précédemment concernant les boucles, les listes et les fonctions, afin de réaliser de nombreux dessins avec seulement quelques lignes de code. Il s'agit d'une petite application qui





affiche l'un ou l'autre des deux dessins reproduits ci-contre, en fonction du bouton choisi.

```
from Tkinter import *

def cercle(x, y, r, coul ='black'):
    "tracé d'un cercle de centre (x,y) et de rayon r"
    can.create_oval(x-r, y-r, x+r, y+r, outline=coul)

def figure_1():
    "dessiner une cible"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)
    # tracer les deux lignes (vert. et horiz.) :
    can.create_line(100, 0, 100, 200, fill ='blue')
    can.create_line(0, 100, 200, 100, fill ='blue')
    # tracer plusieurs cercles concentriques :
    rayon = 15
    while rayon < 100:
        cercle(100, 100, rayon)
        rayon += 15

def figure_2():
    "dessiner un visage simplifié"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)
    # Les caractéristiques de chaque cercle sont
    # placées dans une liste de listes :
    cc =[[100, 100, 80, 'red'],     # visage
         [70, 70, 15, 'blue'],      # yeux
         [130, 70, 15, 'blue'],
         [70, 70, 5, 'black'],
         [130, 70, 5, 'black'],
         [44, 115, 20, 'red'],      # joues
         [156, 115, 20, 'red'],
         [100, 95, 15, 'purple'],   # nez
         [100, 145, 30, 'purple']]  # bouche
    # on trace tous les cercles à l'aide d'une boucle :
    i =0
    while i < len(cc):      # parcours de la liste
        el = cc[i]          # chaque élément est lui-même une liste
        cercle(el[0], el[1], el[2], el[3])
        i += 1

##### Programme principal : #############
```

```
fen = Tk()
can = Canvas(fen, width =200, height =200, bg ='ivory')
can.pack(side =TOP, padx =5, pady =5)
b1 = Button(fen, text ='dessin 1', command =figure_1)
b1.pack(side =LEFT, padx =3, pady =3)
b2 = Button(fen, text ='dessin 2', command =figure_2)
b2.pack(side =RIGHT, padx =3, pady =3)
fen.mainloop()
```

Commençons par analyser le programme principal, à la fin du script :

Nous y créons une fenêtre, par instanciation d'un objet de la classe **Tk()** dans la variable **fen**. Ensuite, nous installons 3 widgets dans cette fenêtre : un canevas et deux boutons. Le canevas est instancié dans la variable **can**, et les deux boutons dans les variables **b1** et **b2**. Comme dans le script précédent, les widgets sont mis en place dans la fenêtre à l'aide de leur méthode **pack()**, mais cette fois nous utilisons celle-ci avec des options :

- l'option **side** peut accepter les valeurs TOP, BOTTOM, LEFT ou RIGHT, pour « pousser » le widget du côté correspondant dans la fenêtre. Ces noms écrits en majuscules sont en fait ceux d'une série de variables importées avec le module Tkinter, et que vous pouvez considérer comme des « pseudo-constantes ».
- les options **padx** et **pady** permettent de réserver un petit espace autour du widget. Cet espace est exprimé en nombre de pixels : **padx** réserve un espace à gauche et à droite du widget, **pady** réserve un espace au-dessus et au-dessous du widget.

Les boutons commandent l'affichage des deux dessins, en invoquant les fonctions **figure_1()** et **figure_2()**. Considérant que nous aurions à tracer un certain nombre de cercles dans ces dessins, nous avons estimé qu'il serait bien utile de définir d'abord une fonction **cercle()** spécialisée. En effet : Vous savez probablement déjà que le canevas Tkinter est doté d'une méthode **create_oval()** qui permet de dessiner des ellipses quelconques (et donc aussi des cercles), mais cette méthode doit être invoquée avec quatre arguments qui seront les coordonnées des coins supérieur gauche et inférieur droit d'un rectangle fictif, dans lequel l'ellipse viendra alors s'inscrire. Cela n'est pas très pratique dans le cas particulier du cercle : il nous semblera plus naturel de commander ce tracé en fournissant les coordonnées de son centre ainsi que son rayon. C'est ce que nous obtiendrons avec notre fonction **cercle()**, laquelle invoque la méthode **create_oval()** en effectuant la conversion des coordonnées. Remarquez que cette fonction attend un argument facultatif en ce qui concerne la couleur du cercle à tracer (noir par défaut).

L'efficacité de cette approche apparaît clairement dans la fonction **figure_1()**, ou nous trouvons une simple boucle de répétition pour dessiner toute la série de cercles (de même centre et de rayon croissant). Notez au passage l'utilisation de l'opérateur `+=` qui permet d'incrémenter une variable (dans notre exemple, la variable **r** voit sa valeur augmenter de 15 unités à chaque itération).

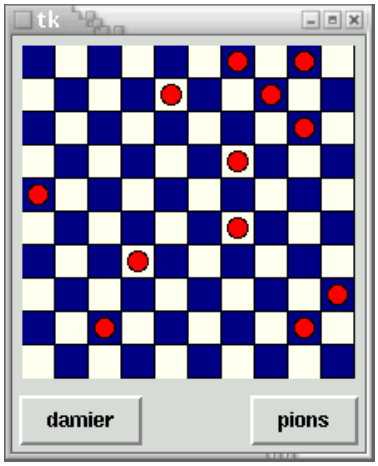
Le second dessin est un peu plus complexe, parce qu'il est composé de cercles de tailles variées centrés sur des points différents. Nous pouvons tout de même tracer tous ces cercles à l'aide d'une seule boucle de répétition, si nous mettons à profit nos connaissances concernant les listes.

En effet. Ce qui différencie les cercles que nous voulons tracer tient en quatre caractéristiques : coordonnées **x** et **y** du centre, rayon et couleur. Pour chaque cercle, nous pouvons placer ces quatre caractéristiques dans une petite liste, et rassembler toutes les petites listes ainsi obtenues dans une autre liste plus grande. Nous disposerons ainsi d'une liste de listes, qu'il suffira ensuite de parcourir à l'aide d'une boucle pour effectuer les tracés correspondants.

Exercices :

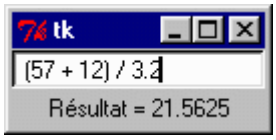
8.9 Inspirez-vous du script précédent pour écrire une petite application qui fait apparaître un damier (dessin de cases noires sur fond blanc) lorsque l'on clique sur un bouton :

8.10 À l'application de l'exercice précédent, ajoutez un bouton qui fera apparaître des pions au hasard sur le damier (chaque pression sur le bouton fera apparaître un nouveau pion).



Exemple graphique : calculatrice minimaliste

Bien que très court, le petit script ci-dessous implémente une calculatrice complète, avec laquelle vous pourrez même effectuer des calculs comportant des parenthèses et des fonctions scientifiques. N'y voyez rien d'extraordinaire. Toute cette fonctionnalité n'est qu'une conséquence du fait que vous utilisez un interpréteur plutôt qu'un compilateur pour exécuter vos programmes.



Comme vous le savez, le compilateur n'intervient qu'une seule fois, pour traduire l'ensemble de votre code source en un programme exécutable. Son rôle est donc terminé *avant même* l'exécution du programme. L'interpréteur, quant à lui, est toujours actif *pendant* l'exécution du programme, et donc tout à fait disponible pour traduire un nouveau code source quelconque, comme par exemple une expression mathématique entrée au clavier par l'utilisateur.

Les langages interprétés disposent donc toujours de fonctions permettant d'évaluer une chaîne de caractères comme une suite d'instructions du langage lui-même. Il devient alors possible de construire en peu de lignes des structures de programmes très dynamiques. Dans l'exemple ci-dessous, nous utilisons la fonction intégrée **eval()** pour analyser l'expression mathématique entrée par l'utilisateur dans le champ prévu à cet effet, et nous n'avons plus ensuite qu'à afficher le résultat.

```
# Exercice utilisant la bibliothèque graphique Tkinter et le module math

from Tkinter import *
from math import *

# définition de l'action à effectuer si l'utilisateur actionne
# la touche "enter" alors qu'il édite le champ d'entrée :

def evaluer(event):
    chaine.configure(text = "Résultat = " + str(eval(entree.get())))

# ----- Programme principal : -----

fenetre = Tk()
entree = Entry(fenetre)
entree.bind("<Return>", evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()

fenetre.mainloop()
```

Au début du script, nous commençons par importer les modules **Tkinter** et **math**, ce dernier étant nécessaire afin que la dite calculatrice puisse disposer de toutes les fonctions mathématiques et scientifiques usuelles : sinus, cosinus, racine carrée, etc.

Ensuite nous définissons une fonction **evaluer()**, qui sera en fait la commande exécutée par le programme lorsque l'utilisateur actionnera la touche *Return* (ou *Enter*) après avoir entré une expression mathématique quelconque dans le champ d'entrée décrit plus loin.

Cette fonction utilise la méthode **configure()** du widget **chaine**[32], pour modifier son attribut **text**. L'attribut en question reçoit donc ici une nouvelle valeur, déterminée par ce que nous avons écrit à la droite du signe égale : il s'agit en l'occurrence d'une chaîne de caractères construite dynamiquement, à l'aide de deux fonctions intégrées dans Python : **eval()** et **str()**, et d'une méthode associée à un widget Tkinter : la méthode **get()**.

eval() fait appel à l'interpréteur pour évaluer une expression Python qui lui est transmise dans une chaîne de caractères. Le résultat de l'évaluation est fourni en retour. Exemple :

```
chaine = "(25 + 8)/3"     # chaîne contenant une expression mathématique
res = eval(chaine)        # évaluation de l'expression contenue dans la chaîne
print res +5               # => le contenu de la variable res est numérique
```

str() transforme une expression numérique en chaîne de caractères. Nous devons faire appel à cette fonction parce que la précédente renvoie une valeur numérique, que nous convertissons à nouveau en chaîne de caractères pour pouvoir l'incorporer au message **`Résultat =`**.

get() est une méthode associée aux widgets de la classe **Entry**. Dans notre petit programme exemple, nous utilisons un widget de ce type pour permettre à l'utilisateur d'entrer une expression numérique quelconque à l'aide de son clavier. La méthode **get()** permet en quelque sorte « d'extraire » du widget **entree** la chaîne de caractères qui lui a été fournie par l'utilisateur.

Le corps du programme principal contient la phase d'initialisation, qui se termine par la mise en route de l'observateur d'événements (**mainloop**). On y trouve l'instanciation d'une fenêtre **Tk()**, contenant un widget **chaine** instancié à partir de la classe **Label()**, et un widget **entree** instancié à partir de la classe **Entry()**.

Attention, à présent : afin que ce dernier widget puisse vraiment faire son travail, c'est-à-dire transmettre au programme l'expression que l'utilisateur y aura encodée, nous lui associons un événement à l'aide de la méthode **bind()**[33] :

```
    entree.bind("<Return>",evaluer)
```

Cette instruction signifie : « *Lier l'événement <pression sur la touche Return> à l'objet <entree>, le gestionnaire de cet événement étant la fonction <evaluer>* ».

L'événement à prendre en charge est décrit dans une chaîne de caractères spécifique (dans notre exemple, il s'agit de la chaîne **`"<Return>"`**. Il existe un grand nombre de ces événements (mouvements et clics de la souris, enfoncement des touches du clavier, positionnement et redimensionnement des fenêtres, passage au premier plan, etc.). Vous trouverez la liste des chaînes spécifiques de tous ces événements dans les ouvrages de référence traitant de Tkinter.

Remarquez bien qu'il n'y a pas de parenthèses après le nom de la fonction **evaluer**. En effet : dans cette instruction, nous ne souhaitons pas déjà invoquer la fonction elle-même (ce serait prématuré) ; ce que nous voulons, c'est établir un lien entre un type d'événement particulier et cette fonction, de manière à ce qu'elle soit invoquée plus tard, chaque fois que l'événement se produira. Si

[32] La méthode **configure()** peut s'appliquer à n'importe quel widget préexistant, pour en modifier les propriétés.

[33] En anglais, le mot *bind* signifie "lier"

nous mettions des parenthèses, l'argument qui serait transmis à la méthode **bind()** serait la valeur de retour de cette fonction et non sa référence.

Profitons aussi de l'occasion pour observer encore une fois la syntaxe des instructions destinées à mettre en œuvre une méthode associée à un objet :

objet.méthode(arguments)

On écrit d'abord le nom de l'objet sur lequel on désire intervenir, puis le point (qui fait office d'opérateur), puis le nom de la méthode à mettre en œuvre ; entre les parenthèses associées à cette méthode, on indique enfin les arguments qu'on souhaite lui transmettre.

Exemple graphique : détection et positionnement d'un clic de souris

Dans la définition de la fonction « evaluer » de l'exemple précédent, vous aurez remarqué que nous avons fourni un argument **event** (entre les parenthèses).

Cet argument est obligatoire[34]. Lorsque vous définissez une fonction gestionnaire d'événement qui est associée à un widget quelconque à l'aide de sa méthode **bind()**, vous devez toujours l'utiliser comme premier argument. Cet argument désigne en effet un objet créé automatiquement par Tkinter, qui permet de transmettre au gestionnaire d'événement un certain nombre d'attributs de l'événement :

- le type d'événement : déplacement de la souris, enfoncement ou relâchement de l'un de ses boutons, appui sur une touche du clavier, entrée du curseur dans une zone prédéfinie, ouverture ou fermeture d'une fenêtre, etc.
- une série de propriétés de l'événement : l'instant où il s'est produit, ses coordonnées, les caractéristiques du ou des widget(s) concerné(s), etc.

Nous n'allons pas entrer dans trop de détails. Si vous voulez bien encoder et expérimenter le petit script ci-dessous, vous aurez vite compris le principe.

```
# Détection et positionnement d'un clic de souris dans une fenêtre :

from Tkinter import *

def pointeur(event):
    chaine.configure(text = "Clic détecté en X =" + str(event.x) +\
                            ", Y =" + str(event.y))

fen = Tk()
cadre = Frame(fen, width =200, height =150, bg="light yellow")
cadre.bind("<Button-1>", pointeur)
cadre.pack()
chaine = Label(fen)
chaine.pack()

fen.mainloop()
```

Le script fait apparaître une fenêtre contenant un *cadre* (**Frame**) rectangulaire de couleur jaune pâle, dans lequel l'utilisateur est invité à effectuer des clics de souris.

[34] La présence d'un argument est obligatoire, mais le nom *event* est une simple convention. Vous pourriez utiliser un autre nom quelconque à sa place, bien que cela ne soit pas recommandé.

La méthode **bind()** du widget *cadre* associe l'événement *<clic à l'aide du premier bouton de la souris>* au gestionnaire d'événement « pointeur ».

Ce gestionnaire d'événement peut utiliser les attributs **x** et **y** de l'objet **event** généré automatiquement par Tkinter, pour construire la chaîne de caractères qui affichera la position de la souris au moment du clic.

Exercice :

8.11 Modifiez le script ci-dessus de manière à faire apparaître un petit cercle rouge à l'endroit où l'utilisateur a effectué son clic (vous devrez d'abord remplacer le widget **Frame** par un widget Canvas).

Les classes de widgets Tkinter

> *Note : Au long de ce cours, nous vous présenterons petit à petit le mode d'utilisation d'un certain nombre de widgets. Comprenez bien cependant qu'il n'entre pas dans nos intentions de fournir ici un manuel de référence complet sur Tkinter. Nous limiterons nos explications aux widgets qui nous semblent les plus intéressants d'un point de vue didactique, c'est-à-dire ceux qui pourront nous aider à mettre en évidence des concepts importants, tel le concept de classe. Veuillez donc consulter la littérature (voir page 10) si vous souhaitez davantage de précisions.*

Il existe 15 classes de base pour les widgets Tkinter :

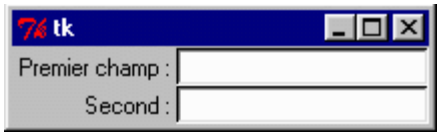
Widget	**Description**
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
Checkbutton	Une « case à cocher » qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.
Label	Un texte (ou libellé) quelconque (éventuellement une image).
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de « boutons radio » ou de cases à cocher.
Menu	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « *pop up* » apparaissant n'importe où à la suite d'un clic.
Menubutton	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
Message	Permet d'afficher un texte. Ce widget est une variante du widget Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un « bouton radio » donne la valeur correspondante à la variable, et "vide" tous les autres boutons radio associés à la même variable.
Scale	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
Scrollbar	« ascenseur » ou « barre de défilement » que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
Text	Affichage de texte formatté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées.
Toplevel	Une fenêtre affichée séparément, « par-dessus ».

Ces classes de widgets intègrent chacune un grand nombre de méthodes. On peut aussi leur associer (lier) des événements, comme nous venons de le voir dans les pages précédentes. Vous allez apprendre en outre que tous ces widgets peuvent être positionnés dans les fenêtres à l'aide de trois méthodes différentes : la méthode **grid()**, la méthode **pack()** et la méthode **place()**.

L'utilité de ces méthodes apparaît clairement lorsque l'on s'efforce de réaliser des programmes portables (c'est-à-dire susceptibles de fonctionner indifféremment sur des systèmes d'exploitation aussi différents que *Unix*, *MacOS* ou *Windows*), et dont les fenêtres soient *redimensionnables.*

Utilisation de la méthode grid() pour contrôler la disposition des widgets

Jusqu'à présent, nous avons toujours disposé les widgets dans leur fenêtre, à l'aide de la méthode **pack()**. Cette méthode présentait l'avantage d'être extraordinairement simple, mais elle ne nous donnait pas beaucoup de liberté pour disposer les widgets à notre guise. Comment faire, par exemple, pour obtenir la fenêtre ci-contre ?



Nous pourrions effectuer un certain nombre de tentatives en fournissant à la méthode **pack()** des arguments de type « side = », comme nous l'avons déjà fait précédemment, mais cela ne nous mène pas très loin. Essayons par exemple :

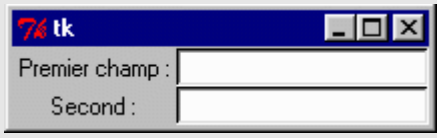
```
from Tkinter import *

fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.pack(side =LEFT)
txt2.pack(side =LEFT)
entr1.pack(side =RIGHT)
entr2.pack(side =RIGHT)

fen1.mainloop()
```

... mais le résultat n'est pas vraiment celui que nous recherchions !!! :

Pour mieux comprendre comment fonctionne la méthode **pack()**, vous pouvez encore essayer différentes combinaisons d'options, telles que `side =TOP`, `side =BOTTOM`, pour chacun de ces quatre widgets. Mais vous n'arriverez certainement pas à obtenir ce qui vous a été demandé. Vous pourriez peut-être y parvenir en définissant deux widgets **Frame()** supplémentaires, et en y incorporant ensuite séparément les widgets **Label()** et **Entry()**. Cela devient fort compliqué.

Il est temps que nous apprenions à utiliser une autre approche du problème. Veuillez donc analyser le script ci-dessous : il contient en effet (presque) la solution :

```
from Tkinter import *

fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.grid(row =0)
txt2.grid(row =1)
entr1.grid(row =0, column =1)
entr2.grid(row =1, column =1)
fen1.mainloop()
```



Dans ce script, nous avons donc remplacé la méthode **pack()** par la méthode **grid()**. Comme vous pouvez le constater, l'utilisation de la méthode **grid()** est très simple. Cette méthode considère la fenêtre comme un tableau (ou une grille). Il suffit alors de lui indiquer dans quelle ligne (**row**) et dans quelle colonne (**column**) de ce tableau on souhaite placer les widgets. On peut numéroter les lignes et les colonnes comme on veut, en partant de zéro, ou de un, ou encore d'un nombre quel-

conque : Tkinter ignorera les lignes et colonnes vides. Notez cependant que si vous ne fournissez aucun numéro pour une ligne ou une colonne, la valeur par défaut sera zéro.

Tkinter détermine automatiquement le nombre de lignes et de colonnes nécessaire. Mais ce n'est pas tout : si vous examinez en détail la petite fenêtre produite par le script ci-dessus, vous constaterez que nous n'avons pas encore tout à fait atteint le but poursuivi. Les deux chaînes apparaissant dans la partie gauche de la fenêtre sont *centrées*, alors que nous souhaitions les *aligner* l'une et l'autre par la droite. Pour obtenir ce résultat, il nous suffit d'ajouter un argument dans l'appel de la méthode **grid()** utilisée pour ces widgets. L'option **sticky** peut prendre l'une des quatre valeurs **N, S, W, E** (les quatre points cardinaux en anglais). En fonction de cette valeur, on obtiendra un alignement des widgets par le haut, par le bas, par la gauche ou par la droite. Remplacez donc les deux premières instructions **grid()** du script par :

```
txt1.grid(row =0, sticky =E)
txt2.grid(row =1, sticky =E)
```

... et vous atteindrez enfin exactement le but recherché.

Analysons à présent la fenêtre suivante :

Cette fenêtre comporte 3 colonnes : une première avec les 3 chaînes de caractères, une seconde avec les 3 champs d'entrée, et une troisième avec l'image. Les deux premières colonnes comportent chacune 3 lignes, mais l'image située dans la dernière colonne *s'étale* en quelque sorte sur les trois.

Le code correspondant est le suivant :

```
from Tkinter import *

fen1 = Tk()

# création de widgets 'Label' et 'Entry' :
txt1 = Label(fen1, text ='Premier champ :')
txt2 = Label(fen1, text ='Second :')
txt3 = Label(fen1, text ='Troisième :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
entr3 = Entry(fen1)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_P.gif')
item = can1.create_image(80, 80, image =photo)

# Mise en page à l'aide de la méthode 'grid' :
txt1.grid(row =1, sticky =E)
txt2.grid(row =2, sticky =E)
txt3.grid(row =3, sticky =E)
entr1.grid(row =1, column =2)
```

```
entr2.grid(row =2, column =2)
entr3.grid(row =3, column =2)
can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)

# démarrage :
fen1.mainloop()
```

Pour pouvoir faire fonctionner ce script, il vous faudra probablement remplacer le nom du fichier image (*Martin_P.gif*) par le nom d'une image de votre choix. Attention : la bibliothèque Tkinter standard n'accepte qu'un petit nombre de formats pour cette image. Choisissez de préférence le format GIF[35].

Nous pouvons remarquer un certain nombre de choses dans ce script :

1. La technique utilisée pour incorporer une image :
   Tkinter ne permet pas d'insérer directement une image dans une fenêtre. Il faut d'abord installer un canevas, et ensuite positionner l'image dans celui-ci. Nous avons opté pour un canevas de couleur blanche, afin de pouvoir le distinguer de la fenêtre. Vous pouvez remplacer le paramètre `bg ='white'` par `bg ='gray'` si vous souhaitez que le canevas devienne invisible. Étant donné qu'il existe de nombreux types d'images, nous devons en outre déclarer l'objet image comme étant un bitmap GIF, à partir de la classe **PhotoImage()**.

2. La ligne où nous installons l'image dans le canevas est la ligne :
   `item = can1.create_image(80, 80, image =photo)`
   Pour employer un vocabulaire correct, nous dirons que nous utilisons ici la méthode **create_image()** associée à l'objet **can1** (lequel objet est lui-même une instance de la classe **Canvas**). Les deux premiers arguments transmis (**80**, **80**) indiquent les coordonnées **x** et **y** du canevas où il faut placer le centre de l'image. (Les dimensions du canevas étant de 160x160, notre choix aboutira donc à un centrage de l'image au milieu du canevas).

3. La numérotation des lignes et colonnes dans la méthode **grid()** :
   On peut constater que la numérotation des lignes et des colonnes dans la méthode **grid()** utilisée ici commence cette fois à partir de 1 (et non à partir de zéro comme dans le script précédent). Comme nous l'avons déjà signalé plus haut, ce choix de numérotation est tout à fait libre.
   On pourrait tout aussi bien numéroter : 5, 10, 15, 20... puisque Tkinter ignore les lignes et les colonnes vides. Numéroter à partir de l augmente probablement la lisibilité de notre code.

4. Les arguments utilisés avec **grid()** pour positionner le canevas :
   `can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)`
   Les deux premiers arguments indiquent que le canevas sera placé dans la première ligne de la troisième colonne. Le troisième (`rowspan =3`) indique qu'il pourra « s'étaler » sur trois lignes. Les deux derniers (`padx =10, pady =5`) indiquent la dimension de l'espace qu'il faut réserver autour de ce widget (en largeur et en hauteur).

5. Et tant que nous y sommes, profitons de cet exemple de script que nous avons déjà bien décortiqué, pour apprendre à simplifier quelque peu notre code ...

[35] D'autres formats d'image sont possibles, mais à la condition de les traiter à l'aide des modules graphiques de la bibliothèque PIL (*Python Imaging Library*), qui est une extension de Python disponible sur : *http://www.pythonware.com/products/pil/* . Cette bibliothèque permet en outre d'effectuer une multitude de traitements divers sur des images, mais l'étude de ces techniques dépasse le cadre que nous nous sommes fixés pour ce manuel.

Composition d'instructions pour écrire un code plus compact

Du fait que Python est un langage de programmation de haut niveau, il est souvent possible (et souhaitable) de retravailler un script afin de le rendre plus compact.

Vous pouvez par exemple assez fréquemment utiliser la composition d'instructions pour appliquer la méthode de mise en page des widgets (**grid()**, **pack()** ou **place()**) au moment même où vous créez ces widgets. Le code correspondant devient alors un peu plus simple, et parfois plus lisible. Vous pouvez par exemple remplacer les deux lignes :

```
txt1 = Label(fen1, text ='Premier champ :')
txt1.grid(row =1, sticky =E)
```

du script précédent par une seule, telle que :

```
Label(fen1, text ='Premier champ :').grid(row =1, sticky =E)
```

Dans cette nouvelle écriture, vous pouvez constater que nous faisons l'économie de la variable intermédiaire **txt1**. Nous avions utilisé cette variable pour bien dégager les étapes successives de notre démarche, mais elle n'est pas toujours indispensable. Le simple fait d'invoquer la classe **Label()** provoque en effet l'instanciation d'un objet de cette classe, même si l'on ne mémorise pas la référence de cet objet dans une variable (Tkinter la conserve de toute façon dans sa représentation interne de la fenêtre). Si l'on procède ainsi, la référence est perdue pour le restant du script, mais elle peut tout de même être transmise à une méthode de mise en page telle que **grid()** au moment même de l'instanciation, en une seule instruction composée. Voyons cela un peu plus en détail :

Jusqu'à présent, nous avons créé des objets divers (par instanciation à partir d'une classe quelconque), en les affectant à chaque fois à des variables. Par exemple, lorsque nous avons écrit :

```
txt1 = Label(fen1, text ='Premier champ :')
```

nous avons créé une instance de la classe **Label()**, que nous avons assignée à la variable **txt1.**

La variable **txt1** peut alors être utilisée pour faire référence à cette instance, partout ailleurs dans le script, mais dans les faits nous ne l'utilisons qu'une seule fois pour lui appliquer la méthode **grid()**, le widget dont il est question n'étant rien d'autre qu'une simple étiquette descriptive. Or, créer ainsi une nouvelle variable pour n'y faire référence ensuite qu'une seule fois (et directement après sa création) n'est pas une pratique très recommandable, puisqu'elle consiste à réserver inutilement un certain espace mémoire.

Lorsque ce genre de situation se présente, il est plus judicieux d'utiliser la composition d'instructions. Par exemple, on préférera le plus souvent remplacer les deux instructions :

```
somme = 45 + 72
print somme
```

par une seule instruction composée, telle que :

```
print 45 + 72
```

on fait ainsi l'économie d'une variable.

De la même manière, lorsque l'on met en place des widgets auxquels on ne souhaite plus revenir par après, comme c'est souvent le cas pour les widgets de la classe **Label()**, on peut en général ap-

pliquer la méthode de mise en page (**grid()** , **pack()** ou **place()**) directement au moment de la création du widget, en une seule instruction composée.

Cela s'applique seulement aux widgets qui ne sont plus référencés après qu'on les ait créés. *Tous les autres doivent impérativement être assignés à des variables, afin que l'on puisse encore interagir avec eux ailleurs dans le script.*

Et dans ce cas, il faut obligatoirement utiliser deux instructions distinctes, l'une pour instancier le widget, et l'autre pour lui appliquer ensuite la méthode de mise en page. Vous ne pouvez pas, par exemple, construire une instruction composée telle que :

```
entree = Entry(fen1).pack()          # faute de programmation !!!
```

En apparence, cette instruction devrait instancier un nouveau widget et l'assigner à la variable **entree**, la mise en page s'effectuant dans la même opération à l'aide de la méthode **pack()**.
Dans la réalité, cette instruction produit bel et bien un nouveau widget de la classe **Entry()**, et la méthode **pack()** effectue bel et bien sa mise en page dans la fenêtre, mais la valeur qui est mémorisée dans la variable **entree** n'est pas la référence du widget ! C'est *la valeur de retour de la méthode* **pack()** : vous devez vous rappeler en effet que les méthodes, comme les fonctions, renvoient toujours une valeur au programme qui les appelle. Et vous ne pouvez rien faire de cette valeur de retour : il s'agit en l'occurrence d'un objet vide (**None**).

Pour obtenir une vraie référence du widget, vous devez obligatoirement utiliser deux instructions :

```
entree = Entry(fen1)          # instanciation du widget
entree.pack()                 # application de la mise en page
```

Note : Lorsque vous utilisez la méthode **grid()**, vous pouvez simplifier encore un peu votre code, en omettant l'indication de nombreux numéros de lignes et de colonnes. A partir du moment où c'est la la méthode **grid()** qui est utilisée pour positionner les widgets, Tkinter considère en effet qu'il existe forcément des lignes et des colonnes[36]. Si un numéro de ligne ou de colonne n'est pas indiqué, le widget correspondant est placé dans la première case vide disponible.

Le script ci-dessous intègre les simplifications que nous venons d'expliquer :

```
from Tkinter import *
fen1 = Tk()

# création de widgets Label(), Entry(), et Checkbutton() :
Label(fen1, text = 'Premier champ :').grid(sticky =E)
Label(fen1, text = 'Deuxième :').grid(sticky =E)
Label(fen1, text = 'Troisième :').grid(sticky =E)
entr1 = Entry(fen1)
entr2 = Entry(fen1)                  # ces widgets devront certainement
entr3 = Entry(fen1)                  # être  référencés plus loin :
entr1.grid(row =0, column =1)        # il faut donc les assigner chacun
entr2.grid(row =1, column =1)        # à une variable distincte
entr3.grid(row =2, column =1)
chek1 = Checkbutton(fen1, text ='Case à cocher, pour voir')
chek1.grid(columnspan =2)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_P.gif')
can1.create_image(80,80, image =photo)
can1.grid(row =0, column =2, rowspan =4, padx =10, pady =5)
```

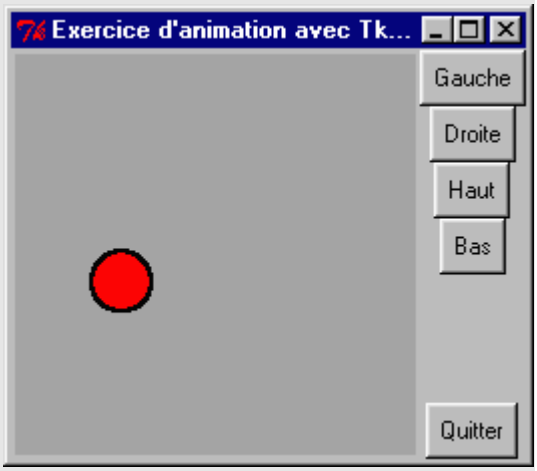
[36] Surtout, n'utilisez pas plusieurs méthodes de positionnement différentes dans la même fenêtre ! Les méthodes grid(), pack() et place() sont mutuellement exclusives.

```
# démarrage :
fen1.mainloop()
```

Modification des propriétés d'un objet - Animation

A ce stade de votre apprentissage, vous souhaitez probablement pouvoir faire apparaître un petit dessin quelconque dans un canevas, et puis le déplacer à volonté, par exemple à l'aide de boutons.

Veuillez donc écrire, tester, puis analyser le script ci-dessous :




```
from Tkinter import *

# procédure générale de déplacement :
def avance(gd, hb):
    global x1, y1
    x1, y1 = x1 +gd, y1 +hb
    can1.coords(oval1, x1, y1, x1+30, y1+30)

# gestionnaires d'événements :
def depl_gauche():
    avance(-10, 0)

def depl_droite():
    avance(10, 0)

def depl_haut():
    avance(0, -10)

def depl_bas():
    avance(0, 10)

#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10    # coordonnées initiales

# Création du widget principal ("maître") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")

# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=300,width=300)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
can1.pack(side=LEFT)
Button(fen1,text='Quitter',command=fen1.quit).pack(side=BOTTOM)
Button(fen1,text='Gauche',command=depl_gauche).pack()
Button(fen1,text='Droite',command=depl_droite).pack()
Button(fen1,text='Haut',command=depl_haut).pack()
Button(fen1,text='Bas',command=depl_bas).pack()

# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()
```

Le corps de ce programme reprend de nombreuses éléments connus : nous y créons une fenêtre **fen1**, dans laquelle nous installons un canevas contenant lui-même un cercle coloré, plus cinq boutons de contrôle. Veuillez remarquer au passage que nous n'instancions pas les widgets boutons dans des variables (c'est inutile, puisque nous n'y faisons plus référence par après) : nous devons donc appliquer la méthode **pack()** directement au moment de la création de ces objets.

La vraie nouveauté de ce programme réside dans la fonction **avance()** définie au début du script. Chaque fois qu'elle sera appelée, cette fonction redéfinira les coordonnées de l'objet « cercle coloré » que nous avons installé dans le canevas, ce qui provoquera l'animation de cet objet.

Cette manière de procéder est tout à fait caractéristique de la programmation « orientée objet » : On commence par créer des objets, et puis *on agit sur ces objets en modifiant leurs propriétés, par l'intermédiaire de méthodes.*

En programmation impérative « à l'ancienne » (c'est-à-dire sans utilisation d'objets), on anime des figures, en les effaçant à un endroit, pour les redessiner ensuite un petit peu plus loin. En programmation « orientée objet », par contre, ces tâches sont prises en charge automatiquement par les classes dont les objets dérivent, et il ne faut donc pas perdre son temps à les reprogrammer.

Exercices :

8.12 Écrivez un programme qui fait apparaître une fenêtre avec un canevas. Dans ce canevas on verra deux cercles (de tailles et de couleurs différentes), qui sont censés représenter deux astres. Des boutons doivent permettre de les déplacer à volonté tous les deux dans toutes les directions. Sous le canevas, le programme doit afficher en permanence : a) la distance séparant les deux astres; b) la force gravitationnelle qu'ils exercent l'un sur l'autre (Penser à afficher en haut de fenêtre les masses choisies pour chacun d'eux, ainsi que l'échelle des distances). Dans cet exercice, vous utiliserez évidemment la loi de la gravitation universelle de Newton (cf. exercice 6.16, page 66, et un manuel de Physique générale).

8.13 En vous inspirant du programme qui détecte les clics de souris dans un canevas, modifiez le programme ci-dessus pour y réduire le nombre de boutons : pour déplacer un astre, il suffira de le choisir avec un bouton, et ensuite de cliquer sur le canevas pour que cet astre se positionne à l'endroit où l'on a cliqué.

8.14 Extension du programme ci-dessus. Faire apparaître un troisième astre, et afficher en permanence la force résultante agissant sur chacun des trois (en effet : chacun subit en permanence l'attraction gravitationnelle exercée par les deux autres !).

8.15 Même exercice avec des charges électriques (loi de Coulomb). Donner cette fois une possibilité de choisir le signe des charges.

8.16 Écrivez un petit programme qui fait apparaître une fenêtre avec deux champs : l'un indique une température en degrés *Celsius*, et l'autre la même température exprimée en degrés *Fahrenheit*. Chaque fois que l'on change une quelconque des deux températures, l'autre est corrigée en conséquence. Pour convertir les degrés *Fahrenheit* en *Celsius* et vice-versa, on utilise la formule $T_F = T_C \times 1{,}80 + 32$ (cf. cours de Physique générale). Revoyez aussi le petit programme concernant la calculatrice simplifiée.

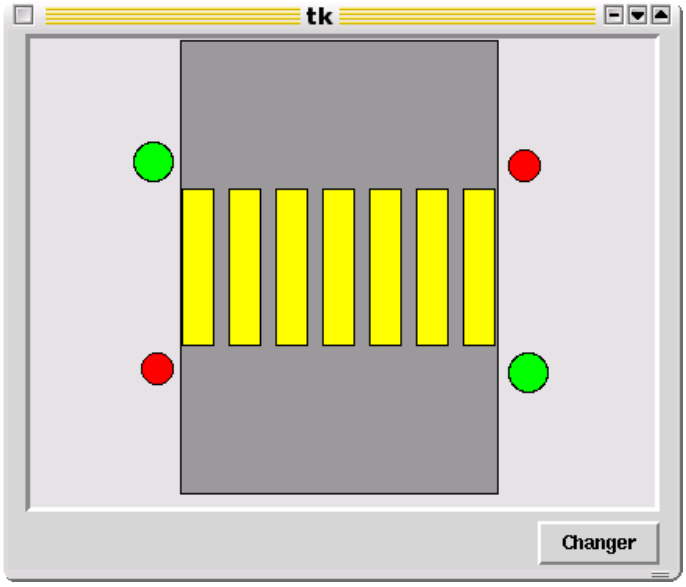
8.17 Écrivez un programme qui fasse apparaître une fenêtre avec un canevas. Dans ce canevas, placez un petit cercle censé représenter une balle. Sous le canevas, placez un bouton. Chaque fois que l'on clique sur le bouton, la balle doit avancer d'une petite distance vers la droite, jusqu'à ce qu'elle atteigne l'extrémité du canevas. Si l'on continue à cliquer, la balle doit alors revenir en arrière jusqu'à l'autre extrémité, et ainsi de suite.

8.18 Améliorez le programme ci-dessus pour que la balle décrive cette fois une trajectoire circulaire ou elliptique dans le canevas (lorsque l'on clique continuellement). Note : pour arriver au résultat escompté, vous devrez nécessairement définir une variable qui représentera l'angle décrit, et utiliser les fonctions *sinus* et *cosinus* pour positionner la balle en fonction de cet angle.

8.19 Modifiez le programme ci-dessus, de telle manière que la balle en se déplaçant laisse derrière elle une trace de la trajectoire décrite.

8.20 Modifiez le programme ci-dessus de manière à tracer d'autres figures. Consultez votre professeur pour des suggestions (courbes de *Lissajous*).

8.21 Écrivez un programme qui fasse apparaître une fenêtre avec un canevas et un bouton. Dans le canevas, tracez un rectangle gris foncé, lequel représentera une route, et par-dessus, une série de rectangles jaunes censés représenter un passage pour piétons. Ajoutez quatre cercles colorés pour figurer les feux de circulation concernant les piétons et les véhicules. Chaque utilisation du bouton devra provoquer le changement de couleur des feux :

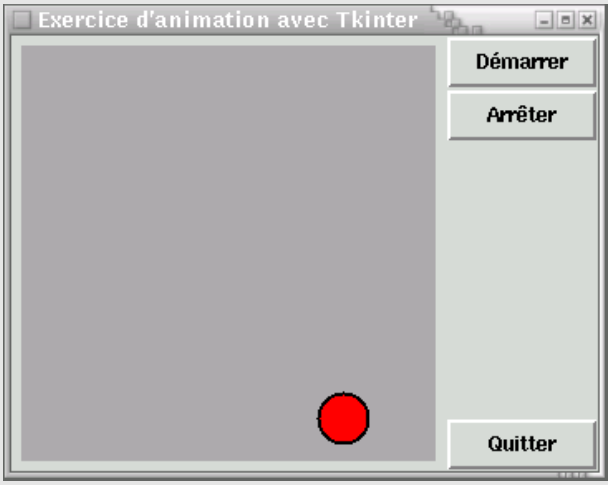


8.22 Écrivez un programme qui montre un canevas dans lequel est dessiné un circuit électrique simple (générateur + interrupteur + résistance). La fenêtre doit être pourvue de champs d'entrée qui permettront de paramétrer chaque élément (c'est-à-dire choisir les valeurs des résistances et tensions). L'interrupteur doit être fonctionnel (Prévoyez un bouton « Marche/arrêt » pour cela). Des « étiquettes » doivent afficher en permanence les tensions et intensités résultant des choix opérés par l'utilisateur.

Animation automatique - Récursivité

Pour conclure cette première prise de contact avec l'interface graphique Tkinter, voici un dernier exemple d'animation, qui fonctionne cette fois de manière autonome dès qu'on l'a mise en marche.

```
from Tkinter import *

# définition des gestionnaires
# d'événements :

def move():
    "déplacement de la balle"
    global x1, y1, dx, dy, flag
    x1, y1 = x1 +dx, y1 + dy
    if x1 >210:
        x1, dx, dy = 210, 0, 15
    if y1 >210:
        y1, dx, dy = 210, -15, 0
    if x1 <10:
        x1, dx, dy = 10, 0, -15
    if y1 <10:
        y1, dx, dy = 10, 15, 0
    can1.coords(oval1,x1,y1,x1+30,y1+30)
    if flag >0:
        fen1.after(50,move)       # =>
boucler après 50 millisecondes
```



```
def stop_it():
    "arret de l'animation"
    global flag
    flag =0

def start_it():
    "démarrage de l'animation"
    global flag
    if flag ==0:       # pour ne lancer qu'une seule boucle
       flag =1
       move()

#========== Programme principal =============

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10         # coordonnées initiales
dx, dy = 15, 0          # 'pas' du déplacement
flag =0                 # commutateur

# Création du widget principal ("parent") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")
# création des widgets "enfants" :
can1 = Canvas(fen1,bg='dark grey',height=250, width=250)
can1.pack(side=LEFT, padx =5, pady =5)
oval1 = can1.create_oval(x1, y1, x1+30, y1+30, width=2, fill='red')
bou1 = Button(fen1,text='Quitter', width =8, command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1, text='Démarrer', width =8, command=start_it)
bou2.pack()
bou3 = Button(fen1, text='Arrêter', width =8, command=stop_it)
bou3.pack()
# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()
```

La seule nouveauté mise en œuvre dans ce script se trouve tout à la fin de la définition de la fonction **move()** : vous y noterez l'utilisation de la méthode **after()**. Cette méthode peut s'appliquer à un widget quelconque. Elle déclenche l'appel d'une fonction *après qu'un certain laps de temps se soit écoulé*. Ainsi par exemple, `window.after(200,qqc)` déclenche pour le widget **window** un appel de la fonction **qqc()** après une pause de 200 millisecondes.

Dans notre script, la fonction qui est appelée par la méthode **after()** est la fonction **move()** elle-même. Nous utilisons donc ici pour la première fois une technique de programmation très puissante, que l'on appelle *récursivité*. Pour faire simple, nous dirons que la récursivité est ce qui se passe lorsqu'une fonction s'appelle elle-même. On obtient bien évidemment ainsi un bouclage, qui peut se perpétuer indéfiniment si l'on ne prévoit pas aussi un moyen pour l'interrompre.

Voyons comment cela fonctionne dans notre exemple :

La fonction **move()** est invoquée une première fois lorsque l'on clique sur le bouton « Démarrer ». Elle effectue son travail (c'est-à-dire positionner la balle). Ensuite, par l'intermédiaire de la méthode **after()**, elle s'invoque elle-même après une petite pause. Elle repart donc pour un second tour, puis s'invoque elle-même à nouveau, et ainsi de suite indéfiniment...

C'est du moins ce qui se passerait si nous n'avions pas pris la précaution de placer quelque part dans la boucle une instruction de sortie. En l'occurrence, il s'agit d'un simple test conditionnel : à chaque itération de la boucle, nous examinons le contenu de la variable **flag** à l'aide d'une instruction **if**. Si le contenu de la variable **flag** est zéro, alors le bouclage ne s'effectue plus et l'animation s'arrête. Remarquez que nous avons pris la précaution de définir **flag** comme une variable globale. Ainsi nous pouvons aisément changer sa valeur à l'aide d'autres fonctions, en l'occurrence celles que nous avons associées aux boutons « Démarrer » et « Arrêter ».

Nous obtenons ainsi un mécanisme simple pour lancer ou arrêter notre animation :

Un premier clic sur le bouton « Démarrer » assigne une valeur non-nulle à la variable **flag**, puis provoque immédiatement un premier appel de la fonction **move()**. Celle-ci s'exécute, puis continue ensuite à s'appeler elle-même toutes les 50 millisecondes, tant que **flag** ne revient pas à zéro. Si l'on continue à cliquer sur le bouton « Démarrer », la fonction **move()** ne peut plus être appelée, parce que la valeur de **flag** vaut désormais 1. On évite ainsi le démarrage de plusieurs boucles concurrentes.

Le bouton « Arrêter » remet **flag** à zéro, et la boucle s'interrompt.

Exercices :

8.23 Dans la fonction **start_it()**, supprimez l'instruction `if flag == 0:` (et l'indentation des deux lignes suivantes). Que se passe-t-il ? (Cliquez plusieurs fois sur le bouton « Démarrer »). Tâchez d'exprimer le plus clairement possible votre explication des faits observés.

8.24 Modifiez le programme de telle façon que la balle change de couleur à chaque « virage ».

8.25 Modifiez le programme de telle façon que la balle effectue des mouvements obliques comme une bille de billard qui rebondit sur les bandes (« en zig-zag »).

8.26 Modifiez le programme de manière à obtenir d'autres mouvements. Tâchez par exemple d'obtenir un mouvement circulaire. (Comme dans les exercices de la page 107).

8.27 Modifiez ce programme, ou bien écrivez-en un autre similaire, de manière à simuler le mouvement d'une balle qui tombe (sous l'effet de la pesanteur), et rebondit sur le sol. Attention : il s'agit cette fois de mouvements accélérés !

8.28 A partir des scripts précédents, vous pouvez à présent écrire un programme de jeu fonctionnant de la manière suivante :
Une balle se déplace au hasard sur un canevas, à vitesse faible. Le joueur doit essayer de cliquer sur cette balle à l'aide de la souris. S'il y arrive, il gagne un point, mais la balle se déplace désormais un peu plus vite, et ainsi de suite. Arrêter le jeu après un certain nombre de clics et afficher le score atteint.

8.29 Variante du jeu précédent : chaque fois que le joueur parvient à « l'attraper », la balle devient plus petite (elle peut également changer de couleur).

8.30 Écrivez un programme dans lequel évoluent plusieurs balles de couleurs différentes, qui rebondissent les unes sur les autres ainsi que sur les parois.

8.31 Perfectionnez le jeu des précédents exercices en y intégrant l'algorithme ci-dessus. Il s'agit à présent pour le joueur de cliquer seulement sur la balle rouge. Un clic erroné (sur une balle d'une autre couleur) lui fait perdre des points.

8.32 Écrivez un programme qui simule le mouvement de 2 planètes tournant autour du soleil sur des orbites circulaires différentes (ou deux électrons tournant autour d'un noyau d'atome...).

8.33 Écrivez un programme pour le jeu du serpent : un « serpent » (constitué en fait d'une courte ligne de carrés) se déplace sur le canevas dans l'une des 4 directions : droite, gauche, haut, bas. Le joueur peut à tout moment changer la direction suivie par le serpent à l'aide des touches fléchées du clavier. Sur le canevas se trouvent également des « proies » (des petits cercles fixes disposés au hasard). Il faut diriger le serpent de manière à ce qu'il « mange » les proies sans arriver en contact avec les bords du canevas. A chaque fois qu'une proie est mangée, le serpent s'allonge d'un carré, le joueur gagne un point, et une nouvelle proie apparaît ailleurs. La partie s'arrête lorsque le serpent touche l'une des parois, ou lorsqu'il a atteint une certaine taille.

8.34 Perfectionnement du jeu précédent : la partie s'arrête également si le serpent « se recoupe ».

9

Les fichiers

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'information plus importante.

Utilité des fichiers

Imaginons par exemple que nous voulons écrire un petit programme exerciseur qui fasse apparaître à l'écran des questions à choix multiple, avec traitement automatique des réponses de l'utilisateur. Comment allons-nous mémoriser le texte des questions elles-mêmes ?

L'idée la plus simple consiste à placer chacun de ces textes dans une variable, en début de programme, avec des instructions d'affectation du genre :

```
a = "Quelle est la capitale du Guatémala ?"
b = "Qui à succédé à Henri IV ?"
c = "Combien font 26 x 43 ?"
        ... etc.
```

Cette idée est malheureusement beaucoup trop simpliste. Tout va se compliquer en effet lorsque nous essayerons d'élaborer la suite du programme, c'est-à-dire les instructions qui devront servir à sélectionner au hasard l'une ou l'autre de ces questions pour les présenter à l'utilisateur. Employer par exemple une longue suite d'instructions **if** ... **elif** ... **elif** ... comme dans l'exemple ci-dessous n'est certainement pas la bonne solution (ce serait d'ailleurs bien pénible à écrire : n'oubliez pas que nous souhaitons traiter un grand nombre de questions !) :

```
if choix == 1:
    selection = a
elif choix == 2:
    selection = b
elif choix == 3:
    selection = c
        ... etc.
```

La situation se présente déjà beaucoup mieux si nous faisons appel à une liste :

```
liste = ["Qui a vaincu Napoléon à Waterloo ?",
         "Comment traduit-on 'informatique' en anglais ?",
         "Quelle est la formule chimique du méthane ?", ... etc ...]
```

On peut en effet extraire n'importe quel élément de cette liste à l'aide de son indice. Exemple :

```
print liste[2]        ===>  "Quelle est la formule chimique du méthane ?"
```

(rappel : l'indiçage commence à partir de zéro)

Même si cette façon de procéder est déjà nettement meilleure que la précédente, nous sommes toujours confrontés à plusieurs problèmes gênants :

- La lisibilité du programme va se détériorer très vite lorsque le nombre de questions deviendra important. En corollaire, nous accroîtrons la probabilité d'insérer l'une ou l'autre erreur de syntaxe dans la définition de cette longue liste. De telles erreurs seront bien difficiles à débusquer.
- L'ajout de nouvelles questions, ou la modification de certaines d'entre elles, imposeront à chaque fois de rouvrir le code source du programme. En corollaire, il deviendra malaisé de retravailler ce même code source, puisqu'il comportera de nombreuses lignes de données encombrantes.
- L'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.

Cette dernière remarque nous suggère la direction à prendre : il est temps que nous apprenions à *séparer les données, et les programmes qui les traitent, dans des fichiers différents.*

Pour que cela devienne possible, nous devrons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer par après.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches. Lorsque les quantités de données deviennent très importantes, il devient d'ailleurs rapidement nécessaire de structurer les relations entre ces données, et l'on doit alors élaborer des systèmes appelés *bases de données relationnelles*, dont la gestion peut s'avérer très complexe. Lorsque l'on est confronté à ce genre de problème, il est d'usage de déléguer une bonne part du travail à des logiciels très spécialisés tels que *Oracle*, *IBM DB2*, *Sybase*, *Adabas*, *PostgreSQL*, *MySQL*, etc. Python est parfaitement capable de dialoguer avec ces systèmes, mais nous laisserons cela pour un peu plus tard (voir : « Gestion d'une base de données », page 257).

Nos ambitions présentes sont plus modestes. Nos données ne se comptent pas encore par centaines de milliers, aussi nous pouvons nous contenter de mécanismes simples pour les enregistrer dans un fichier de taille moyenne, et les en extraire ensuite.

Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver (à l'aide de son titre), puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

Tout ce que nous venons de dire des livres s'applique également aux fichiers informatiques.
Un fichier se compose de données enregistrées sur votre disque dur, sur une disquette, une clef USB ou un CD-ROM. Vous y accédez grâce à son nom (lequel peut inclure aussi un nom de répertoire). Vous pouvez toujours considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères.

Noms de fichiers - Répertoire courant

Pour simplifier les explications qui vont suivre, nous indiquerons seulement des noms simples pour les fichiers que nous allons manipuler. Si vous procédez ainsi dans vos exercices, les fichiers

en question seront créés et/ou recherchés par Python *dans le répertoire courant*. Celui-ci est habituellement le répertoire où se trouve le script lui-même, sauf si vous lancez ce script depuis la fenêtre d'un shell *IDLE*, auquel cas le répertoire courant est défini au lancement de *IDLE* lui-même (Sous *Windows*, la définition de ce répertoire fait partie des propriétés de l'icône de lancement).

Si vous travaillez avec *IDLE*, vous souhaiterez donc certainement forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes. Pour ce faire, utilisez les commandes suivantes en début de session. (Nous supposons pour la démonstartion que le répertoire visé est le répertoire **`/home/jules/exercices`** . Même si vous travaillez sous *Windows* (ou ce n'est pas la règle), vous pouvez franchement utiliser cette syntaxe (c'est-à-dire des caractères / et non \ en guise de séparateurs : c'est la convention en vigueur dans le monde *Unix*). Python effectuera automatiquement les conversions nécessaires, suivant que vous travaillez sous *MacOS*, *Linux*, ou *Windows*.[37]

```
>>> from os import chdir
>>> chdir("/home/jules/exercices")
```

La première commande importe la fonction **chdir()** du module **os**. Le module **os** contient toute une série de fonctions permettant de dialoguer avec le système d'exploitation (**os** = *operating system*), quel que soit celui-ci.

La seconde commande provoque le changement de répertoire (**chdir** =*change directory*).

Notes :

- Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès complet dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.
- Choisissez de préférence des noms de fichiers courts. *Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux.*

Les deux formes d'importation

Les lignes d'instructions que nous venons d'utiliser sont l'occasion d'expliquer un mécanisme intéressant. Vous savez qu'en complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des *modules*. Ainsi vous connaissez déjà fort bien le module *math* et le module *Tkinter*.

Pour utiliser les fonctions d'un module, il suffit de les importer. Mais cela peut se faire de deux manières différentes, comme nous allons le voir ci-dessous. Chacune des deux méthodes présente des avantages et des inconvénients.

Voici un exemple de la première méthode :

```
>>>>>> import os
>>> rep_cour = os.getcwd()
>>> print rep_cour
C:\Python22\essais
```

La première ligne de cet exemple importe *l'intégralité* du module **os**, lequel contient de nombreuses fonctions intéressantes pour l'accès au système d'exploitation. La seconde ligne utilise la

[37] Dans le cas de *Windows*, vous pouvez également inclure dans ce chemin la lettre qui désigne le périphérique de stockage où se trouve le fichier. Par exemple : "D:/home/jules/exercices".

fonction **getcwd()** du module **os**[38] Comme vous pouvez le constater, la fonction **getcwd()** renvoie le nom du répertoire courant (**getcwd** = *get current working directory*).

Par comparaison, voici un exemple similaire utilisant la seconde méthode d'importation :

```
>>> from os import getcwd

>>> rep_cour = getcwd()

>>> print rep_cour
C:\Python22\essais
```

Dans ce nouvel exemple, nous n'avons importé du module **os** que la fonction **getcwd()** seule. Importée de cette manière, la fonction s'intègre à notre propre code comme si nous l'avions écrite nous-mêmes. Dans les lignes où nous l'utilisons, il n'est pas nécessaire de rappeler qu'elle fait partie du module **os**.

Nous pouvons de la même manière importer plusieurs fonctions du même module :

```
>>> from math import sqrt, pi, sin, cos
>>> print pi
3.14159265359
>>> print sqrt(5)         # racine carrée de 5
2.2360679775
>>> print sin(pi/6)       # sinus d'un angle de 30°
0.5
```

Nous pouvons même importer toutes les fonctions d'un module, comme dans :

```
from Tkinter import *
```

Cette méthode d'importation présente l'avantage d'alléger l'écriture du code. Elle présente l'inconvénient (surtout dans sa dernière forme, celle qui importe toutes les fonctions d'un module) d'encombrer l'espace de noms courant. Il se pourrait alors que certaines fonctions importées aient le même nom que celui d'une variable définie par vous-même, ou encore le même nom qu'une fonction importée depuis un autre module. (Si cela se produit, l'un des deux noms en conflit n'est évidemment plus accessible).

Dans les programmes d'une certaine importance, qui font appel à un grand nombre de modules d'origines diverses, il sera donc toujours préférable de privilégier plutôt la première méthode, c'est-à-dire celle qui utilise des noms pleinement qualifiés.

On fait généralement exception à cette règle dans le cas particulier du module Tkinter, parce que les fonctions qu'il contient sont très sollicitées (dès lors que l'on décide d'utiliser ce module).

Écriture séquentielle dans un fichier

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un objet-interface particulier, que l'on appelle *objet-fichier*. On crée cet objet à l'aide de la fonction intégrée **open()**[39]. Celle-ci renvoie un objet doté de méthodes spécifiques, qui vous permettront de lire et écrire dans le fichier.

L'exemple ci-dessous vous montre comment ouvrir un fichier « en écriture », y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé

[38] Le point séparateur exprime donc ici une relation d'appartenance. Il s'agit d'un exemple de la ***qualification des noms*** qui sera de plus en plus largement exploitée dans la suite de ce cours. Relier ainsi des noms à l'aide de points est une manière de désigner sans ambiguïté des éléments faisant partie d'ensembles, lesquels peuvent eux-mêmes faire partie d'ensembles plus vastes, etc. Par exemple, l'étiquette **systeme.machin.truc** désigne l'élément **truc**, qui fait partie de l'ensemble **machin**, lequel fait lui-même partie de l'ensemble **systeme**. Nous verrons de nombreux exemples de cette technique de désignation, notamment lors de notre étude des **classes** d'objets.

[39] Une telle fonction, dont la valeur de retour est un objet particulier, est souvent appelée *fonction-fabrique*.

automatiquement. Par contre, si le nom utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que vous y enregistrerez viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Vous pouvez faire tout cet exercice directement à la ligne de commande :

```
>>> obFichier = open('Monfichier','a')
>>> obFichier.write('Bonjour, fichier !')
>>> obFichier.write("Quel beau temps, aujourd'hui !")
>>> obFichier.close()
>>>
```

Notes :

- La première ligne crée l'*objet-fichier* **obFichier**, lequel fait référence à un fichier véritable (sur disque ou disquette) dont le nom sera **Monfichier**. Attention : *ne confondez pas le nom de fichier avec le nom de l'objet-fichier* qui y donne accès ! A la suite de cet exercice, vous pouvez vérifier qu'il s'est bien créé sur votre système (dans le répertoire courant) un fichier dont le nom est **Monfichier** (et dont vous pouvez visualiser le contenu à l'aide d'un éditeur quelconque).
- La fonction **open()** attend deux arguments, qui doivent tous deux être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture. `'a'` indique qu'il faut ouvrir ce fichier en mode « ajout » (*append*), ce qui signifie que les données à enregistrer doivent être ajoutées à la fin du fichier, à la suite de celles qui s'y trouvent éventuellement déjà. Nous aurions pu utiliser aussi le mode `'w'` (pour *write*), mais lorsqu'on utilise ce mode, Python crée toujours un nouveau fichier (vide), et l'écriture des données commence à partir du début de ce nouveau fichier. S'il existe déjà un fichier de même nom, celui-ci est effacé au préalable.
- La méthode **write()** réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier à accès séquentiel). Chaque nouvel appel de **write()** continue l'écriture à la suite de ce qui est déjà enregistré.
- La méthode **close()** referme le fichier. Celui-ci est désormais disponible pour tout usage.

Lecture séquentielle d'un fichier

Vous allez maintenant rouvrir le fichier, mais cette fois « en lecture », de manière à pouvoir y relire les informations que vous avez enregistrées dans l'étape précédente :

```
>>> ofi = open('Monfichier', 'r')
>>> t = ofi.read()
>>> print t
Bonjour, fichier !Quel beau temps, aujourd'hui !
>>> ofi.close()
```

Comme on pouvait s'y attendre, la méthode **read()** lit les données présentes dans le fichier et les transfère dans une variable de type « chaîne de caractères » (*string*) . Si on utilise cette méthode sans argument, la totalité du fichier est transférée.

Notes :

- Le fichier que nous voulons lire s'appelle **Monfichier**. L'instruction d'ouverture de fichier devra donc nécessairement faire référence à ce nom-là. Si le fichier n'existe pas, nous obtenons un message d'erreur. Exemple :

  ```
  >>> ofi = open('Monficier','r')
  IOError: [Errno 2] No such file or directory: 'Monficier'
  ```

  Par contre, nous ne sommes tenus à aucune obligation concernant le nom à choisir pour *l'ob-*

jet-fichier. C'est un nom de variable quelconque. Ainsi donc, dans notre première instruction, nous avons choisi de créer un objet-fichier **ofi**, faisant référence au fichier réel **Monfichier**, lequel est ouvert en lecture (argument `'r'`).

- Les deux chaînes de caractères que nous avions entrées dans le fichier sont à présent accolées en une seule. C'est normal, puisque nous n'avons fourni aucun caractère de séparation lorsque nous les avons enregistrées. Nous verrons un peu plus loin comment enregistrer des lignes de texte distinctes.
- La méthode **read()** peut également être utilisée avec un argument. Celui-ci indiquera combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier :

```
>>> ofi = open('Monfichier', 'r')
>>> t = ofi.read(7)
>>> print t
Bonjour
>>> t = ofi.read(15)
>>> print t
, fichier !Quel
```

S'il ne reste pas assez de caractères au fichier pour satisfaire la demande, la lecture s'arrête tout simplement à la fin du fichier :

```
>>> t = ofi.read(1000)
>>> print t
 beau temps, aujourd'hui !
```

Si la fin du fichier est déjà atteinte, **read()** renvoie une chaîne vide :

```
>>> t = ofi.read()
>>> print t

```

N'oubliez pas de refermer le fichier après usage :

```
>>> ofi.close()
```

L'instruction break pour sortir d'une boucle

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire ce fichier morceau par morceau, jusqu'à ce que l'on ait atteint la fin du fichier.

La fonction ci-dessous illustre cette idée. Elle copie l'intégralité d'un fichier, quelle que soit sa taille, en transférant des portions de 50 caractères à la fois :

```
def copieFichier(source, destination):
    "copie intégrale d'un fichier"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.read(50)
        if txt =="":
            break
        fd.write(txt)
    fs.close()
    fd.close()
    return
```

Si vous voulez tester cette fonction, vous devez lui fournir deux arguments : le premier est le nom du fichier original, le second est le nom à donner au fichier qui accueillera la copie. Exemple :

```
copieFichier('Monfichier','Tonfichier')
```

Vous aurez remarqué que la boucle **while** utilisée dans cette fonction est construite d'une manière différente de ce que vous avez rencontré précédemment. Vous savez en effet que l'instruction **while** doit toujours être suivie d'une condition à évaluer ; le bloc d'instructions qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie. Or nous avons remplacé ici la condition à évaluer par une simple constante, et vous savez également[40] que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro.

Une boucle **while** construite comme nous l'avons fait ci-dessus devrait donc boucler indéfiniment, puisque la condition de continuation reste toujours vraie. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction **break**, laquelle permet éventuellement de mettre en place plusieurs mécanismes de sortie différents pour une même boucle :

```
while <condition 1> :
    --- instructions diverses ---
    if <condition 2> :
        break
    --- instructions diverses ---
    if <condition 3>:
        break
    etc.
```

Dans notre fonction **copieFichier()**, il est facile de voir que l'instruction **break** s'exécutera seulement lorsque la fin du fichier aura été atteinte.

Fichiers texte

Un fichier texte est un fichier qui contient des caractères imprimables et des espaces organisés en lignes successives, ces lignes étant séparées les unes des autres par un caractère spécial non-imprimable appelé « marqueur de fin de ligne »[41].

Il est très facile de traiter ce genre de fichiers sous Python. Les instructions suivantes créent un fichier texte de quatre lignes :

```
>>> f = open("Fichiertexte", "w")
>>> f.write("Ceci est la ligne un\nVoici la ligne deux\n")
>>> f.write("Voici la ligne trois\nVoici la ligne quatre\n")
>>> f.close()
```

Notez bien le marqueur de fin de ligne **\n** inséré dans les chaînes de caractères, aux endroits où l'on souhaite séparer les lignes de texte dans l'enregistrement. Sans ce marqueur, les caractères seraient enregistrés les uns à la suite des autres, comme dans les exemples précédents.

Lors des opérations de lecture, les lignes d'un fichier texte peuvent être extraites séparément les unes des autres. La méthode **readline()**, par exemple, ne lit qu'une seule ligne à la fois (en incluant le caractère de fin de ligne) :

```
>>> f = open('Fichiertexte','r')
>>> t = f.readline()
```

[40] Voir page 63 : Véracité/fausseté d'une expression

[41] Suivant le système d'exploitation utilisé, le codage correspondant au marqueur de fin de ligne peut être différent. Sous *Windows*, par exemple, il s'agit d'une séquence de deux caractères (Retour chariot et Saut de ligne), alors que dans les systèmes de type *Unix* (comme *Linux*) il s'agit d'un seul saut de ligne, *MacOS* pour sa part utilisant un seul retour chariot. En principe, vous n'avez pas à vous préoccuper de ces différences. Lors des opérations d'écriture, Python utilise la convention en vigueur sur votre système d'exploitation. Pour la lecture, Python interprète correctement chacune des trois conventions (qui sont donc considérées comme équivalentes).

```
>>> print t
Ceci est la ligne un
>>> print f.readline()

Voici la ligne deux
```

La méthode **readlines()** transfère toutes les lignes restantes dans une liste de chaînes :

```
>>> t = f.readlines()
>>> print t
['Voici la ligne trois\012', 'Voici la ligne quatre\012']
>>> f.close()
```

Remarques :

- La liste apparaît ci-dessus en format brut, avec des apostrophes pour délimiter les chaînes, et les caractères spéciaux sous forme de codes numériques. Vous pourrez bien évidemment parcourir cette liste (à l'aide d'une boucle **while**, par exemple) pour en extraire les chaînes individuelles.
- La méthode **readlines()** permet donc de lire l'intégralité d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros (Puisqu'il est copié intégralement dans une variable, c'est-à-dire dans la mémoire vive de l'ordinateur, il faut que la taille de celle-ci soit suffisante). Si vous devez traiter de gros fichiers, utilisez plutôt la méthode **readline()** dans une boucle, comme le montrera l'exemple de la page suivante.
- Notez bien que **readline()** est une méthode qui renvoie une chaîne de caractères, alors que la méthode **readlines**() renvoie une liste. A la fin du fichier, **readline()** renvoie une chaîne vide, tandis que **readlines()** renvoie une liste vide.

Le script qui suit vous montre comment créer une fonction destinée à effectuer un certain traitement sur un fichier texte. En l'occurrence, il s'agit ici de recopier un fichier texte, en omettant toutes les lignes qui commencent par un caractère `'#'` :

```
def filtre(source,destination):

    "recopier un fichier en éliminant les lignes de remarques"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.readline()
        if txt =='':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
    return
```

Pour appeler cette fonction, vous devez utiliser deux arguments : le nom du fichier original, et le nom du fichier destiné à recevoir la copie filtrée. Exemple :

```
filtre('test.txt', 'test_f.txt')
```

Enregistrement et restitution de variables diverses

L'argument de la méthode **write()** doit être une chaîne de caractères. Avec ce que nous avons appris jusqu'à présent, nous ne pouvons donc enregistrer d'autres types de valeurs qu'en les transformant d'abord en chaînes de caractères (*string*).
Nous pouvons réaliser cela à l'aide de la fonction intégrée **str()** :

```
>>> x = 52
>>> f.write(str(x))
```

Nous verrons plus loin qu'il existe d'autres possibilités pour convertir des valeurs numériques en chaînes de caractères (voir à ce sujet : *Formatage des chaînes de caractères*, page 138). Mais la question n'est pas vraiment là. Si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les re-transformer correctement en valeurs numériques lorsque nous allons relire le fichier. Exemple :

```
>>> a = 5
>>> b = 2.83
>>> c = 67
>>> f = open('Monfichier', 'w')
>>> f.write(str(a))
>>> f.write(str(b))
>>> f.write(str(c))
>>> f.close()
>>> f = open('Monfichier', 'r')
>>> print f.read()
52.8367
>>> f.close()
```

Nous avons enregistré trois valeurs numériques. Mais comment pouvons-nous les distinguer dans la chaîne de caractères résultante, lorsque nous effectuons la lecture du fichier ? C'est impossible ! Rien ne nous indique d'ailleurs qu'il y a là trois valeurs plutôt qu'une seule, ou 2, ou 4, ...

Il existe plusieurs solutions à ce genre de problèmes. L'une des meilleures consiste à importer un module Python spécialisé : le module **pickle**[42]. Voici comment il s'utilise :

```
>>> import pickle
>>> f = open('Monfichier', 'w')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f = open('Monfichier', 'r')
>>> t = pickle.load(f)
>>> print t, type(t)
5 <type 'int'>
>>> t = pickle.load(f)
>>> print t, type(t)
2.83 <type 'float'>
>>> t = pickle.load(f)
>>> print t, type(t)
67 <type 'int'>
>>> f.close()
```

Pour cet exemple, on considère que les variables **a**, **b** et **c** contiennent les mêmes valeurs que dans l'exemple précédent. La fonction **dump()** du module **pickle** attend deux arguments : le premier est la variable à enregistrer, le second est l'objet fichier dans lequel on travaille. La fonction **pickle.load()** effectue le travail inverse, c'est-à-dire la restitution de chaque variable avec son type.

Vous pouvez aisément comprendre ce que font exactement les fonctions du module **pickle** en effectuant une lecture « classique » du fichier résultant, à l'aide de la méthode **read()** par exemple.

Gestion des exceptions. Les instructions try – except - else

Les *exceptions* sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché. Exemple :

[42] En anglais, le terme *pickle* signifie "conserver". Le module a été nommé ainsi parce qu'il sert effectivement à enregistrer des données en conservant leur type.

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

> *D'autres informations complémentaires sont affichées, qui indiquent notamment à quel endroit du script l'erreur a été détectée, mais nous ne les reproduisons pas ici.*

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme, et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à *tout un ensemble d'instructions*, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Un mécanisme de ce type s'appelle en général *mécanisme de traitement des exceptions*. Celui de Python utilise l'ensemble d'instructions **try** - **except** – **else** , qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit :

Le bloc d'instructions qui suit directement une instruction **try** est exécuté par Python *sous réserve*. Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction **except.** Si aucune erreur ne s'est produite dans les instructions qui suivent **try**, alors c'est le bloc qui suit l'instruction **else** qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se « plante ». Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
filename = raw_input("Veuillez entrer un nom de fichier : ")
try:
    f = open(filename, "r")
except:
    print "Le fichier", filename, "est introuvable"
```

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits d'un programme, nous pouvons aussi l'inclure dans une fonction :

```
def existe(fname):
    try:
        f = open(fname,'r')
        f.close()
        return 1
    except:
        return 0

filename = raw_input("Veuillez entrer le nom du fichier : ")
if existe(filename):
    print "Ce fichier existe bel et bien."
else:
    print "Le fichier", filename, "est introuvable."
```

Il est également possible de faire suivre l'instruction **try** de plusieurs blocs **except**, chacun d'entre eux traitant un type d'erreur spécifique, mais nous ne développerons pas ces complé-

ments ici. Veuillez consulter un ouvrage de référence sur Python si nécessaire.

9 *Exercices :*

9.1 Écrivez un script qui permette de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier.
L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche <Enter> pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide (c'est-à-dire utiliser la touche <Enter> seule).
L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître).

9.2 Considérons que vous avez à votre disposition un fichier texte contenant des phrases de différentes longueurs. Écrivez un script qui recherche et affiche la phrase la plus longue.

9.3 Écrivez un script qui génère automatiquement un fichier texte contenant les tables de multiplication de 2 à 30 (chacune d'entre elles incluant 20 termes seulement).

9.4 Écrivez un script qui recopie un fichier texte en triplant tous les espaces entre les mots.

9.5 Vous avez à votre disposition un fichier texte dont chaque ligne est la représentation d'une valeur numérique de type réel (mais sans exposants). Par exemple :

```
14.896
7894.6
123.278
etc.
```

Écrivez un script qui recopie ces valeurs dans un autre fichier en les arrondissant en nombres entiers (l'arrondi doit être correct).

9.6 Écrivez un script qui compare les contenus de deux fichiers et signale la première différence rencontrée.

9.7 A partir de deux fichiers préexistants A et B, construisez un fichier C qui contienne alternativement un élément de A, un élément de B, un élément de A, ... et ainsi de suite jusqu'à atteindre la fin de l'un des deux fichiers originaux. Complétez ensuite C avec les éléments restant sur l'autre.

9.8 Écrivez un script qui permette d'encoder un fichier texte dont les lignes contiendront chacune les noms, prénom, adresse, code postal et n° de téléphone de différentes personnes (considérez par exemple qu'il s'agit des membres d'un club)

9.9 Écrivez un script qui recopie le fichier utilisé dans l'exercice précédent, en y ajoutant la date de naissance et le sexe des personnes (l'ordinateur devra afficher les lignes une par une, et demander à l'utilisateur d'entrer pour chacune les données complémentaires).

9.10 Considérons que vous avez fait les exercices précédents et que vous disposez à présent d'un fichier contenant les coordonnées d'un certain nombre de personnes. Écrivez un script qui permette d'extraire de ce fichier les lignes qui correspondent à un code postal bien déterminé.

9.11 Modifiez le script de l'exercice précédent, de manière à retrouver les lignes correspondant à des prénoms dont la première lettre est située entre F et M (inclus) dans l'alphabet.

9.12 Écrivez des fonctions qui effectuent le même travail que celles du module pickle (voir page 119). Ces fonctions doivent permettre l'enregistrement de variables diverses dans un

fichier texte, en les accompagnant systématiquement d'informations concernant leur format exact.

10

Approfondir les structures de données

Jusqu'à présent, nous nous sommes contentés d'opérations assez simples. Nous allons maintenant passer à la vitesse supérieure. Les structures de données que vous utilisez déjà présentent quelques caractéristiques que vous ne connaissez pas encore, et il est également temps de vous faire découvrir d'autres structures plus complexes.

Le point sur les chaînes de caractères

Nous avons déjà rencontré les chaînes de caractères au chapitre 5. A la différence des données numériques, qui sont des entités singulières, les chaînes de caractères constituent *un type de donnée composite*. Nous entendons par là une entité bien définie qui est faite elle-même d'un ensemble d'entités plus petites, en l'occurrence : les caractères. Suivant les circonstances, nous serons amenés à traiter une telle donnée composite, tantôt comme un seul objet, tantôt comme une *suite ordonnée d'éléments*. Dans ce dernier cas, nous souhaiterons probablement pouvoir accéder à chacun de ces éléments à titre individuel.

En fait, les chaînes de caractères font partie d'une catégorie d'objets Python que l'on appelle des *séquences*, et dont font partie aussi les *listes* et les *tuples*. On peut effectuer sur les séquences tout un ensemble d'opérations similaires. Vous en connaissez déjà quelques unes, et nous allons en décrire quelques autres dans les paragraphes suivants.

Indiçage, extraction, longueur

Petit rappel du chapitre 5 : Les chaînes sont des *séquences* de caractères. Chacun de ceux-ci occupe une place précise dans la séquence. Sous Python, les éléments d'une séquence sont toujours *indicés* (ou numérotés) de la même manière, c'est-à-dire *à partir de zéro*. Pour extraire un caractère d'une chaîne, il suffit d'accoler au nom de la variable qui contient cette chaîne, son *indice* entre crochets :

```
>>> nom = 'Cédric'
>>> print nom[1], nom[3], nom[5]
é r c
```

Si l'on désire déterminer le nombre de caractères présents dans une chaîne, on utilise la fonction intégrée **len()** :

```
>>> print len(nom)
6
```

Nous avons cependant déjà attiré votre attention sur le fait que cela ne marche pas toujours comme prévu. Les résultats mentionnés dans les deux exemples ci-dessus seront peut-être ceux-là, mais seulement si vous travaillez dans un environnement fonctionnant toujours avec une

norme d'encodage ancienne, telle que *Windows-1252* ou *ISO-8859-1 (Latin-1)*. Par contre, si vous utilisez un poste de travail moderne, configuré de manière à utiliser l'encodage *Utf-8* par défaut, vous obtiendrez plutôt :

```
>>> nom = 'Cédric'
>>> print nom[1], nom[3], nom[5]
 d i

>>> print len(nom)
7
```

Ces résultats gênants peuvent être évités, mais il faut pour cela approfondir nos connaissances.

Les types « string » et « unicode »

À l'origine du développement des technologies informatiques, alors que les capacités de mémorisation des ordinateurs étaient encore assez limitées, on n'imaginait pas que ceux-ci seraient utilisés un jour pour traiter d'autres textes que des communications techniques, essentiellement en anglais. Il semblait donc tout-à-fait raisonnable de ne prévoir pour celles-ci qu'un jeu de caractères restreint, de manière à pouvoir représenter chacun de ces caractères avec un petit nombre de bits, et ainsi occuper aussi peu d'espace que possible dans les coûteuses unités de stockage de l'époque. Le jeu de caractères *ASCII*[43] fut donc choisi en ce temps là, avec l'estimation que 128 caractères suffiraient (à savoir le nombre de combinaisons possibles pour des groupes de 7 bits[44]). En l'étendant par la suite à 256 caractères, on put l'adapter aux exigences du traitement des textes écrits dans d'autres langues que l'anglais, mais au prix d'une dispersion des normes (ainsi par exemple, la norme *ISO-8859-1* codifie tous les caractères accentués du français ou de l'allemand (entre autres), mais aucun caractère grec, hébreu ou cyrillique. Pour ces langues, il faudra respectivement utiliser les normes *ISO-8859-7, ISO-8859-8, ISO-8859-5*, bien évidemment incompatibles, et d'autres normes encore pour l'arabe, le tchèque, le hongrois...).

L'intérêt résiduel de ces normes anciennes réside dans leur simplicité. Elles permettent en effet aux développeurs d'applications informatiques de considérer que *chaque caractère typographique est assimilable à un octet*, et que par conséquent une chaîne de caractères n'est rien d'autre qu'une séquence d'octets. C'est ainsi que fonctionne le type de données *string* de Python.

Toutefois, comme nous l'avons déjà évoqué sommairement au chapitre 5, les applications informatiques modernes ne peuvent plus se satisfaire de ces normes étriquées. Il faut désormais pouvoir encoder, dans un même texte, tous les caractères de n'importe quel alphabet de n'importe quelle langue. Une organisation internationale a donc été créée : le *Consortium Unicode*, laquelle a effectivement développé une norme universelle sous le nom de *Unicode*. Cette nouvelle norme vise à donner à tout caractère de n'importe quel système d'écriture de langue un nom et un identifiant numérique, et ce de manière unifiée, quelle que soit la plate-forme informatique ou le logiciel.

Une difficulté se présente, cependant. Se voulant universelle, la norme *Unicode* doit attribuer un identifiant numérique *différent* à plusieurs dizaines de milliers de caractères. Tous ces identifiants ne pourront évidemment pas être encodés sur un seul octet. À première vue, ils serait donc tentant de décréter qu'à l'avenir, chaque caractère devra être encodé sur deux octets (cela ferait 65536 possibilités), ou trois (16777216 possibilités) ou quatre (+ de 4 milliards de possibilités). Chacun de ces choix rigides entraîne cependant son lot d'inconvénients. Le premier, commun à tous, est que l'on perd la compatibilité avec la multitude de documents informatiques préexistants (et

[43] ASCII = *American Standard Code for Information Interchange*

[44] En fait, on utilisait déjà les octets à l'époque, mais l'un des bits de l'octet devait être réservé comme bit de contrôle pour les systèmes de rattrapage d'erreur. L'amélioration ultérieure de ces systèmes permit de libérer ce huitième bit pour y stocker de l'information utile : cela autorisa l'extension du jeu ASCII à 256 caractères (Normes ISO-8859, etc.).

notamment de logiciels), qui ont été encodés aux normes anciennes, sur la base du paradigme « un caractère égale un octet ». Le second est lié à l'impossibilité de satisfaire deux exigences contradictoires : si l'on se contente de deux octets, on risque de manquer de possibilités pour identifier des caractères rares ou des attributs de caractères qui seront probablement souhaités dans l'avenir ; si l'on impose trois, quatre octets ou davantage, par contre, on aboutit à un monstrueux gaspillage de ressources : la plupart des textes courants se satisfaisant d'un jeu de caractères restreint, l'immense majorité de ces octets ne contiendraient en effet que des zéros.

Afin de ne pas se retrouver piégée dans un carcan de ce genre, la norme *Unicode* ne fixe aucune règle concernant le nombre d'octets ou de bits à réserver pour l'encodage. Elle spécifie seulement la valeur numérique de l'identifiant associé à chaque caractère. En fonction des besoins, chaque système informatique est donc libre d'encoder « en interne » cet identifiant comme bon lui semble, par exemple sous la forme d'un entier ordinaire. Comme tous les langages de programmation modernes, Python s'est donc pourvu d'un type de données « chaîne de caractères Unicode » que nous désignerons désormais comme le type *unicode*.

Par opposition au type *string*, qui doit être compris plutôt comme *une séquence d'octets* (représentés il est vrai par des caractères, mais seulement lorsque c'est possible), le type *unicode* peut être véritablement être considéré comme *une séquence de caractères*, avec un statut identique pour chacun d'eux. Dans une chaîne *unicode* en effet, tous les caractères sont traités sur le même pied, qu'il s'agisse de caractères *ASCII* standards, de caractères accentués, grecs, cyrilliques, etc. Par exemple, si nous modifions un tout petit peu les deux derniers exemples de la rubrique précédente (en supposant toujours que nous travaillons sur un terminal moderne utilisant l'encodage Utf-8), nous pouvons obtenir les résultats attendus :

```
>>> nom = u'Cédric'
>>> print nom[1], nom[3], nom[5]
é r c

>>> print len(nom)
6
```

Qu'avons nous changé, au juste ? À vrai dire, pas grand chose : juste un petit préfixe **u** accolé à la chaîne littérale qui assigne un contenu à la variable **nom**. Ce préfixe devant une chaîne littérale indique à l'interpréteur Python que la variable à laquelle on affecte cette valeur doit être initialisée comme une variable de type *unicode*, et que la chaîne littérale qui suit doit donc être convertie au format interne *unicode*. Pour effectuer cette conversion, il est clair que l'interpréteur doit savoir dans quel encodage cette chaîne littérale lui est fournie. Si vous travaillez à la ligne de commande, comme c'est le cas dans notre exemple, Python considère que la chaîne est encodée avec l'encodage par défaut du terminal utilisé[45]. À l'intérieur d'un script, par contre, Python considère que l'encodage utilisé pour toutes les chaînes littérales de ce script est celui qui est déclaré à la première ou à la deuxième ligne, dans un « pseudo-commentaire » du genre : `-*- coding:Utf-8 -*-` ou `-*- coding:Latin-1 -*-`.

L'encodage Utf-8

À ce stade de nos explications, il devient urgent de préciser encore quelque chose.

Nous avons vu que la norme *Unicode* ne fixe en fait rien d'autre que des valeurs numériques, pour tous les identifiants standardisés destinés à désigner de manière univoque les caractères des alphabets du monde entier (plus de 240000 en novembre 2005). Elle ne précise en aucune façon la manière dont ces valeurs numériques doivent être encodées concrètement sous forme d'octets ou de bits.

[45] Si vous ne connaissez pas l'encodage utilisé par votre terminal, voyez page 44.

Pour le fonctionnement *interne* des applications informatiques, cela n'a pas d'importance. Les concepteurs de langages de programmation, de compilateurs ou d'interpréteurs pourront décider librement de représenter ces caractères par des entiers sur 8, 16, 24, 32, 64 bits, ou même (bien que l'on n'en voie pas l'intéret !) par des réels en virgule flottante : c'est leur affaire et cela ne nous concerne pas. Nous ne devons donc pas nous préoccuper du format réel des caractères, à l'intérieur d'une chaîne *unicode* de Python.

Il en va tout autrement, par contre, pour les *entrées/sorties*. Les développeurs que nous sommes devons absolument pouvoir préciser sous quelle forme exacte les données doivent être introduites dans nos programmes, que ce soit par l'intermédiaire de frappes au clavier ou par importation depuis une source quelconque. De même, nous devons pouvoir choisir le format des données que nous exportons vers n'importe quel dispositif périphérique, qu'il s'agisse d'une imprimante, d'un disque dur, d'un écran ...

Pour toutes ces entrées ou sorties de chaînes de caractères, nous devrons donc toujours considérer qu'il s'agit concrètement de séquences d'octets, et donc utiliser le type *string* de Python, en veillant à gérer convenablement *l'encodage* qui y sera utilisé.

Même si cela peut vous paraître à première vue un peu compliqué, dites-vous bien que malheureusement l'encodage idéal n'existe pas. En fonction de ce que l'on veut en faire, il peut être préférable d'encoder un même texte de plusieurs manières différentes. C'est pour cette raison qu'ont été définies, en parallèle avec la norme *Unicode*, plusieurs *normes d'encodage* : *Utf-8, Utf-16, Utf-32,* et quelques variantes. Ne vous affolez pas, cependant : vous ne serez vraisemblablement jamais confronté qu'à la première d'entre elles. Les autres ne concerneront que certains spécialistes de domaines « pointus ».

La norme d'encodage *Utf-8* est désormais la norme préférentielle pour la plupart des textes courants, parce que :

- d'une part, elle assure une parfaite compatibilité avec les textes encodés en « pur » *ASCII* (ce qui est le cas de nombreux codes sources de logiciels), ainsi qu'une compatibilité partielle avec les textes encodés à l'aide de ses variantes « étendues », telles que *Latin-1* ;
- d'autre part, cette nouvelle norme est celle qui est la plus économe en ressources, tout au moins pour les textes écrits dans une langue occidentale.

Suivant cette norme, les caractères du jeu *ASCII* standard sont encodés sur un seul octet. Les autres seront encodés en général sur deux octets, parfois trois ou même quatre pour les caractères plus rares.

Conversion (encodage/décodage) des chaînes

Nous entrons à présent dans le vif du sujet. Pour leur traitement à l'intérieur d'un script Python, il faudra fréquemment convertir les chaînes de caractères, du type *string* au type *unicode*, ou vice-versa. Vous devez donc savoir comment effectuer ces conversions. Python vous fournit fort heureusement les outils nécessaires, sous la forme de *méthodes* des objets concernés.

Conversion d'une chaîne string en chaîne unicode

Considérons un script dans lequel on utilise la fonction interne **raw_input()** pour accepter les entrées d'un utilisateur. Cette fonction renvoie toujours une valeur de type *string*, laquelle est encodée en *Latin-1*, en *Utf-8*, ou encore une autre norme, suivant l'encodage utilisé par défaut sur le poste de travail de cet utilisateur. Supposons par exemple qu'il s'agisse d'*Utf-8*. Pour convertir cette chaîne *Utf-8* en chaîne *unicode*, vous appliquerez à cet objet *string* la méthode **decode()**, avec l'argument `"Utf-8"` (Vous pouvez aussi utiliser `"utf-8"`, `"Utf8"` ou `"utf8"`). Exemple :

```
print "Veuillez entrer une chaîne avec des caractères accentués, svp :"
chs = raw_input()           # la chaîne est entrée est un objet de type string

chu = chs.decode("utf-8")   # conversion en chaîne unicode (décodage)
```

À la troisième ligne de cet exemple, la variable **chu** se voit affecter une chaîne de type *unicode*. Vous pouvez vous en assurer, par exemple en extrayant de cette chaîne l'un ou l'autre caractère accentué, ou en testant sa longueur comme nous l'avons fait dans les exemples précédents. Vous pouvez aussi plus simplement faire appel à la fonction interne **type()**. Ainsi par exemple, l'instruction :

```
print type(chs), type(chu)
```

ajoutée à la fin du petit script script ci-dessus, provoquerait l'affichage :

```
<type 'str'> <type 'unicode'>
```

Si le poste de travail de l'utilisateur utilise l'encodage *Latin-1* (Cas de nombreux postes *Windows*), vous pouvez également décoder la chaîne entrée. Il vous suffit de remplacer l'argument **"Utf-8"** par **"Latin-1"** (ou encore **"Latin1"**, **"latin-1"**, **"latin1"**).

Conversion d'une chaîne unicode en chaîne string

Considérons à présent que vous souhaitiez mémoriser une chaîne de caractères dans un fichier texte. Si la chaîne à mémoriser est du type *unicode* dans votre script, vous devez d'abord choisir un encodage, et la convertir en *string* avant de pouvoir l'enregistrer. Pour ce faire, vous appliquerez à cet objet *unicode* sa méthode **encode()**, avec l'argument correspondant à l'encodage souhaité. Par exemple, pour enregistrer des chaînes avec l'encodage *Utf-8*, vous ferez :

```
chu = u"Chaîne avec accents : déjà Noël !"    # chaîne unicode
chs = chu.encode("utf8")                      # conversion unicode -> string
of = open("MonFichier", "w")
of.write(chs)
of.close()
```

Attention : Vous ne pouvez pas enregistrer une chaîne *unicode* telle quelle dans un fichier texte, car le choix d'un encodage est indispensable. Si vous essayez par exemple :

```
chu = u"Chaîne avec accents : déjà Noël !"    # chaîne unicode
of = open("MonFichier", "w")
of.write(chu)
of.close()
```

vous allez obtenir un message d'erreur similaire au suivant :

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    of.write(chu)
UnicodeEncodeError: 'ascii' codec can't encode character u'\xee' in position 3:
ordinal not in range(128)
```

Explication : Du fait que vous ne précisez aucun encodage, Python essaie d'encoder la chaîne *unicode* **chu** avec l'encodage par défaut, à savoir presque toujours *ASCII*[46]. Cela ne marche évidemment pas avec les caractères accentués, ce qu'indique assez clairement le message d'erreur[47].

---

[46] Attention : cet encodage par défaut de Python reste toujours ASCII, même si vous avez pris la peine d'indiquer **votre** encodage par défaut en début de script, à l'aide d'une ligne telle que : **#** -*- coding:Utf-8 -*-
Une ligne de ce genre indique en effet à Python l'encodage que **vous** utilisez pour toutes les chaînes de texte littérale, les commentaires, etc. que vous insérez vous-même dans votre script. Elle ne force pas l'encodage des entrées/sorties. Il est possible de modifier ce comportement par défaut de Python, mais cela sort du cadre de ce manuel.

[47] En informatique, on appelle **codec** (*codeur/décodeur*) tout dispositif de conversion de format. Vous rencontrerez par exemple de nombreux codecs dans le monde du multimedia (codecs audio, vidéo, ...). Python dispose de nombreux codecs pour convertir les chaînes de caractères suivant les différentes normes en vigueur.

Conversion d'une chaîne Utf-8 en Latin-1, ou vice-versa

Si vous avez compris le fonctionnement des méthodes **decode()** et **encode()** décrites dans les rubriques précédentes, vous pouvez bien évidemment les appliquer à la conversion d'une chaîne de type *string*, d'un format d'encodage à un autre :

```
# Conversion Utf-8 -> Latin-1 :
chLat = chUtf.decode('Utf8').encode('Latin1')
```

Vous remarquerez encore une fois que la *composition* d'instructions permet d'éviter le passage par une variable intermédiaire. La chaîne d'origine **chUtf** est convertie en *unicode*, puis re-convertie en *string*, en une seule opération.

```
# Conversion Latin-1 -> Utf-8 :
chUtf = chLat.decode('Latin1').encode('Utf8')
```

Quand faut-il convertir ?

Si vous travaillez dans un environnement « ancien » utilisant toujours l'encodage *ISO-8859-1* par défaut (ce qui est encore le cas de nombreux postes de travail fonctionnant sous *Windows*), vous pouvez peut-être provisoirement ignorer la question, et vous contenter de traiter toutes vos chaînes de caractères en objets de type *string*.

Ce n'est cependant pas une bonne idée si vous avez l'ambition de vous tenir au courant de l'évolution des techniques informatiques. Vous devrez tôt ou tard vous préoccuper de rendre vos programmes compatibles avec les exigences de l'internationalisation. Et le plus tôt sera le mieux.

Et si votre poste de travail fonctionne avec un système d'exploitation aux normes modernes, tel *Ubuntu Linux*, par exemple, vous n'avez déjà plus le choix : vous *devez* comprendre ce que sont les types *string* et *unicode* de Python, et vous *devez* maîtriser leurs conversions réciproques.

Ce n'est finalement pas très compliqué : en résumé, il faut retenir que :

- Toutes les entrées/sorties de chaînes de caractères doivent être du type *string*.
- Avant d'effectuer sur une chaîne toute opération nécessitant *un accès à ses caractères individuels*, il faut impérativement la convertir au préalable en objet *unicode*.
- Les opérations qui ne nécessitent pas un accès aux caractères individuels peuvent souvent encore être effectuées directement sur les chaînes de type *string*, sans conversion préalable en *unicode*.
- Convertir une chaîne *unicode* en *string* constitue un *encodage*. On l'effectue à l'aide de la méthode **encode()** appliquée à cet objet. Exemple : `chs = chu.encode('Latin-1')`.
- Convertir une chaîne de type *string* en *unicode* constitue un *décodage*. On l'effectue en appliquant la méthode **decode()** à cet objet. Exemple : `chu = chs.decode('Utf-8')`.

Dans la suite de ce chapitre, nous rencontrerons divers exemples d'application de ces règles. Vous vous familiariserez vite avec elles.

> *À l'avenir, il sera certainement plus rationnel d'effectuer toutes les manipulations de chaînes à l'intérieur de vos programmes à l'aide du seul type unicode, et de réserver le type string aux entrées/sorties. Dans la suite de ce livre, cependant, nous utiliserons encore beaucoup le type string, essentiellement pour des raisons « historiques » : la plus grande grande partie des scripts que nous vous y présentons comme exemples ont en effet été développés sur des machines utilisant toujours la norme ancienne Latin-1. Nous n'y avons ajouté des conversions qu'aux seuls endroits où cela s'avérait indispensable pour que ces scripts s'exécutent correctement sur des machines récentes. Dans vos propres scripts, prenez l'habitude d'utiliser*

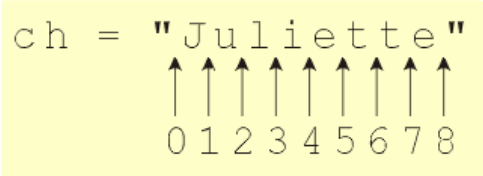
> *préférentiellement le type unicode pour traiter toutes vos chaînes de caractères.*

Extraction de fragments de chaînes

Il arrive fréquemment, lorsque l'on travaille avec des chaînes, que l'on souhaite extraire une petite chaîne hors d'une chaîne plus longue. Python propose pour cela une technique simple que l'on appelle *slicing* (« découpage en tranches »). Elle consiste à indiquer entre crochets les indices correspondant au début et à la fin de la « tranche » que l'on souhaite extraire :

```
>>> ch = "Juliette"
>>> print ch[0:3]
Jul
```

Dans la tranche **[n,m]**, le n-ième caractère est inclus, mais pas le m-ième. Si vous voulez mémoriser aisément ce mécanisme, il faut vous représenter que les indices pointent des emplacements situés ***entre*** les caractères, comme dans le schéma ci-dessous :



Au vu de ce schéma, il n'est pas difficile de comprendre que **ch[3:7]** extraira « **iett** »

Les indices de découpage ont des valeurs par défaut : un premier indice non défini est considéré comme zéro, tandis que le second indice omis prend par défaut la taille de la chaîne complète :

```
>>> print ch[:3]       # les 3 premiers caractères
Jul
>>> print ch[3:]       # tout ce qui suit les 3 premiers caractères
iette
```

Attention : comme nous l'avons expliqué dans les rubriques précédentes, les manipulations de chaînes qui impliquent un accès à leurs caractères individuels ne fonctionneront pas correctement sur les chaînes de type *string*, si celles-ci sont encodées suivant une norme récente telle que *Utf-8*. Avant d'effectuer une opération de *slicing* sur une chaîne, comme décrit ci-dessus, il faudra donc dorénavant veiller à convertir d'abord celle-ci en *unicode*. Exemple (test effectué sous encodage Utf-8) :

```
>>> chs = 'Adélaïde'
>>> print chs[:3], chs[4:8]
Ad laï
```

Ce résultat est à l'évidence incorrect. Mais si nous faisons :

```
>>> chu = chs.decode('Utf-8')
>>> print chu[:3], chu[4:8]
Adé aïde
```

Le résultat est cette fois tout à fait correct. Si la sous-chaîne à extraire doit être ensuite ré-encodée en objet de type *string*, nous pouvons utiliser la *composition* pour écrire une instruction compacte :

```
>>> chs1 = 'Cunégonde'
>>> chs2 = chs1.decode('utf8')[:4].encode('utf8')
>>> print chs1, type(chs1), chs2, type(chs2)
Cunégonde <type 'str'> Cuné <type 'str'>
```

À la deuxième ligne de cet exemple, la chaîne **chs1** est d'abord convertie en *unicode*, puis une tranche de 4 caractères en est extraite par *slicing*, et enfin cette tranche est elle-même reconvertie en *string* (encodé au format *Utf-8*).

Concaténation, Répétition

Les chaînes peuvent être *concaténées* avec l'opérateur + et *répétées* avec l'opérateur * :

```
>>> n = 'abc' + 'def'       # concaténation
>>> m = 'zut ! ' * 4        # répétition
>>> print n, m
abcdef zut ! zut ! zut ! zut !
```

Remarquez au passage que les opérateurs + et * peuvent aussi être utilisés pour l'addition et la multiplication lorsqu'ils s'appliquent à des arguments numériques. Le fait que les mêmes opérateurs puissent fonctionner différemment en fonction du contexte dans lequel on les utilise est un mécanisme fort intéressant que l'on appelle *surcharge des opérateurs*. Dans d'autres langages, la surcharge des opérateurs n'est pas toujours possible : on doit alors utiliser des symboles différents pour l'addition et la concaténation, par exemple.

10 Exercices

10.1 Déterminez vous-même ce qui se passe, dans la technique de *slicing*, lorsque l'un ou l'autre des indices de découpage est erroné, et décrivez cela le mieux possible. (Si le second indice est plus petit que le premier, par exemple, ou bien si le second indice est plus grand que la taille de la chaîne).

10.2 Découpez une grande chaîne en fragments de 5 caractères chacun. Rassemblez ces morceaux dans l'ordre inverse. La chaîne doit pouvoir contenir des caractères accentués.

10.3 Tâchez d'écrire une petite fonction **trouve()** qui fera exactement le contraire de ce que fait l'opérateur d'indexage (c'est-à-dire les crochets **[ ]**). Au lieu de partir d'un index donné pour retrouver le caractère correspondant, cette fonction devra retrouver l'index correspondant à un caractère donné.
En d'autres termes, il s'agit d'écrire une fonction qui attend deux arguments : le nom de la chaîne à traiter et le caractère à trouver. La fonction doit fournir en retour l'index du premier caractère de ce type dans la chaîne. Ainsi par exemple, l'instruction :
`print trouve("Juliette & Roméo", "&")` devra afficher : 9
Attention : Il faut penser à tous les cas possibles. Il faut notamment veiller à ce que la fonction renvoie une valeur particulière (par exemple la valeur -1) si le caractère recherché n'existe pas dans la chaîne traitée. Les caractères accentués doivent être acceptés.

10.4 Améliorez la fonction de l'exercice précédent en lui ajoutant un troisième paramètre : l'index à partir duquel la recherche doit s'effectuer dans la chaîne. Ainsi par exemple, l'instruction :
`print trouve ("César & Cléopâtre", "r", 5)` devra afficher : 15 (et non 4 !)

10.5 Écrivez une fonction **compteCar()** qui compte le nombre d'occurrences d'un caractère donné dans une chaîne. Ainsi :
`print compteCar("ananas au jus","a")` devra afficher : 4
`print compteCar("Gédéon est déjà là","é")` devra afficher : 3

10.6 Écrivez un programme qui recopie un fichier source, encodé à l'origine en *Latin-1*, dans un nouveau fichier destinataire, encodé cette fois en *Utf-8*, en effectuant au passage un traitement de toutes ses lignes : dans celles-ci, chaque caractère « espace » sera remplacé par le groupe de 3 caractères « -*- ».
Le programme demandera les noms des fichiers à l'utilisateur.

Parcours d'une séquence. L'instruction for ... in ...

Il arrive très souvent que l'on doive traiter l'intégralité d'une chaîne caractère par caractère, du premier jusqu'au dernier, pour effectuer à partir de chacun d'eux une opération quelconque. Nous appellerons cette opération un *parcours*. En nous limitant aux outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours à l'aide d'une boucle, articulée autour de l'instruction **while** :

```
nom = u'Joséphine'                # chaîne unicode, de préférence
index = 0
while index < len(nom):
    print nom[index] + ' *',
    index = index +1
```

Cette boucle *parcourt* donc la chaîne **nom** pour en extraire un à un tous les caractères, lesquels sont ensuite imprimés avec interposition d'astérisques. Notez bien que la condition utilisée avec l'instruction **while** est `index < len(nom)`, ce qui signifie que le bouclage doit s'effectuer jusqu'à ce que l'on soit arrivé à l'indice numéro 9 (la chaîne compte en effet 10 caractères). Nous aurons effectivement traité tous les caractères de la chaîne, puisque ceux-ci sont indicés de zéro à 9.

Le *parcours d'une séquence* est une opération très fréquente en programmation. Pour en faciliter l'écriture, Python vous propose une structure de boucle plus appropriée que la boucle **while**, basée sur le couple d'instructions **for ... in ... :**

Avec ces instructions, le programme ci-dessus devient :

```
nom = u'Joséphine'
for caract in nom:
    print caract + ' *',
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle vous évite d'avoir à définir et à incrémenter une variable spécifique (un « compteur ») pour gérer l'indice du caractère que vous voulez traiter à chaque itération (C'est Python qui s'en charge). La structure **for ... in ...** ne montre donc que l'essentiel, à savoir que variable **caract** contiendra successivement tous les caractères de la chaîne, du premier jusqu'au dernier.

L'instruction **for** permet donc d'écrire des boucles, dans lesquelles *l'itération traite successivement tous les éléments d'une séquence donnée*. Dans l'exemple ci-dessus, la séquence était une chaîne de caractères. L'exemple ci-après démontre que l'on peut appliquer le même traitement aux *listes* (et il en sera de même pour les *tuples* étudiés plus loin) :

```
liste = ['chien', 'chat', 'crocodile', u'éléphant']
for animal in liste:
    print 'longueur de la chaîne', animal, '=', len(animal)
```

L'exécution de ce script donne :

```
longueur de la chaîne chien = 5
longueur de la chaîne chat = 4
longueur de la chaîne crocodile = 9
longueur de la chaîne éléphant = 8
```

L'instruction **for ... in ... :** est un nouvel exemple d'*instruction composée*. N'oubliez donc pas le double point obligatoire à la fin de la ligne, et l'indentation pour le bloc d'instructions qui suit.

Le nom qui suit le mot réservé **in** est celui de la séquence qu'il faut traiter. Le nom qui suit le mot réservé **for** est celui que vous choisissez pour la *variable* destinée à contenir successivement tous les éléments de la séquence. Cette variable est définie automatiquement (c'est-à-dire qu'il est inutile de la définir au préalable), et *son type est automatiquement adapté* à celui de l'élément de la séquence qui est en cours de traitement (rappelons en effet que dans le cas d'une liste, tous les éléments ne sont pas nécessairement du même type). Exemple :

```
divers = ['cheval', u'lézard', 3, 17.25, [5, 'Jean']]
for e in divers:
    print e, type(e)
```

L'exécution de ce script donne :

```
cheval <type 'str'>
lézard <type 'unicode'>
3 <type 'int'>
17.25 <type 'float'>
[5, 'Jean'] <type 'list'>
```

Bien que les éléments de la liste **divers** soient tous de types différents (une chaîne de caractères string, une chaîne unicode, un entier, un réel, une liste), on peut affecter successivement leurs contenus à la variable **e**, sans qu'il s'ensuive des erreurs (ceci est rendu possible grâce au *typage dynamique* des variables Python).

Exercices :

10.7 Dans un conte américain, huit petits canetons s'appellent respectivement : *Jack, Kack, Lack, Mack, Nack, Oack, Pack et Qack*. Écrivez un petit script qui génère tous ces noms à partir des deux chaînes suivantes :

`prefixes = 'JKLMNOP'` et `suffixe = 'ack'`

Si vous utilisez une instruction **for ... in ...** , votre script ne devrait comporter que deux lignes.

10.8 Écrivez un script qui recherche le nombre de mots contenus dans une phrase donnée.

Appartenance d'un élément à une séquence. L'instruction in utilisée seule

L'instruction **in** peut être utilisée indépendamment de **for**, pour *vérifier si un élément donné fait partie ou non d'une séquence*. Vous pouvez par exemple vous servir de **in** pour vérifier si tel caractère alphabétique fait partie d'un groupe bien déterminé :

```
car = "e"
voyelles = "aeiouyAEIOUY"
if car in voyelles:
    print car, "est une voyelle"
```

D'une manière similaire, vous pouvez vérifier l'appartenance d'un élément à une liste :

```
n = 5
premiers = [1, 2, 3, 5, 7, 11, 13, 17]
if n in premiers:
    print n, "fait partie de notre liste de nombres premiers"
```

> *Note : Cette instruction très puissante effectue donc à elle seule un véritable parcours de la séquence. À titre d'exercice, écrivez les instructions qui effectueraient le même travail à l'aide d'une boucle classique utilisant l'instruction while.*

Exercices :

10.9 Écrivez une fonction **chiffre()** qui renvoie « vrai » si l'argument transmis est un chiffre.

10.10 Écrivez une fonction **majuscule()** qui renvoie « vrai » si l'argument transmis est une majuscule. *Tâchez de tenir compte des majuscules accentuées !*

10.11 Écrivez une fonction **chaineListe()** qui convertisse une phrase en une liste de mots.

10.12 Utilisez les fonctions définies dans les exercices précédents pour écrire un script qui puisse extraire d'un texte tous les mots qui commencent par une majuscule.

10.13 Utilisez les fonctions définies dans les exercices précédents pour écrire une fonction qui renvoie le nombre de caractères majuscules contenus dans une phrase donnée en argument.

Les chaînes sont des séquences non modifiables

Vous ne pouvez pas modifier le contenu d'une chaîne existante. En d'autres termes, vous ne pouvez pas utiliser l'opérateur **[ ]** dans la partie gauche d'une instruction d'affectation. Essayez par exemple d'exécuter le petit script suivant (qui cherche intuitivement à remplacer une lettre dans une chaîne) :

```
salut = 'bonjour à tous'
salut[0] = 'B'
print salut
```

Le résultat attendu par le programmeur qui a écrit ces instructions est « Bonjour à tous ». Mais contrairement à ses attentes, ce script *lève* une erreur du genre : *TypeError: object doesn't support item assignment*. Cette erreur est provoquée à la deuxième ligne du script. On y essaie de remplacer une lettre par une autre dans la chaîne, mais cela n'est pas permis.

Par contre, le script ci-dessous fonctionne parfaitement :

```
salut = 'bonjour à tous'
salut = 'B' + salut[1:]
print salut
```

Dans cet autre exemple, en effet, nous ne modifions pas la chaîne **salut**. Nous en re-créons une nouvelle avec le même nom à la deuxième ligne du script (à partir d'un morceau de la précédente, soit, mais qu'importe : il s'agit bien d'une *nouvelle* chaîne).

Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux (c'est-à-dire les instructions **if ... elif ... else**) fonctionnent aussi avec les chaînes de caractères. Cela peut vous être utile pour trier des mots par ordre alphabétique :

```
mot = raw_input("Entrez un mot quelconque : ")
if mot < "limonade":
    place = "précède"
elif mot > "limonade":
    place = "suit"
else:
    place = "se confond avec"
print "Le mot", mot, place, "le mot 'limonade' dans l'ordre alphabétique"
```

Ces comparaisons sont possibles, parce que dans toutes les normes d'encodage, les codes numériques représentant les caractères ont été attribués dans l'ordre alphabétique, tout au moins pour les caractères non accentués. Dans le système de codage *ASCII*, par exemple, A=65, B=66, C=67, etc.

Comprenez cependant que cela ne fonctionne bien que pour des mots qui sont tous entièrement en minuscules, ou entièrement en majuscules, et qui ne comportent aucun caractère accentué. Vous savez en effet que les majuscules et minuscules utilisent des ensembles de codes distincts. Quant aux caractères accentués, vous avez vu qu'ils sont encodés tout-à-fait en dehors de l'ensemble constitué par les caractères du standard *ASCII*. Construire un algorithme de tri alphabétique qui prenne en compte à la fois la casse des caractères et tous leurs accents n'est donc pas une mince affaire !

Classement des caractères

À ce stade, il peut être utile de s'intéresser aux valeurs des identifiants numériques associés à chaque caractère. Cela peut simplifier parfois certains algorithmes, notamment dans le cas où les chaînes traitées n'utilisent que des caractères strictement *ASCII*.

Afin que vous puissiez effectuer plus aisément toutes sortes de traitements sur les caractères, Python met à votre disposition un certain nombre de fonctions prédéfinies :

La fonction **ord(ch)** accepte n'importe quel caractère (*string* ou *unicode*) comme argument. En retour, elle fournit la valeur de l'identifiant numérique correspondant à ce caractère. Ainsi `ord('A')` renvoie la valeur 65, et `ord(u'Ĩ')` renvoie la valeur 296.

La fonction **unichr(num)** fait exactement le contraire, pour toute valeur 'légale' **num** correspondant effectivement à un caractère dans la norme *unicode*. Ainsi par exemple, `unichr(65)` renvoie le caractère **A**, et `unichr(1046)` renvoie le caractère cyrillique **Ж** .

La fonction **chr(num)** fait la même chose, mais uniquement pour les caractères strictement *ASCII*. L'argument qu'on lui transmet doit donc être un entier compris entre 32 et 127. Cette fonction obsolète ne devrait plus être utilisée désormais.

Vous pouvez exploiter ces fonctions prédéfinies pour explorer le jeu de caractères disponible sur votre ordinateur. Vous pouvez par exemple retrouver les caractères minuscules de l'alphabet grec, en sachant que les codes qui leur sont attribués vont de 945 à 969. Ainsi le petit script ci-dessous :

```
s = u""                     # chaîne unicode vide
i = 945                     # premier code
while i <= 969:             # dernier code
    s += unichr(i)
    i = i + 1
print "Alphabet grec (minuscule) : ", s
```

devrait afficher le résultat suivant :

```
Alphabet grec (minuscule) :  αβγδεζηθικλμνξοπρςστυφχψω
```

Exercices :

10.14 Écrivez un petit script qui affiche une table des codes *ASCII*. Le programme doit afficher tous les caractères en regard des codes correspondants. A partir de cette table, établissez la relation numérique reliant chaque caractère majuscule à chaque caractère minuscule.

10.15 A partir de la relation trouvée dans l'exercice précédent, écrivez une fonction qui convertit tous les caractères d'une phrase donnée en minuscules.

10.16 A partir de la même relation, écrivez une fonction qui convertit tous les caractères minuscules en majuscules, et vice-versa (dans une phrase fournie en argument).

10.17 Écrivez une fonction qui compte le nombre de fois qu'apparaît tel caractère (fourni en argument) dans une phrase donnée.

10.18 Écrivez une fonction **voyelle(cu)**, qui renvoie « vrai » si le caractère *unicode* fourni en argument est une voyelle.

10.19 Écrivez une fonction **compteVoyelles(chu)**, qui renvoie le nombre de voyelles contenues dans une phrase donnée, fournie à la fonction sous forme de chaîne *unicode.*

10.20 La relation entre majuscules et minuscules trouvée à l'exercice 10.14 reste-t-elle valable pour les caractères accentués du français ? Vérifiez-le en écrivant un script qui convertisse la chaîne unicode « àéèêëîïôöûüù » en son équivalent majuscule.

Les chaînes sont des objets

Dans les chapitres précédents, vous avez déjà rencontré de nombreux *objets*. Vous savez donc que l'on peut agir sur un objet à l'aide de *méthodes* (c'est-à-dire des fonctions associées à cet objet).
Sous Python, les chaînes de caractères sont des objets. On peut donc effectuer de nombreux traitements sur les chaînes de caractères en utilisant des méthodes appropriées. En voici quelques-unes, choisies parmi les plus utiles[48] . Elles fonctionnent en général de la même manière sur les chaînes *string* et les chaînes *unicode*, bien que l'on doive évidemment s'attendre à rencontrer quelques résultats incongrus pour les chaînes de type *string* contenant des caractères accentués :

- **split()** : convertit une chaîne en une liste de sous-chaînes. On peut choisir le caractère séparateur en le fournissant comme argument, sinon c'est un espace, par défaut :

```
>>> c2 ="Votez pour moi"
>>> a = c2.split()
>>> print a
['Votez', 'pour', 'moi']
>>> c4 ="Cet exemple, parmi d'autres, peut encore servir"
>>> c4.split(",")
['Cet exemple', " parmi d'autres", ' peut encore servir']
```

- **join(liste)** : rassemble une liste de chaînes en une seule (Cette méthode fait donc l'inverse de la précédente). Attention : la chaîne à laquelle on applique cette méthode est celle qui servira de séparateur (un ou plusieurs caractères) ; l'argument transmis est la liste des chaînes à rassembler :

```
>>> b2 = ["Salut", "les", "copains"]
>>> print  " ".join(b2)
Salut les copains
>>> print  "---".join(b2)
Salut---les---copains
```

- **find(sch)** : cherche la position d'une sous-chaîne **sch** dans la chaîne :

```
>>> ch1 = "Cette leçon vaut bien un fromage, sans doute ?"
>>> ch2 = "fromage"
>>> print ch1.find(ch2)
26
```

Attention ! Comme on peut le constater dans l'exemple ci-dessus, le résultat est incorrect si la chaîne traitée est de type *string* et encodée en *Utf-8*, car dans cet encodage les caractères *non-ASCII* sont codés sur 2 octets. Le résultat correct n'est garanti que pour les chaînes *unicode* :

[48] Il s'agit de quelques exemples seulement. La plupart de ces méthodes peuvent être utilisées avec différents paramètres que nous n'indiquons pas tous ici (par exemple, certains paramètres permettent de ne traiter qu'une partie de la chaîne). Vous pouvez obtenir la liste complète de toutes les méthodes associées à un objet à l'aide de la fonction intégrée **dir()**. Veuillez consulter l'un ou l'autre des ouvrages de référence (ou la documentation en ligne de Python) si vous souhaitez en savoir davantage.

```
>>> ch1 = u"Cette leçon vaut bien un fromage, sans doute ?"
>>> ch2 = u"fromage"
>>> print ch1.find(ch2)
25
```

- **count(sch)** : compte le nombre de sous-chaînes **sch** dans la chaîne :

```
>>> ch1 = "Le héron au long bec emmanché d'un long cou"
>>> ch2 = 'long'
>>> print ch1.count(ch2)
2
```

- **lower()** : convertit une chaîne en minuscules :

```
>>> ch = "CÉLIMÈNE est un prénom ancien"
>>> print ch.lower()
cÉlimÈne est un prénom ancien
```

On voit que le résutat n'est pas parfait avec le type string. Essayons en unicode :

```
>>> ch = u"CÉLIMÈNE est un prénom ancien"
>>> print ch.lower()
célimène est un prénom ancien
```

- **upper()** : convertit une chaîne en majuscules :

```
>>> ch = "Maître Jean-Noël Hébèrt"
>>> print ch.upper()
MAîTRE JEAN-NOëL HéBèRT

>>> ch = u"Maître Jean-Noël Hébèrt"
>>> print ch.upper()
MAÎTRE JEAN-NOËL HÉBÈRT
```

Même remarque que pour la fonction précédente : il faut privilégier *unicode.*

- **title()** : convertit en majuscule l'initiale de chaque mot :

```
>>> ch ="albert rené élise véronique"
>>> print ch.title()
Albert René éLise VéRonique

>>> ch =u"albert rené élise véronique"
>>> print ch.title()
Albert René Élise Véronique
```

Même remarque encore.

- **capitalize()** : variante de la précédente. Convertit en majuscule seulement la première lettre :

```
>>> b3 = "quel beau temps, aujourd'hui !"
>>> print b3.capitalize()
"Quel beau temps, aujourd'hui !"
```

- **swapcase()** : convertit toutes les majuscules en minuscules, et vice-versa :

```
>>> ch = "Le Lièvre Et La Tortue"
>>> print ch.swapcase()
lE lIèVRE eT lA tORTUE

>>> ch = u"Le Lièvre Et La Tortue"
```

```
>>> print ch.swapcase()
lE lIÈVRE eT lA tORTUE
```

- **strip()** : enlève les espaces éventuels au début et à la fin de la chaîne :
```
>>> ch = "   Monty Python   "
>>> ch.strip()
'Monty Python'
```

- **replace(c1, c2) :** remplace tous les caractères **c1** par des caractères **c2** dans la chaîne :

```
>>> ch8 = "Si ce n'est toi c'est donc ton frère"
>>> print ch8.replace(" ","*")
Si*ce*n'est*toi*c'est*donc*ton*frère
```

- **index(car)** : retrouve l'index de la première occurrence du caractère **car** dans la chaîne :
```
>>> ch9 ="Portez ce vieux whisky au juge blond qui fume"
>>> print ch9.index("w")
16
```

  Comme signalé déjà à plusieurs reprises, une fonction de ce genre peut renvoyer un résultat incorrect avec certaines chaînes de type *string*.

Dans la plupart de ces méthodes, il est possible de préciser quelle portion de la chaîne doit être traitée, en ajoutant des arguments supplémentaires. Exemples :

```
>>> print ch9.index("e")       # cherche à partir du début de la chaîne
4                              # et trouve le premier 'e'
>>> print ch9.index("e",5)     # cherche seulement à partir de l'indice 5
8                              # et trouve le second 'e'
>>> print ch9.index("e",15)    # cherche à partir du caractère n° 15
29                             # et trouve le quatrième 'e'
```

Etc., etc.

Comprenez bien qu'il n'est pas possible de décrire toutes les méthodes disponibles, ainsi que leur paramétrage, dans le cadre restreint de ce cours. Si vous souhaitez en savoir davantage, il vous faut consulter la documentation en ligne de Python (*Library reference*), ou un bon ouvrage de référence.

Fonctions intégrées

A toutes fins utiles, rappelons également ici que l'on peut aussi appliquer aux chaînes un certain nombre de fonctions intégrées dans le langage :

- **len(ch)** renvoie la longueur de la chaîne **ch** (Comprenez bien qu'il ne s'agit toujours du nombre de caractères, que pour une chaîne de type unicode. Pour une chaîne de type string, il s'agit du nombre d'octets, lequel peut être très différent, en particulier avec les encodages Utf-16 et Utf-32).

- **float(ch)** convertit la chaîne **ch** en un nombre réel (*float*)
(bien entendu, cela ne pourra fonctionner que si la chaîne représente bien un tel nombre) :

```
>>> a = float("12.36")      # Attention : pas de virgule décimale !
>>> print a + 5
17.36
```

- **int(ch)** convertit la chaîne **ch** en un nombre entier :

```
>>> a = int("184")
>>> print a + 20
204
```

- **str(obj)** convertit l'objet **obj** en une chaîne de type *string*. **obj** peut être une donnée de tout type :

```
>>> n, l = 15, [17.334, "zut"]
>>> ch = str(n) + " est un entier, et " + str(l) + " est une liste."
>>> print ch
15 est un entier, et [17.334, 'zut'] est une liste.
```

- **unicode(obj)** convertit l'objet **obj** en une chaîne de type *unicode*. Même utilité que ci-dessus.

```
>>> n, l = 15, [17.334, "zut"]
>>> ch = unicode(n) + " est un entier, et " + unicode(l) + " est une liste."
>>> print ch, type(ch)
15 est un entier, et [17.334, 'zut'] est une liste. <type 'unicode'>
```

  Remarquons au passage que la concaténation de chaînes des types *string* et *unicode* est possible, le résultat étant dans ce cas une chaîne du type *unicode*.

Formatage des chaînes de caractères

Pour terminer ce tour d'horizon des fonctionnalités associées aux chaînes de caractères, il nous semble intéressant de vous présenter encore une technique que l'on appelle *formatage des chaînes*. Cette technique se révèle particulièrement utile dans tous les cas où vous devez construire une chaîne de caractères complexe à partir d'un certain nombre de morceaux, tels que les valeurs de variables diverses.

Considérons par exemple que vous avez écrit un programme qui traite de la couleur et de la température d'une solution aqueuse, en chimie. La couleur est mémorisée dans une chaîne de caractères nommée **coul**, et la température dans une variable nommée **temp** (variable de type *float*). Vous souhaitez à présent que votre programme construise une nouvelle chaîne de caractères à partir de ces données, par exemple une phrase telle que la suivante : « La solution est devenue rouge et sa température atteint 12,7 °C ».

Vous pouvez construire cette chaîne en assemblant des morceaux à l'aide de l'opérateur de concaténation (le symbole +), mais il vous faudra aussi utiliser plusieurs fois l'une des fonctions intégrées **str()** ou **unicode()** pour convertir en chaîne de caractères la valeur numérique contenue dans la variable de type *float* (faites l'exercice).

Python vous offre une autre possibilité. Vous pouvez construire votre chaîne en assemblant deux éléments à l'aide de l'opérateur **%** . À gauche de cet opérateur, vous fournissez *une chaîne de formatage* (un patron, en quelque sorte) qui contient des marqueurs de conversion, et à droite (entre parenthèses) le ou les objets que Python devra insérer dans la chaîne, en lieu et place des marqueurs. Exemple :

```
>>> coul ="verte"
>>> temp = 1.347 + 15.9
>>> print "La couleur est %s et la température vaut %s °C" % (coul, temp)
La couleur est verte et la température vaut 17.247 °C
```

Dans cet exemple, la chaîne de formatage contient deux marqueurs de conversion **%s** , qui seront remplacés respectivement par les contenus des deux variables **coul** et **temp**.

Le marqueur **%s** accepte n'importe quel objet (chaîne, entier, float, liste, ...). Vous pouvez expérimenter d'autres mises en forme, plus intéressantes encore, en utilisant d'autres marqueurs. Essayez par exemple de remplacer le deuxième **%s** par **%d** , ou par **%8.2f** , ou encore par **%8.2g** . Le marqueur **%d** attend un nombre et le convertit en entier ; les marqueurs **%f** et **%g** attendent des nombres réels et peuvent déterminer la largeur et la précision qui seront affichées.

La description complète de toutes ces possibilités de *formatage* sort du cadre de ces notes. S'il vous faut un formatage très particulier, veuillez consulter la documentation en ligne de Python ou des manuels plus spécialisés.

Le formatage fonctionne bien évidemment aussi avec des chaînes *unicode* :

```
>>> c = u"Couleur = %s - température = %s°C" % ('rouge', 14.783)
>>> print c, type(c)
Couleur = rouge - température = 14.783°C <type 'unicode'>
```

Exercices :

10.21 Écrivez un script qui recopie en *Utf-8* un fichier texte encodé à l'origine en *Latin-1*, en veillant en outre à ce que chaque mot commence par une majuscule.

10.22 Écrivez un script qui compte le nombre de mots contenus dans un fichier texte.

10.23 Écrivez un script qui compte dans un fichier texte quelconque le nombre de lignes contenant des caractères numériques.

10.24 Écrivez un script qui recopie un fichier texte en fusionnant (avec la précédente) les lignes qui ne commencent pas par une majuscule.

10.25 Vous disposez d'un fichier contenant des valeurs numériques. Considérez que ces valeurs sont les diamètres d'une série de sphères. Écrivez un script qui utilise les données de ce fichier pour en créer un autre, organisé en lignes de texte qui exprimeront « en clair » les autres caractéristiques de ces sphères (surface de section, surface extérieure et volume), dans des phrases telles que :

```
Diam.  46.20 cm Section  1676.39 cm² Surf.  6705.54 cm² Vol.  51632.67 cm³
Diam. 120.00 cm Section 11309.73 cm² Surf. 45238.93 cm² Vol. 904778.68 cm³
Diam.   0.03 cm Section     0.00 cm² Surf.     0.00 cm² Vol.      0.00 cm³
Diam.  13.90 cm Section   151.75 cm² Surf.   606.99 cm² Vol.   1406.19 cm³
Diam.  88.80 cm Section  6193.21 cm² Surf. 24772.84 cm² Vol. 366638.04 cm³
etc.
```

10.26 Vous avez à votre disposition un fichier texte dont les lignes représentent des valeurs numériques de type réel, sans exposant (et encodées sous forme de chaînes de caractères).
Écrivez un script qui recopie ces valeurs dans un autre fichier en les arrondissant de telle sorte que leur partie décimale ne comporte plus qu'un seul chiffre après la virgule, celui-ci ne pouvant être que 0 ou 5 (l'arrondi doit être correct).

Le point sur les listes

Nous avons déjà rencontré les listes à plusieurs reprises, depuis leur présentation sommaire au chapitre 5. Les listes sont des collections ordonnées d'objets. Comme les chaînes de caractères, les listes font partie d'un type général que l'on appelle *séquences* sous Python. Comme les caractères dans une chaîne, les objets placés dans une liste sont rendus accessibles par l'intermédiaire d'un *index* (un nombre qui indique l'emplacement de l'objet dans la séquence).

Définition d'une liste – Accès à ses éléments

Vous savez déjà que l'on délimite une liste à l'aide de crochets :

```
>>> nombres = [5, 38, 10, 25]
>>> mots = ["jambon", "fromage", "confiture", "chocolat"]
>>> stuff = [5000, "Brigitte", 3.1416, ["Albert", "René", 1947]]
```

Dans le dernier exemple ci-dessus, nous avons rassemblé un entier, une chaîne, un réel et même une liste, pour vous rappeler que l'on peut combiner dans une liste des données de n'importe quel type, y compris des listes, des dictionnaires et des tuples (ceux-ci seront étudiés plus loin).

Pour accéder aux éléments d'une liste, on utilise les mêmes méthodes (index, découpage en tranches) que pour accéder aux caractères d'une chaîne :

```
>>> print nombres[2]
10
>>> print nombres[1:3]
[38, 10]
>>> print nombres[2:3]
[10]
>>> print nombres[2:]
[10, 25]
>>> print nombres[:2]
[5, 38]
>>> print nombres[-1]
25
>>> print nombres[-2]
10
```

Les exemples ci-dessus devraient attirer votre attention sur le fait qu'une *tranche* découpée dans une liste est toujours elle-même une liste (même s'il s'agit d'une tranche qui ne contient qu'un seul élément, comme dans notre troisième exemple), alors qu'un élément isolé peut contenir n'importe quel type de donnée. Nous allons approfondir cette distinction tout au long des exemples suivants.

Les listes sont modifiables

Contrairement aux chaînes de caractères, les listes sont des *séquences modifiables*. Cela nous permettra de construire plus tard des listes de grande taille, morceau par morceau, d'une manière dynamique (c'est-à-dire à l'aide d'un algorithme quelconque). Exemples :

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
```

Dans l'exemple ci-dessus, on a remplacé le premier élément de la liste **nombres**, en utilisant l'opérateur **[ ]** (opérateur d'indiçage) *à la gauche* du signe égale.

Si l'on souhaite accéder à un élément faisant partie d'une liste, elle-même située dans une autre liste, il suffit d'indiquer les deux index *entre crochets successifs* :

```
>>> stuff[3][1] = "Isabelle"
>>> stuff
[5000, 'Brigitte', 3.1415999999999999, ['Albert', 'Isabelle', 1947]]
```

Comme c'est le cas pour toutes les séquences, il ne faut jamais oublier que la numérotation des éléments commence à partir de zéro. Ainsi, dans l'exemple ci-dessus on remplace l'élément n° 1 d'une liste, qui est elle-même l'élément n° 3 d'une autre liste : la liste **stuff**.

Les listes sont des objets

Sous Python, les listes sont des objets à part entière, et vous pouvez donc leur appliquer un certain nombre de *méthodes* particulièrement efficaces. En voici quelques-unes :

```
>>> nombres = [17, 38, 10, 25, 72]
>>> nombres.sort()                      # trier la liste
>>> nombres
[10, 17, 25, 38, 72]

>>> nombres.append(12)                  # ajouter un élément à la fin
>>> nombres
[10, 17, 25, 38, 72, 12]

>>> nombres.reverse()                   # inverser l'ordre des éléments
>>> nombres
[12, 72, 38, 25, 17, 10]

>>> nombres.index(17)                   # retrouver l'index d'un élément
4

>>> nombres.remove(38)                  # enlever (effacer) un élément
>>> nombres
[12, 72, 25, 17, 10]
```

En plus de ces méthodes, vous disposez encore de l'instruction intégrée **del** , qui vous permet d'effacer un ou plusieurs éléments à partir de leur(s) index :

```
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

Notez bien la différence entre la méthode **remove()** et l'instruction **del** : **del** travaille avec *un index* ou *une tranche d'index*, tandis que **remove()** recherche *une valeur* (si plusieurs éléments de la liste possèdent la même valeur, seul le premier est effacé).

Exercices :

10.27 Écrivez un script qui génère la liste des carrés et des cubes des nombres de 20 à 40.

10.28 Écrivez un script qui crée automatiquement la liste des *sinus* des angles de 0° à 90° , par pas de 5°. Attention : la fonction **sin()** du module **math** considère que les angles sont fournis en *radians* (360° = 2 π radians)

10.29 Écrivez un script qui permette d'obtenir à l'écran les 15 premiers termes des tables de multiplication par 2, 3, 5, 7, 11, 13, 17, 19 (ces nombres seront placés au départ dans une liste) sous la forme d'une table similaire à la suivante :

```
 2   4   6   8  10  12  14  16  18  20  22  24  26  28  30
 3   6   9  12  15  18  21  24  27  30  33  36  39  42  45
```

```
  5   10   15   20   25   30   35   40   45   50   55   60   65   70   75
```
etc.

10.30 Soit la liste suivante :
['Jean-Michel', 'Marc', 'Vanessa', 'Anne', 'Maximilien', 'Alexandre-Benoît', 'Louise']
Écrivez un script qui affiche chacun de ces noms avec le nombre de caractères correspondant. *Faites attention aux caractères accentués !*

10.31 Vous disposez d'une liste de nombres entiers quelconques, certains d'entre eux étant présents en plusieurs exemplaires. Écrivez un script qui recopie cette liste dans une autre, *en omettant les doublons*. La liste finale devra être *triée*.

10.32 Écrivez un script qui recherche le mot le plus long dans une phrase donnée (l'utilisateur du programme doit pouvoir entrer une phrase de son choix). Tâchez de tenir compte des caractères accentués.

10.33 Écrivez un script capable d'afficher la liste de tous les jours d'une année imaginaire, laquelle commencerait un Jeudi. Votre script utilisera lui-même trois listes : une liste des noms de jours de la semaine, une liste des noms des mois, et une liste des nombres de jours que comportent chacun des mois (ne pas tenir compte des années bissextiles).
Exemple de sortie :
```
Jeudi 1 Janvier   Vendredi 2 Janvier   Samedi 3 Janvier   Dimanche 4 Janvier
```
... et ainsi de suite, jusqu'au Jeudi 31 Décembre.

10.34 Vous avez à votre disposition un fichier texte qui contient un certain nombre de noms d'élèves. Écrivez un script qui effectue une copie *triée* de ce fichier.

10.35 Écrivez une fonction permettant de trier une liste. Cette fonction ne pourra pas utiliser la méthode intégrée **sort()** de Python : Vous devez donc *définir vous-même l'algorithme de tri.*

> *Note : cette dernière question devrait faire l'objet d'une discussion-synthèse en classe.*

Techniques de « slicing » avancé pour modifier une liste

Comme nous venons de le signaler, vous pouvez ajouter ou supprimer des éléments dans une liste en utilisant une instruction (**del**) et une méthode (**append()**) intégrées. Si vous avez bien assimilé le principe du « découpage en tranches » (*slicing*), vous pouvez cependant obtenir les mêmes résultats à l'aide du seul opérateur **[ ]**. L'utilisation de cet opérateur est un peu plus délicate que celle d'instructions ou de méthodes dédiées, mais elle permet davantage de souplesse :

Insertion d'un ou plusieurs éléments n'importe où dans une liste

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] =["miel"]
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']

>>> mots[5:5] =['saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson', 'ketchup']
```

Pour utiliser cette technique, vous devez prendre en compte les particularités suivantes :

- Si vous utilisez l'opérateur **[ ]** *à la gauche du signe égale* pour effectuer une insertion ou une suppression d'élément(s) dans une liste, vous devez obligatoirement y indiquer une

« tranche » dans la liste cible (c'est-à-dire deux index réunis par le symbole **:**), et non un élément isolé dans cette liste.

- L'élément que vous fournissez à la droite du signe égale *doit lui-même être une liste*. Si vous n'insérez qu'un seul élément, il vous faut donc le présenter entre crochets pour le transformer d'abord en une liste d'un seul élément. Notez bien que l'élément **mots[1]** n'est pas une liste (c'est la chaîne 'fromage'), alors que l'élément **mots[1:3]** en est une.

Vous comprendrez mieux ces contraintes en analysant ce qui suit :

Suppression / remplacement d'éléments

```
>>> mots[2:5] = []                # [] désigne une liste vide
>>> mots
['jambon','fromage','saucisson', 'ketchup']

>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade', 'ketchup']

>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
```

- A la première ligne de cet exemple, nous remplaçons la tranche [2:5] par une liste vide, ce qui correspond à un effacement.
- A la quatrième ligne, nous remplaçons une tranche par un seul élément. (Notez encore une fois que cet élément doit lui-même être « présenté » comme une liste).
- A la 7e ligne, nous remplaçons une tranche de deux éléments par une autre qui en comporte 3.

Création d'une liste de nombres à l'aide de la fonction range()

Si vous devez manipuler des séquences de nombres, vous pouvez les créer très aisément à l'aide de cette fonction :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La fonction **range()** génère une liste de nombres entiers de valeurs croissantes. Si vous appelez **range()** avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, mais en commençant *à partir de zéro* (c'est-à-dire que **range(n)** génère les nombres de 0 à n-1). Notez bien que l'argument fourni n'est jamais dans la liste générée.

On peut aussi utiliser **range()** avec deux, ou même trois arguments séparés par des virgules, afin de générer des séquences de nombres plus spécifiques :

```
>>> range(5,13)
[5, 6, 7, 8, 9, 10, 11, 12]
>>> range(3,16,3)
[3, 6, 9, 12, 15]
```

Si vous avez du mal à assimiler l'exemple ci-dessus, considérez que range() attend toujours de un à trois arguments, que l'on pourrait intituler *FROM, TO & STEP*. *FROM* est la première valeur à générer, *TO* est la dernière (ou plutôt la dernière + un), et *STEP* le « pas » à sauter pour passer d'une valeur à la suivante. S'ils ne sont pas fournis, les paramètres *FROM* et *STEP* prennent leurs valeurs par défaut, qui sont respectivement 0 et 1.

Parcours d'une liste à l'aide de for, range() et len()

L'instruction **for** est l'instruction idéale pour parcourir une liste :

```
>>> prov = ['La','raison','du','plus','fort','est','toujours','la','meilleure']
>>> for mot in prov:
        print mot,
La raison du plus fort est toujours la meilleure
```

Il est très pratique de combiner les fonctions **range()** et **len()** pour obtenir automatiquement tous les indices d'une séquence (liste ou chaîne). Exemple :

```
fable = ['Maître','Corbeau','sur','un','arbre','perché']
for index in range(len(fable)):
    print index, fable[index]
```

L'exécution de ce script donne le résultat :

```
0 Maître
1 Corbeau
2 sur
3 un
4 arbre
5 perché
```

Une conséquence du typage dynamique

Comme nous l'avons déjà signalé plus haut (page 132), le type de la variable utilisée avec l'instruction **for** *est redéfini continuellement* au fur et à mesure du parcours : même si les éléments d'une liste sont de types différents, on peut parcourir cette liste à l'aide de **for** sans qu'il ne s'ensuive une erreur, car le type de la variable de parcours s'adapte automatiquement à celui de l'élément en cours de lecture. Exemple :

```
>>> divers = [3, 17.25, [5, 'Jean'], 'Linux is not Windoze']
>>> for item in divers:
        print item, type(item)

3 <type 'int'>
17.25 <type 'float'>
[5, 'Jean'] <type 'list'>
Linux is not Windoze <type 'str'>
```

Dans l'exemple ci-dessus, on utilise la fonction intégrée **type()** pour montrer que la variable **item** change effectivement de type à chaque itération (ceci est rendu possible grâce au typage dynamique des variables Python).

Opérations sur les listes

On peut appliquer aux listes les opérateurs + (concaténation) et * (multiplication) :

```
>>> fruits = ['orange','citron']
>>> legumes = ['poireau','oignon','tomate']
>>> fruits + legumes
['orange', 'citron', 'poireau', 'oignon', 'tomate']
>>> fruits * 3
['orange', 'citron', 'orange', 'citron', 'orange', 'citron']
```

L'opérateur * est particulièrement utile pour créer une liste de n éléments identiques :

```
>>> sept_zeros = [0]*7
>>> sept_zeros
[0, 0, 0, 0, 0, 0, 0]
```

Supposons par exemple que vous voulez créer une liste **B** qui contienne le même nombre d'éléments qu'une autre liste **A**. Vous pouvez obtenir ce résultat de différentes manières, mais l'une des plus simples consistera à effectuer : `B = [0]*len(A)`

Test d'appartenance

Vous pouvez aisément déterminer si un élément fait partie d'une liste à l'aide de l'instruction **in** :

```
>>> v = 'tomate'
>>> if v in legumes:
        print 'OK'
OK
```

Copie d'une liste

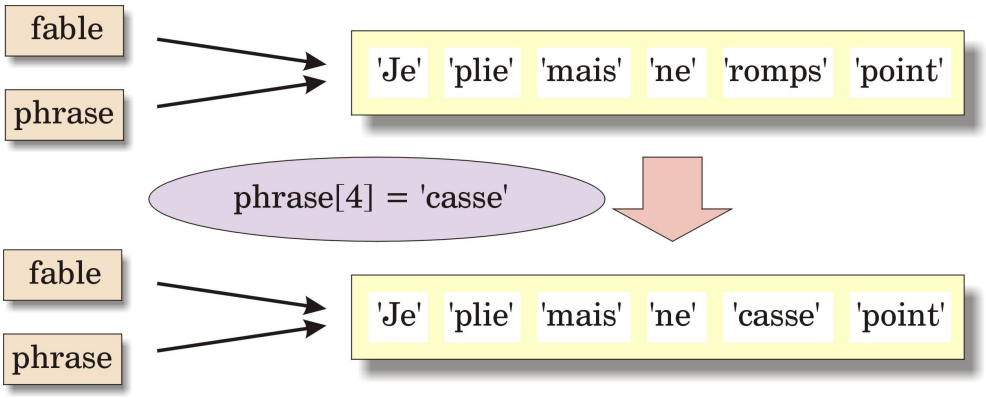
Considérons que vous disposez d'une liste **fable** que vous souhaitez recopier dans une nouvelle variable que vous appellerez **phrase**. La première idée qui vous viendra à l'esprit sera certainement d'écrire une simple affectation telle que :

```
>>> phrase = fable
```

En procédant ainsi, sachez que *vous ne créez pas une véritable copie*. A la suite de cette instruction, il n'existe toujours qu'une seule liste dans la mémoire de l'ordinateur. Ce que vous avez créé est seulement *une nouvelle référence* vers cette liste. Essayez par exemple :

```
>>> fable = ['Je','plie','mais','ne','romps','point']
>>> phrase = fable
>>> fable[4] ='casse'
>>> phrase
['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Si la variable **phrase** contenait une véritable copie de la liste, cette copie serait indépendante de l'original et ne devrait donc pas pouvoir être modifiée par une instruction telle que celle de la troisième ligne, qui s'applique à la variable **fable.** Vous pouvez encore expérimenter d'autres modifications, soit au contenu de **fable**, soit au contenu de **phrase.** Dans tous les cas, vous constaterez que les modifications de l'une sont répercutées dans l'autre, et vice-versa.



En fait, les noms **fable** et **phrase** désignent tous deux *un seul et même objet* en mémoire. Pour décrire cette situation, les informaticiens diront que le nom **phrase** est un *alias* du nom **fable.**

Nous verrons plus tard l'utilité des alias. Pour l'instant, nous voudrions disposer d'une technique pour effectuer une véritable copie d'une liste. Avec les notions vues précédemment, vous devriez pouvoir en trouver une par vous-même.

Petite remarque concernant la syntaxe :

Python vous autorise à « étendre » une longue instruction sur plusieurs lignes, si vous continuez à encoder quelque chose qui est délimité par une paire de parenthèses, de crochets ou d'accolades. Vous pouvez traiter ainsi des expressions parenthésées, ou encore la définition de longues listes, de grands tuples ou de grands dictionnaires (voir plus loin). Le niveau d'indentation n'a pas d'importance : l'interpréteur détecte la fin de l'instruction, là où la paire syntaxique est refermée.

Cette fonctionnalité vous permet d'améliorer la lisibilité de vos programmes. Exemple :

```
couleurs = ['noir', 'brun', 'rouge',
            'orange', 'jaune', 'vert',
            'bleu', 'violet', 'gris', 'blanc']
```

Exercices :

10.36 Soient les listes suivantes :

```
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
 'Juillet','Août','Septembre','Octobre','Novembre','Décembre']
```

Écrivez un petit programme qui insère dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier',31,'Février',28,'Mars',31, etc...]`.

10.37 Créez une liste **A** contenant quelques éléments. Effectuez une *vraie copie* de cette liste dans une nouvelle variable **B**. Suggestion : créez d'abord une liste **B** de même taille que **A** mais ne contenant que des zéros. Remplacez ensuite tous ces zéros par les éléments tirés de **A**.

10.38 Même question, mais autre suggestion : créez d'abord une liste **B** vide. Remplissez-la ensuite à l'aide des éléments de **A** ajoutés l'un après l'autre.

10.39 Même question, autre suggestion encore : pour créer la liste **B**, découpez dans la liste **A** une tranche incluant tous les éléments (à l'aide de l'opérateur **[:]**).

10.40 *Un nombre premier est un nombre qui n'est divisible que par un et par lui-même.* Écrivez un programme qui établisse la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du *crible d'Eratosthène* :
- Créez une liste de 1000 éléments, chacun initialisé à la valeur 1.
- Parcourez cette liste à partir de l'élément d'indice 2 :
si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivé.

Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.
En effet : A partir de l'indice 2, vous annulez tous les éléments d'indices pairs : 4, 6, 8, 10, etc.
Avec l'indice 3, vous annulez les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite.
Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

Nombres aléatoires - Histogrammes

La plupart des programmes d'ordinateur font exactement la même chose chaque fois qu'on les exécute. De tels programmes sont dits *déterministes*. Le déterminisme est certainement une bonne chose : nous voulons évidemment qu'une même série de calculs appliquée aux mêmes données initiales aboutisse toujours au même résultat. Pour certaines applications, cependant, nous pou-

vons souhaiter que l'ordinateur soit imprévisible. Le cas des jeux constitue un exemple évident, mais il en existe bien d'autres.

Contrairement aux apparences, il n'est pas facile du tout d'écrire un algorithme qui soit réellement non-déterministe (c'est-à-dire qui produise un résultat totalement *imprévisible*). Il existe cependant des techniques mathématiques permettant de simuler plus ou moins bien l'effet du hasard. Des livres entiers ont été écrits sur les moyens de produire ainsi un hasard « de bonne qualité ». Nous n'allons évidemment pas développer ici une telle question, mais rien ne vous empêche de consulter à ce sujet votre professeur de mathématiques.

Dans son module **random**, Python propose toute une série de fonctions permettant de générer des nombres aléatoires qui suivent différentes distributions mathématiques. Nous n'examinerons ici que quelques-unes d'entre elles. Veuillez consulter la documentation en ligne pour découvrir les autres. Vous pouvez importer toutes les fonctions du module par :

```
>>> from random import *
```

La fonction ci-dessous permet de créer une liste de nombres réels aléatoires, de valeur comprise entre zéro et un. L'argument à fournir est la taille de la liste :

```
>>> def list_aleat(n):
        s = [0]*n
        for i in range(n):
            s[i] = random()
        return s
```

Vous pouvez constater que nous avons pris le parti de construire d'abord une liste de zéros de taille n, et ensuite de remplacer les zéros par des nombres aléatoires.

Exercices :

10.41 Réécrivez la fonction **list_aleat()** ci-dessus, en utilisant la méthode **append()** pour construire la liste petit à petit à partir d'une liste vide (au lieu de remplacer les zéros d'une liste préexistante comme nous l'avons fait).

10.42 Ecrivez une fonction **imprime_liste()** qui permette d'afficher ligne par ligne tous les éléments contenus dans une liste de taille quelconque. Le nom de la liste sera fourni en argument. Utilisez cette fonction pour imprimer la liste de nombres aléatoires générés par la fonction **list_aleat()**. Ainsi par exemple, l'instruction `imprime_liste(liste_aleat(8))` devra afficher une colonne de 8 nombres réels aléatoires.

Les nombres ainsi générés sont-ils vraiment aléatoires ? C'est difficile à dire. Si nous ne tirons qu'un petit nombre de valeurs, nous ne pouvons rien vérifier. Par contre, si nous utilisons un grand nombre de fois la fonction **random()**, nous nous attendons à ce que la moitié des valeurs produites soient plus grandes que 0,5 (et l'autre moitié plus petites).

Affinons ce raisonnement. Les valeurs tirées sont toujours dans l'intervalle 0-1. Partageons cet intervalle en 4 fractions égales : de 0 à 0,25 , de 0,25 à 0,5 , de 0,5 à 0,75 , et de 0,75 à 1.
Si nous tirons un grand nombre de valeurs au hasard, nous nous attendons à ce qu'il y en ait autant qui se situent dans chacune de nos 4 fractions. Et nous pouvons généraliser ce raisonnement à un nombre quelconque de fractions, du moment qu'elles soient égales.

Exercice :

10.43 Vous allez écrire un programme destiné à vérifier le fonctionnement du générateur de nombres aléatoires de Python en appliquant la théorie exposée ci-dessus. Votre programme devra donc :

- Demander à l'utilisateur le nombre de valeurs à tirer au hasard à l'aide de la fonction **random()**. Il serait intéressant que le programme propose un nombre par défaut (1000 par exemple).
- Demander à l'utilisateur en combien de fractions il souhaite partager l'intervalle des valeurs possibles (c'est-à-dire l'intervalle de 0 à 1). Ici aussi, il faudrait proposer un nombre par défaut (5 fractions, par exemple). Vous pouvez également limiter le choix de l'utilisateur à un nombre compris entre 2 et le 1/10e du nombre de valeurs tirées au hasard.
- Construire une liste de N compteurs (N étant le nombre de fractions souhaitées). Chacun d'eux sera évidemment initialisé à zéro.
- Tirer au hasard toutes les valeurs demandées, à l'aide de la fonction **random()** , et mémoriser ces valeurs dans une liste.
- Mettre en œuvre un parcours de la liste des valeurs tirées au hasard (boucle), et effectuer un test sur chacune d'elles pour déterminer dans quelle fraction de l'intervalle 0-1 elle se situe. Incrémenter de une unité le compteur correspondant.
- Lorsque c'est terminé, afficher l'état de chacun des compteurs.

Exemple de résultats affichés par un programme de ce type :

```
Nombre de valeurs à tirer au hasard (défaut = 1000) : 100
Nombre de fractions dans l'intervalle 0-1 (entre 2 et 10, défaut =5) : 5
Tirage au sort des 100 valeurs ...
Comptage des valeurs dans chacune des 5 fractions ...
11 30 25 14 20
Nombre de valeurs à tirer au hasard (défaut = 1000) : 10000
Nombre de fractions dans l'intervalle 0-1 (entre 2 et 1000, défaut =5) : 5
Tirage au sort des 10000 valeurs ...
Comptage des valeurs dans chacune des 5 fractions ...
1970 1972 2061 1935 2062
```

Une bonne approche de ce genre de problème consiste à essayer d'imaginer quelles fonctions simples vous pourriez écrire pour résoudre l'une ou l'autre partie du problème, puis de les utiliser dans un ensemble plus vaste.

Par exemple, vous pourriez chercher à définir d'abord une fonction **numeroFraction()** qui servirait à déterminer dans quelle fraction de l'intervalle 0-1 une valeur tirée se situe. Cette fonction attendrait deux arguments (la valeur tirée, le nombre de fractions choisi par l'utilisateur) et fournirait en retour l'index du compteur à incrémenter (c'est-à-dire le n° de la fraction correspondante). Il existe peut-être un raisonnement mathématique simple qui permette de déterminer l'index de la fraction à partir de ces deux arguments. Pensez notamment à la fonction intégrée **int()** , qui permet de convertir un nombre réel en nombre entier en éliminant sa partie décimale.

Si vous ne trouvez pas, une autre réflexion intéressante serait peut-être de construire d'abord une liste contenant les valeurs « pivots » qui délimitent les fractions retenues (par exemple 0 – 0,25 – 0,5 – 0,75 - 1 dans le cas de 4 fractions). La connaissance de ces valeurs faciliterait peut-être l'écriture de la fonction **numeroFraction()** que nous souhaitons mettre au point.

Si vous disposez d'un temps suffisant, vous pouvez aussi réaliser une version graphique de ce programme, qui présentera les résultats sous la forme d'un histogramme (diagramme « en bâtons »).

Tirage au hasard de nombres entiers

Lorsque vous développerez des projets personnels, il vous arrivera fréquemment de souhaiter pouvoir disposer d'une fonction qui permette de tirer au hasard un nombre entier entre certaines limites. Par exemple, si vous voulez écrire un programme de jeu dans lequel des cartes à jouer

sont tirées au hasard (à partir d'un jeu ordinaire de 52 cartes), vous aurez certainement l'utilité d'une fonction capable de tirer au hasard un nombre entier compris entre 1 et 52.

Vous pouvez pour ce faire utiliser la fonction **randrange()** du module **random**.

Cette fonction peut être utilisée avec 1, 2 ou 3 arguments.

Avec un seul argument, elle renvoie un entier compris entre zéro et la valeur de l'argument diminué d'une unité. Par exemple, **randrange(6)** renvoie un nombre compris entre 0 et 5.

Avec deux arguments, le nombre renvoyé est compris entre la valeur du premier argument et la valeur du second argument diminué d'une unité. Par exemple, **randrange(2, 8)** renvoie un nombre compris entre 2 et 7.

Si l'on ajoute un troisième argument, celui-ci indique que le nombre tiré au hasard doit faire partie d'une série limitée d'entiers, séparés les uns des autres par un certain intervalle, défini lui-même par ce troisième argument. Par exemple, **randrange(3, 13, 3)** renverra un des nombres de la série 3, 6, 9, 12 :

```
>>> for i in range(15):
        print random.randrange(3,13,3),

3 12 6 9 6 6 12 6 3 6 9 3 6 12 12
```

Exercices :

10.44 Écrivez un script qui tire au hasard des cartes à jouer. Le nom de la carte tirée doit être correctement présenté, « en clair ». Le programme affichera par exemple :

```
Frappez <Enter> pour tirer une carte :
Dix de Trèfle
Frappez <Enter> pour tirer une carte :
As de Carreau
Frappez <Enter> pour tirer une carte :
Huit de Pique
Frappez <Enter> pour tirer une carte :
```

etc ...

Les tuples

Nous avons étudié jusqu'ici deux types de données composites : les *chaînes*, qui sont composées de caractères, et les *listes*, qui sont composées d'éléments de n'importe quel type. Vous devez vous rappeler une autre différence importante entre chaînes et listes : il n'est pas possible de changer les caractères au sein d'une chaîne existante, alors que vous pouvez modifier les éléments d'une liste. En d'autres termes, les listes sont des séquences modifiables, alors que les chaînes sont des séquences non-modifiables. Exemple :

```
>>> liste =['jambon','fromage','miel','confiture','chocolat']
>>> liste[1:3] =['salade']
>>> print liste
['jambon', 'salade', 'confiture', 'chocolat']

>>> chaine ='Roméo préfère Juliette'
>>> chaine[14:] ='Brigitte'

        ***** ==> Erreur: object doesn't support slice assignment  *****
```

Nous essayons de modifier la fin de la chaîne, mais cela ne marche pas. La seule possibilité d'arriver à nos fins est de créer une nouvelle chaîne et d'y recopier ce que nous voulons changer :

```
>>> chaine = chaine[:14] +'Brigitte'
>>> print chaine
Roméo préfère Brigitte
```

Python propose un type de données appelé ***tuple***[49], qui est assez semblable à une liste mais qui n'est pas modifiable.
Du point de vue de la syntaxe, un tuple est une collection d'éléments séparés par des virgules :

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
>>> print tuple
('a', 'b', 'c', 'd', 'e')
```

Bien que cela ne soit pas nécessaire, il est vivement conseillé de mettre le tuple en évidence en l'enfermant dans une paire de parenthèses, comme l'instruction **print** de Python le fait elle-même.
(Il s'agit simplement d'améliorer la lisibilité du code, mais vous savez que c'est important) :

```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

Les opérations que l'on peut effectuer sur des tuples sont syntaxiquement similaires à celles que l'on effectue sur les listes, si ce n'est que les tuples ne sont pas modifiables :

```
>>> print tuple[2:4]
('c', 'd')
>>> tuple[1:3] = ('x', 'y')          ==> ***** erreur ! *****

>>> tuple = ('André',) + tuple[1:]
>>> print tuple
('André', 'b', 'c', 'd', 'e')
```

Remarquez qu'il faut toujours au moins une virgule pour définir un tuple (le dernier exemple ci-dessus utilise un tuple contenant un seul élément : 'André').

Vous comprendrez l'utilité des tuples petit à petit. Signalons simplement ici qu'ils sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les tuples sont moins « gourmands » en ressources système (ils occupent moins de place en mémoire).

Les dictionnaires

Les types de données composites que nous avons abordés jusqu'à présent (*chaînes*, *listes* et *tuples*) étaient tous des *séquences*, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.

Les ***dictionnaires*** que nous découvrons ici constituent un autre *type composite*. Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables comme elles), mais ce ne sont pas des séquences. Les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une ***clé***, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions.

[49] ce terme n'est pas un mot anglais ordinaire : il s'agit d'un néologisme informatique

Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, et même aussi des fonctions, des *classes* ou des *instances* (voir plus loin)[50].

Création d'un dictionnaire

A titre d'exemple, nous allons créer un dictionnaire de langue, pour la traduction de termes informatiques anglais en français.

Puisque le type *dictionnaire* est un type modifiable, nous pouvons commencer par créer un dictionnaire vide, puis le remplir petit à petit. Du point de vue de la syntaxe, on reconnaît un dictionnaire au fait que ses éléments sont enfermés dans une paire d'accolades. Un dictionnaire vide sera donc noté **{}** :

```
>>> dico = {}
>>> dico['computer'] = 'ordinateur'
>>> dico['mouse'] ='souris'
>>> dico['keyboard'] ='clavier'

>>> print dico
{'computer': 'ordinateur', 'keyboard': 'clavier', 'mouse': 'souris'}
```

Comme vous pouvez l'observer dans la dernière ligne ci-dessus, un dictionnaire apparaît dans la syntaxe Python sous la forme d'une série d'éléments séparés par des virgules, le tout étant enfermé entre deux accolades. Chacun de ces éléments est lui-même constitué d'une paire d'objets : un index et une valeur, séparés par un double point.

Dans un dictionnaire, les index s'appellent des *clés*, et les éléments peuvent donc s'appeler des *paires clé-valeur.* Dans notre dictionnaire exemple, les clés et les valeurs sont des chaînes de caractères.

Veuillez à présent constater que l'ordre dans lequel les éléments apparaissent à la dernière ligne ne correspond pas à celui dans lequel nous les avons fournis. Cela n'a strictement aucune importance : nous n'essaierons jamais d'extraire une valeur d'un dictionnaire à l'aide d'un index numérique. Au lieu de cela, nous utiliserons les clés :

```
>>> print dico['mouse']
souris
```

Remarquez aussi que contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière (telle que **append()**) pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

Opérations sur les dictionnaires

Vous savez déjà comment ajouter des éléments à un dictionnaire. Pour en enlever, vous utiliserez l'instruction intégrée **del**. Créons pour l'exemple un autre dictionnaire, destiné cette fois à contenir l'inventaire d'un stock de fruits. Les index (ou clés) seront les noms des fruits, et les valeurs seront les masses de ces fruits répertoriées dans le stock (les valeurs sont donc cette fois des données de type numérique).

```
>>> invent = {'pommes': 430, 'bananes': 312, 'oranges' : 274, 'poires' : 137}
>>> print invent
{'oranges': 274, 'pommes': 430, 'bananes': 312, 'poires': 137}
```

Si le patron décide de liquider toutes les pommes et de ne plus en vendre, nous pouvons enlever cette entrée dans le dictionnaire :

[50] Les listes et les tuples peuvent eux aussi contenir des dictionnaires, des fonctions, des classes ou des instances, Nous n'avions pas mentionné tout cela jusqu'ici, afin de ne pas alourdir l'exposé,

```
>>> del invent['pommes']
>>> print invent
{'oranges': 274, 'bananes': 312, 'poires': 137}
```

La fonction intégrée **len()** est utilisable avec un dictionnaire : elle en renvoie le nombre d'éléments :

```
>>> print len(invent)
3
```

Les dictionnaires sont des objets

On peut appliquer aux dictionnaires un certain nombre de *méthodes* spécifiques :

La méthode **keys()** renvoie la liste des *clés* utilisées dans le dictionnaire :

```
>>> print dico.keys()
['computer', 'keyboard', 'mouse']
```

La méthode **values()** renvoie la liste des *valeurs* mémorisées dans le dictionnaire :

```
>>> print invent.values()
[274, 312, 137]
```

La méthode **has_key()** permet de savoir si un dictionnaire comprend une clé bien déterminée. On fournit la clé en argument, et la méthode renvoie une valeur 'vraie' ou 'fausse' (en fait, 1 ou 0), suivant que la clé est présente ou pas :

```
>>> print invent.has_key('bananes')
1

>>> if invent.has_key('pommes'):
       print 'nous avons des pommes'
else:
       print 'pas de pommes, sorry'

pas de pommes, sorry
```

La méthode **items()** extrait du dictionnaire une liste équivalente de tuples. Cette méthode se révélera très utile plus loin, lorsque nous voudrons parcourir un dictionnaire à l'aide d'une boucle :

```
>>> print invent.items()
[('oranges', 274), ('bananes', 312), ('poires', 137)]
```

La méthode **copy()** permet d'effectuer une *vraie copie* d'un dictionnaire. Il faut savoir en effet que la simple affectation d'un dictionnaire existant à une nouvelle variable crée seulement *une nouvelle référence* vers le même objet, et non un nouvel objet. Nous avons déjà discuté ce phénomène (*aliasing*) à propos des listes (voir page 145). Par exemple, l'instruction ci-dessous ne définit pas un nouveau dictionnaire (contrairement aux apparences) :

```
>>> stock = invent
>>> print stock
{'oranges': 274, 'bananes': 312, 'poires': 137}
```

Si nous modifions **invent**, alors **stock** aussi est modifié, et vice-versa (ces deux noms désignent en effet le même objet dictionnaire dans la mémoire de l'ordinateur) :

```
>>> del invent['bananes']
>>> print stock
{'oranges': 274, 'poires': 137}
```

Pour obtenir une vraie copie (indépendante) d'un dictionnaire préexistant, il faut employer la méthode **copy()** :

```
>>> magasin = stock.copy()
>>> magasin['prunes'] = 561
```

```
>>> print magasin
{'oranges': 274, 'prunes': 561, 'poires': 137}
>>> print stock
{'oranges': 274, 'poires': 137}
>>> print invent
{'oranges': 274, 'poires': 137}
```

Parcours d'un dictionnaire

Vous pouvez utiliser une boucle **for** pour traiter successivement tous les éléments contenus dans un dictionnaire, mais attention :

- Au cours de l'itération, ce sont les clés utilisées dans le dictionnaire qui seront successivement affectées à la variable de travail, et non les valeurs.
- L'ordre dans lequel les éléments seront extraits est imprévisible (puisqu'un dictionnaire n'est pas une séquence).

Exemple :

```
>>> invent ={"oranges":274, "poires":137, "bananes":312}
>>> for clef in invent:
...     print clef

poires
bananes
oranges
```

Si vous souhaitez effectuer un traitement sur les valeurs, il vous suffit alors de récupérer chacune d'elles à partir de la clé correspondante :

```
for clef in invent:
    print clef, invent[clef]

poires 137
bananes 312
oranges 274
```

Cette manière de procéder n'est cependant pas idéale, ni en termes de performances ni même du point de vue de la lisibilité. Il est recommandé de plutôt faire appel à la méthode **items()** décrite à la section précédente :

```
for clef, valeur in invent.items():
    print clef, valeur

poires 137
bananes 312
oranges 274
```

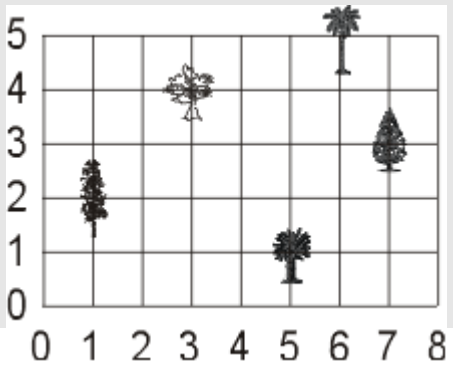
Dans cet exemple, la méthode **items()** appliquée au dictionnaire **invent** renvoie une liste de tuples (clef, valeur). Le parcours effectué sur cette liste à l'aide de la boucle **for** permet d'examiner chacun de ces tuples un par un.

Les clés ne sont pas nécessairement des chaînes de caractères

Jusqu'à présent nous avons décrit des dictionnaires dont les clés étaient à chaque fois des valeurs de type *string*. En fait nous pouvons utiliser en guise de clés n'importe quel type de donnée non modifiable : des entiers, des réels, des chaînes de caractères (*string* ou *unicode*), et même des tuples.

Considérons par exemple que nous voulions répertorier les arbres remarquables situés dans un grand terrain rectangulaire. Nous pouvons pour cela utiliser un dictionnaire, dont les clés seront des tuples indiquant les coordonnées **x,y** de chaque arbre :

```
>>> arb = {}
>>> arb[(1,2)] = 'Peuplier'
>>> arb[(3,4)] = 'Platane'
>>> arb[6,5] = 'Palmier'
>>> arb[5,1] = 'Cycas'
>>> arb[7,3] = 'Sapin'

>>> print arb
{(3, 4): 'Platane', (6, 5): 'Palmier', (5, 1):
 'Cycas', (1, 2): 'Peuplier', (7, 3): 'Sapin'}

>>> print arb[(6,5)]
palmier
```



Vous pouvez remarquer que nous avons allégé l'écriture à partir de la troisième ligne, en profitant du fait que les parenthèses délimitant les tuples sont facultatives (à utiliser avec prudence !).

Dans ce genre de construction, il faut garder à l'esprit que le dictionnaire contient des éléments seulement pour certains couples de coordonnées. Ailleurs, il n'y a rien. Par conséquent, si nous voulons interroger le dictionnaire pour savoir ce qui se trouve là où il n'y a rien, comme par exemple aux coordonnées (2,1), nous allons provoquer une erreur :

```
>>> print arb[1,2]
Peuplier
>>> print arb[2,1]

          ***** Erreur : KeyError: (2, 1)  *****
```

Pour résoudre ce petit problème, nous pouvons utiliser la méthode **get()** :

```
>>> print arb.get((1,2),'néant')
Peuplier
>>> print arb.get((2,1),'néant')
néant
```

Le premier argument transmis à cette méthode est la clé de recherche, le second argument est la valeur que nous voulons obtenir en retour si la clé n'existe pas dans le dictionnaire.

Les dictionnaires ne sont pas des séquences

Comme vous l'avez vu plus haut, les éléments d'un dictionnaire ne sont pas disposés dans un ordre particulier. Des opérations comme la concaténation et l'extraction (d'un groupe d'éléments contigus) ne peuvent donc tout simplement pas s'appliquer ici. Si vous essayez tout de même, Python lèvera une erreur lors de l'exécution du code :

```
>>> print arb[1:3]

         ***** Erreur : KeyError: slice(1, 3, None) *****
```

Vous avez vu également qu'il suffit d'affecter un nouvel indice (une nouvelle clé) pour ajouter une entrée au dictionnaire. Cela ne marcherait pas avec les listes[51] :

```
>>> invent['cerises'] = 987
>>> print invent
{'oranges': 274, 'cerises': 987, 'poires': 137}

>>> liste =['jambon', 'salade', 'confiture', 'chocolat']
>>> liste[4] ='salami'

         ***** IndexError: list assignment index out of range  *****
```

[51] Rappel : les méthodes permettant d'ajouter des éléments à une liste sont décrites page 142.

Du fait qu'ils ne sont pas des séquences, les dictionnaires se révèlent donc particulièrement précieux pour gérer des ensembles de données où l'on est amené à effectuer fréquemment des ajouts ou des suppressions, dans n'importe quel ordre. Ils remplacent avantageusement les listes lorsqu'il s'agit de traiter des ensembles de données numérotées, dont les numéros ne se suivent pas.

Exemple :

```
>>> client = {}
>>> client[4317] = "Dupond"
>>> client[256] = "Durand"
>>> client[782] = "Schmidt"
```

etc.

Exercices :

10.45 Écrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, et dans lequel vous mémoriserez les noms d'une série de copains, leur âge et leur taille. Votre script devra comporter deux fonctions : la première pour le remplissage du dictionnaire, et la seconde pour sa consultation. Dans la fonction de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur.
Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de type réel).
La fonction de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple « âge, taille » correspondant. Le résultat de la requête devra être une ligne de texte bien formatée, telle par exemple : « Nom : Jean Dhoute - âge : 15 ans - taille : 1.74 m ». Pour obtenir ce résultat, servez-vous du formatage des chaînes de caractères décrit à la page 138.

10.46 Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). (On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques).

Construction d'un histogramme à l'aide d'un dictionnaire

Les dictionnaires constituent un outil très élégant pour construire des *histogrammes*.

Supposons par exemple que nous voulions établir l'histogramme qui représente la fréquence d'utilisation de chacune des lettres de l'alphabet dans un texte donné. L'algorithme permettant de réaliser ce travail est extraordinairement simple si on le construit sur base d'un dictionnaire :

```
>>> texte ="les saucisses et saucissons secs sont dans le saloir"
>>> lettres ={}
>>> for c in texte:
        lettres[c] = lettres.get(c, 0) + 1

>>> print lettres
{'t': 2, 'u': 2, 'r': 1, 's': 14, 'n': 3, 'o': 3, 'l': 3, 'i': 3, 'd': 1, 'e':
5, 'c': 3, ' ': 8, 'a': 4}
```

Nous commençons par créer un dictionnaire vide : **lettres**. Ensuite, nous allons remplir ce dictionnaire en utilisant les caractères de l'alphabet en guise de clés. Les valeurs que nous mémoriserons pour chacune de ces clés seront les fréquences des caractères correspondants dans le texte. Afin de calculer celles-ci, nous effectuons un parcours de la chaîne de caractères **texte**. Pour chacun de ces caractères, nous interrogeons le dictionnaire à l'aide de la méthode **get()**, en utilisant le caractère en guise de clé, afin d'y lire la fréquence déjà mémorisée pour ce caractère. Si cette valeur n'existe pas encore, la méthode **get()** doit renvoyer une valeur nulle. Dans tous les cas,

nous incrémentons la valeur trouvée, et nous la mémorisons dans le dictionnaire, à l'emplacement qui correspond à la clé (c'est-à-dire au caractère en cours de traitement).

Pour fignoler notre travail, nous pouvons encore souhaiter afficher l'histogramme dans l'ordre alphabétique. Pour ce faire, nous pensons immédiatement à la méthode **sort()**, mais celle-ci ne peut s'appliquer qu'aux listes. Qu'à cela ne tienne ! Nous avons vu plus haut comment nous pouvions convertir un dictionnaire en une liste de tuples :

```
>>> lettres_triees = lettres.items()
>>> lettres_triees.sort()
>>> print lettres_triees
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3), ('n',
3), ('o', 3), ('r', 1), ('s', 14), ('t', 2), ('u', 2)]
```

Exercices :

10.47 Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte (on simplifiera le problème en ne tenant pas compte des lettres accentuées).

10.48 Modifiez le script ci-dessus afin qu'il établisse une table des occurrences de chaque *mot* dans le texte. Conseil : dans un texte quelconque, les mots ne sont pas seulement séparés par des espaces, mais également par divers signes de ponctuation. Pour simplifier le problème, vous pouvez commencer par remplacer tous les caractères non-alphabétiques par des espaces, et convertir la chaîne résultante en une liste de mots à l'aide de la méthode **split()**.

10.49 Vous avez à votre disposition un fichier texte quelconque (pas trop gros). Écrivez un script qui analyse ce texte, et mémorise dans un dictionnaire l'emplacement exact de chacun des mots (compté en nombre de caractères à partir du début). Lorsqu'un même mot apparaît plusieurs fois, tous ses emplacements doivent être mémorisés : chaque valeur de votre dictionnaire doit donc être une liste d'emplacements.

Contrôle du flux d'exécution à l'aide d'un dictionnaire

Il arrive fréquemment que l'on ait à diriger l'exécution d'un programme dans différentes directions, en fonction de la valeur prise par une variable. Vous pouvez bien évidemment traiter ce problème à l'aide d'une série d'instructions **if - elif - else** , mais cela peut devenir assez lourd et inélégant si vous avez affaire à un grand nombre de possibilités. Exemple :

```
materiau = raw_input("Choisissez le matériau : ")

if materiau == 'fer':
    fonctionA()
elif materiau == 'bois':
    fonctionC()
elif materiau == 'cuivre':
    fonctionB()
elif materiau == 'pierre':
    fonctionD()
elif   ... etc ...
```

Les langages de programmation proposent souvent des instructions spécifiques pour traiter ce genre de problème, telles les instructions *switch* ou *case* du *C* ou du *Pascal*. Python n'en propose aucune, mais vous pouvez vous tirer d'affaire dans bien des cas à l'aide d'une liste (nous en donnons un exemple détaillé à la page 233), ou mieux encore à l'aide d'un dictionnaire. Exemple :

```
materiau = raw_input("Choisissez le matériau : ")

dico = {'fer':fonctionA,
```

```
        'bois:fonctionC,
        'cuivre':fonctionB,
        'pierre':fonctionD,
           ... etc ...}

dico[materiau]()
```

Les deux instructions ci-dessus pourraient être condensées en une seule, mais nous les laissons séparées pour bien détailler le mécanisme :

- La première instruction définit un dictionnaire **dico** dans lequel les clés sont les différentes possibilités pour la variable **materiau**, et les valeurs, les noms des fonctions à invoquer en correspondance. Notez bien qu'il s'agit seulement des noms de ces fonctions, *qu'il ne faut surtout pas faire suivre de parenthèses dans ce cas* (Sinon Python exécuterait chacune de ces fonctions au moment de la création du dictionnaire).
- La seconde instruction invoque la fonction correspondant au choix opéré à l'aide de la variable **materiau**. Le nom de la fonction est extrait du dictionnaire à l'aide de la clé, puis associé à une paire de parenthèses. Python reconnaît alors un appel de fonction tout à fait classique, et l'exécute.

Vous pouvez encore améliorer la technique ci-dessus en remplaçant cette instruction par sa variante ci-dessous, qui fait appel à la méthode **get()** afin de prévoir le cas où la clé demandée n'existerait pas dans le dictionnaire (vous obtenez de cette façon l'équivalent d'une instruction **else** terminant une longue série de **elif**) :

```
dico.get(materiau, fonctAutre)()
```

(Lorsque la la valeur de la variable **materiau** ne correspond à aucune clé du dictionnaire, c'est la fonction **fonctAutre()** qui est invoquée).

Exercices :

10.50 Complétez l'exercice 10.45 (mini-système de base de données) en lui ajoutant deux fonctions : l'une pour enregistrer le dictionnaire résultant dans un fichier texte, et l'autre pour reconstituer ce dictionnaire à partir du fichier correspondant.
Chaque ligne de votre fichier texte correspondra à un élément du dictionnaire. Elle sera formatée de manière à bien séparer :
- la clé et la valeur (c'est-à-dire le nom de la personne, d'une part, et l'ensemble : « âge + taille », d'autre part.
- dans l'ensemble « âge + taille », ces deux données numériques.
Vous utiliserez donc deux caractères séparateurs différents, par exemple « @ » pour séparer la clé et la valeur, et « # » pour séparer les données constituant cette valeur :

```
Juliette@18#1.67
Jean-Pierre@17#1.78
Delphine@19#1.71
Anne-Marie@17#1.63
```

etc.

10.51 Améliorez encore le script de l'exercice précédent, en utilisant un dictionnaire pour diriger le flux d'exécution du programme au niveau du menu principal.
Votre programme affichera par exemple :

```
Choisissez :
```

```
(R)écupérer un dictionnaire préexistant sauvegardé dans un fichier
(A)jouter des données au dictionnaire courant
(C)onsulter le dictionnaire courant
(S)auvegarder le dictionnaire courant dans un fichier
(T)erminer :
```

Suivant le choix opéré par l'utilisateur, vous effectuerez alors l'appel de la fonction correspondante en la sélectionnant dans un dictionnaire de fonctions.

11

Classes, objets, attributs

Les chapitres précédents vous ont déjà mis en contact à plusieurs reprises avec la notion *d'objet*. Vous savez donc déjà qu'un objet est une entité que l'on construit par *instanciation* à partir d'une *classe* (c'est-à-dire en quelque sorte une « catégorie » ou un « type » d'objet). Par exemple, on peut trouver dans la bibliothèque *Tkinter*, une classe **Button()** à partir de laquelle on peut créer dans une fenêtre un nombre quelconque de boutons.

Nous allons à présent examiner comment vous pouvez vous-mêmes définir de nouvelles classes d'objets. Il s'agit là d'un sujet relativement ardu, mais vous l'aborderez de manière très progressive, en commençant par définir des classes d'objets très simples, que vous perfectionnerez ensuite. Attendez-vous cependant à rencontrer des objets de plus en plus complexes par après.

Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés de différentes parties, qui soient elles-mêmes des objets, ceux-ci étant faits à leur tour d'autres objets plus simples, etc.

Utilité des classes

Les classes sont les principaux outils de la programmation orientée objet (*Object Oriented Programming ou OOP*). Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.

Le premier bénéfice de cette approche de la programmation réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'*encapsulation* : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermés » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : l'*interface* de l'objet.

En particulier, l'utilisation de classes dans vos programmes va vous permettre - entre autres avantages - *d'éviter au maximum l'emploi de variables globales*. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, d'autant plus importants que les programmes sont volumineux, parce qu'il est toujours possible que de telles variables soient modifiées, ou même redéfinies, n'importe où dans le corps du programme (ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel).

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de *construire de nouveaux objets à partir d'objets préexistants*, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de *dérivation* et de *polymorphisme*.

- La *dérivation* est le mécanisme qui permet de construire une classe « enfant » au départ d'une classe « parente ». L'enfant ainsi obtenu hérite toutes les propriétés et toute la fonctionnalité de son ancêtre, auxquelles on peut ajouter ce que l'on veut.

- Le *polymorphisme* permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte.

Avant d'aller plus loin, signalons ici que la programmation orientée objet est optionnelle sous Python. Vous pouvez donc mener à bien de nombreux projets sans l'utiliser, avec des outils plus simples tels que les fonctions. Sachez cependant que si vous faites l'effort d'apprendre à programmer à l'aide de classes, vous maîtriserez un niveau d'abstraction plus élevé, ce qui vous permettra de traiter des problèmes de plus en plus complexes. En d'autres termes, vous deviendrez un programmeur beaucoup plus compétent. Pour vous en convaincre, rappelez-vous les progrès que vous avez déjà réalisés au long de ce cours :

- Au début de votre étude, vous avez d'abord utilisé de simples instructions. Vous avez en quelque sorte « programmé à la main » (c'est-à-dire pratiquement sans « outils »).
- Lorsque vous avez découvert les fonctions prédéfinies (cf. chapitre 6), vous avez appris qu'il existait ainsi de vastes collections d'outils spécialisés, réalisés par d'autres programmeurs.
- En apprenant à écrire vos propres fonctions (cf. chapitre 7 et suivants), vous êtes devenus capables de créer vous-même de nouveaux outils, ce qui vous a donné un surcroît de puissance considérable.
- Si vous vous initiez maintenant à la programmation par classes, vous allez apprendre à construire des machines productrices d'outils. C'est évidemment plus complexe que de fabriquer directement ces outils, mais cela vous ouvre des perspectives encore bien plus larges !

Une bonne compréhension des classes vous aidera notamment à bien maîtriser le domaine des interfaces graphiques (*Tkinter*, *wxPython*) et vous préparera efficacement à aborder d'autres langages modernes, tels que *C++* ou *Java*.

Définition d'une classe élémentaire

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction **class**.
Nous allons donc apprendre à utiliser cette instruction, en commençant par définir un type d'objet très rudimentaire, lequel sera simplement un nouveau type de donnée. Nous avons déjà utilisé différentes types de données jusqu'à présent, mais c'étaient à chaque fois des types intégrés dans le langage lui-même. Nous allons maintenant créer un nouveau type composite : le type Point.

Ce type correspondra au concept de *point* en Géométrie plane.
Dans un plan, un point est caractérisé par deux nombres (ses coordonnées suivant x et y). En notation mathématique, on représente donc un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point (25,17). Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type *float.* Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet. Pour y arriver, nous allons définir une *classe* **Point()** :

```
>>> class Point(object):
        "Définition d'un point mathématique"
```

Les définitions de classes peuvent être situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer). L'exemple ci-dessus est probablement le plus simple qui se puisse concevoir. Une seule ligne nous a suffi pour définir le nouveau type d'objet **Point()**.

Remarquons d'emblée que :

- L'instruction **class** est un nouvel exemple d'*instruction composée*. N'oubliez pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne. Dans notre exemple ultra-simplifié, cette ligne n'est rien d'autre qu'un simple commentaire. Comme nous l'avons vu précédemment pour les fonctions (cf. page 76), vous pouvez insérer une chaîne de caractères directement après l'instruction **class**, afin de mettre en place un commentaire qui sera automatiquement incorporé dans le dispositif de documentation interne de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit.
- Les parenthèses sont destinées à contenir la référence d'une classe préexistante. Cela est requis pour permettre le mécanisme d'*héritage*. Toute classe nouvelle que nous créons peut en effet hériter d'une *classe parente* un ensemble de caractéristiques, auxquelles elle ajoutera les siennes propres. Lorsque l'on désire créer une classe fondamentale (c'est-à-dire ne dérivant d'aucune autre, comme c'est le cas ici avec notre classe **Point()**, la référence à indiquer doit être par convention le nom spécial **object**, lequel désigne l'ancêtre de toutes les classes[52].
- Une convention très répandue veut que l'on donne aux classes *des noms qui commencent par une majuscule*. Dans la suite de ce texte, nous respecterons cette convention, ainsi qu'une autre qui demande que dans les textes explicatifs, on associe à chaque nom de classe une paire de parenthèses, comme nous le faisons déjà pour les noms de fonctions.

Nous venons donc de définir une *classe* **Point()**. Nous pouvons à présent nous en servir pour *créer des objets de cette classe*, que l'on appellera aussi des *instances* de cette classe. L'opération s'appelle pour cette raison une *instanciation*. Créons par exemple un nouvel objet **p9**[53] :

```
>>> p9 = Point()
```

Après cette instruction, la variable **p9** contient la référence d'un nouvel *objet* **Point()**.
Nous pouvons dire également que **p9** est une nouvelle *instance* de la classe **Point()**.

> *Attention : comme les fonctions, les classes auxquelles on fait appel dans une instruction doivent toujours être accompagnées de parenthèses (même si aucun argument n'est transmis). Nous verrons un peu plus loin que les classes peuvent effectivement être appelées avec des arguments.*

Voyons maintenant si nous pouvons faire quelque chose avec notre nouvel objet **p9** :

```
>>> print p9
<__main__.Point instance at 0x403e1a8c>
```

Le message renvoyé par Python indique, comme vous l'aurez certainement bien compris tout de suite, que **p9** est une instance de la classe **Point()**, laquelle est définie elle-même au niveau principal (*main*) du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale (Veuillez consulter votre cours d'informatique générale si vous souhaitez des explications complémentaires à ce sujet).

```
>>> print p9.__doc__
Définition d'un point mathématique
```

Comme nous l'avons expliqué pour les fonctions (cf. page 76), les chaînes de documentation de divers objets Python sont associées à l'attribut prédéfini **__doc__**. Il est donc toujours possible de retrouver la documentation associée à un objet Python quelconque, en invoquant cet attribut.

---

[52] Dans les premières versions de Python, il était permis de n'indiquer aucune référence (et même d'omettre les parenthèses) dans la définition d'une classe fondamentale. Cette pratique est encore autorisée aujourd'hui, mais il est vivement conseillé d'adopter la syntaxe actuelle qui fait référence à la classe ancêtre **object**.

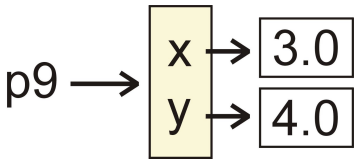
[53] Sous Python, on peut donc instancier un objet à l'aide d'une simple instruction d'affectation. D'autres langages imposent l'emploi d'une instruction spéciale, souvent appelée **new** pour bien montrer que l'on crée un nouvel objet à partir d'un moule. Exemple : **p9 = new Point()**

Attributs (ou variables) d'instance

L'objet que nous venons de créer est juste une coquille vide. Nous allons à présent lui ajouter des composants, par simple assignation, en utilisant le système de qualification des noms par points[54] :

```
>>> p9.x = 3.0
>>> p9.y = 4.0
```

Les variables **x** et **y** que nous avons ainsi définies en les liant d'emblée à **p9** , sont désormais des *attributs* de l'objet **p9.** On peut également les appeler des *variables d'instance*. Elles sont en effet incorporées, ou plutôt *encapsulées* dans cette instance (ou objet). Le diagramme d'état ci-contre montre le résultat de ces affectations : la variable **p9** contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs **x** et **y**.



On pourra utiliser les attributs d'un objet dans n'importe quelle expression, exactement comme toutes les variables ordinaires :

```
>>> print p9.x
3.0
>>> print p9.x**2 + p9.y**2
25.0
```

Du fait de leur *encapsulation* dans l'objet, les attributs sont des variables distinctes d'autres variables qui pourraient porter le même nom. Par exemple, l'instruction `x = p9.x` signifie : « extraire de l'objet référencé par **p9** la valeur de son attribut **x**, et assigner cette valeur à la variable **x** ». Il n'y a pas de conflit entre la variable indépendante **x** , et l'attribut **x** de l'objet **p9**. L'objet **p9** contient en effet son propre espace de noms, indépendant de l'espace de nom principal où se trouve la variable **x**.

> *Remarque importante :*
>
> *Nous venons de voir qu'il est très aisé d'ajouter un attribut à un objet en utilisant une simple instruction d'assignation telle que p9.x = 3.0 On peut se permettre cela sous Python (c'est une conséquence de l'assignation dynamique des variables), mais cela n'est pas vraiment recommandable, comme vous le comprendrez plus loin. Nous n'utiliserons donc cette façon de faire que de manière anecdotique, et uniquement dans le but de simplifier nos explications concernant les attributs d'instances.*
>
> *La bonne manière de procéder sera développée dans le chapitre suivant.*

Passage d'objets comme arguments lors de l'appel d'une fonction

Les fonctions peuvent utiliser des objets comme paramètres, et elles peuvent également fournir un objet comme valeur de retour. Par exemple, vous pouvez définir une fonction telle que celle-ci :

---

[54] Ce système de notation est similaire à celui que nous utilisons pour désigner les variables d'un module, comme par exemple **math.pi** ou **string.uppercase**. Nous aurons l'occasion d'y revenir plus tard, mais sachez dès à présent que les modules peuvent en effet contenir des fonctions, mais aussi des classes et des variables. Essayez par exemple :
>>> import string
>>> print string.uppercase
>>> print string.lowercase
>>> print string.hexdigits

```
>>> def affiche_point(p):
        print "coord. horizontale =", p.x, "coord. verticale =", p.y
```

Le paramètre **p** utilisé par cette fonction doit être un objet de type **Point()**, dont l'instruction qui suit utilisera les variables d'instance **p.x** et **p.y**. Lorsqu'on appelle cette fonction, il faut donc lui fournir un objet de type **Point()** comme argument. Essayons avec l'objet **p9** :

```
>>> affiche_point(p9)
coord. horizontale = 3.0 coord. verticale = 4.0
```

11 *Exercice*

11.1 Écrivez une fonction **distance()** qui permette de calculer la distance entre deux points. Cette fonction attendra évidemment deux objets **Point()** comme arguments.

Similitude et unicité

Dans la langue parlée, les mêmes mots peuvent avoir des significations fort différentes suivant le contexte dans lequel on les utilise. La conséquence en est que certaines expressions utilisant ces mots peuvent être comprises de plusieurs manières différentes (expressions ambiguës).

Le mot « même », par exemple, a des significations différentes dans les phrases : « Charles et moi avons la même voiture » et « Charles et moi avons la même mère ». Dans la première, ce que je veux dire est que la voiture de Charles et la mienne sont du même modèle. Il s'agit pourtant de deux voitures distinctes. Dans la seconde, j'indique que la mère de Charles et la mienne constituent en fait une seule et unique personne.

Lorsque nous traitons d'objets logiciels, nous pouvons rencontrer la même ambiguïté. Par exemple, si nous parlons de l'égalité de deux objets **Point()**, cela signifie-t-il que ces deux objets contiennent les mêmes données (leurs attributs), ou bien cela signifie-t-il que nous parlons de deux références à un même et unique objet ? Considérez par exemple les instructions suivantes :

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> print (p1 == p2)
0
```

Ces instructions créent deux objets **p1** et **p2** qui restent distincts, même s'ils font partie d'une même classe et ont des contenus similaires. La dernière instruction teste l'égalité de ces deux objets (double signe égale), et le résultat est zéro (ce qui signifie que l'expression entre parenthèses est fausse : il n'y a donc pas égalité).

On peut confirmer cela d'une autre manière encore :

```
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C50F9C>
```

L'information est claire : les deux variables **p1** et **p2** référencent bien des objets différents, mémorisés à des emplacements différents dans la mémoire de l'ordinateur.

Essayons autre chose, à présent :

```
>>> p2 = p1
>>> print (p1 == p2)
1
```

Par l'instruction `p2 = p1` , nous assignons le contenu de **p1** à **p2**. Cela signifie que désormais ces deux variables *référencent le même objet*. Les variables **p1** et **p2** sont des *alias*[55] l'une de l'autre.

Le test d'égalité dans l'instruction suivante renvoie cette fois la valeur 1, ce qui signifie que l'expression entre parenthèses est vraie : **p1** et **p2** désignent bien toutes deux un seul et unique objet, comme on peut s'en convaincre en essayant encore :

```
>>> p1.x = 7
>>> print p2.x
7
```

(lorsqu'on modifie l'attribut **x** de **p1**, on constate que l'attribut **x** de **p2** a changé lui aussi)

```
>>> print p1
<__main__.Point instance at 00C2CBEC>
>>> print p2
<__main__.Point instance at 00C2CBEC>
```

(les deux références **p1** et **p2** pointent vers le même emplacement dans la mémoire)

Objets composés d'objets

Supposons maintenant que nous voulions définir une classe qui servira à représenter des rectangles. Pour simplifier, nous allons considérer que ces rectangles seront toujours orientés horizontalement ou verticalement, et jamais en oblique.

De quelles informations avons-nous besoin pour définir de tels rectangles ?

Il existe plusieurs possibilités. Nous pourrions par exemple spécifier la position du centre du rectangle (deux coordonnées) et préciser sa taille (largeur et hauteur). Nous pourrions aussi spécifier les positions du coin supérieur gauche et du coin inférieur droit. Ou encore la position du coin supérieur gauche et la taille. Admettons ce soit cette dernière convention qui soit retenue.

Définissons donc notre nouvelle classe :

```
>>> class Rectangle(oject):
        "définition d'une classe de rectangles"
```

... et servons nous-en tout de suite pour créer une instance :

```
>>> boite = Rectangle()
>>> boite.largeur = 50.0
>>> boite.hauteur = 35.0
```

Nous créons ainsi un nouvel objet **Rectangle()** et lui donnons ensuite deux attributs. Pour spécifier le coin supérieur gauche, nous allons à présent utiliser une nouvelle instance de la classe **Point()** que nous avons définie précédemment. Ainsi nous allons créer un objet, à l'intérieur d'un autre objet !

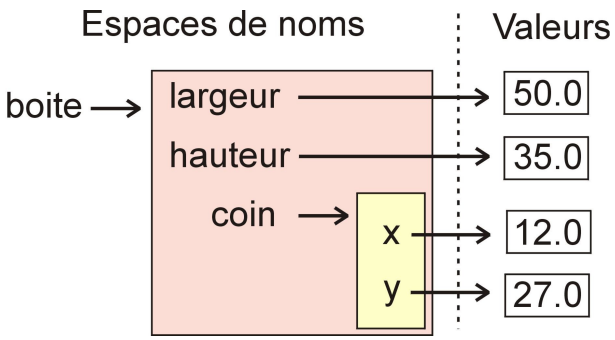
```
>>> boite.coin = Point()
>>> boite.coin.x = 12.0
>>> boite.coin.y = 27.0
```

À la première de ces trois instructions, nous créons un nouvel attribut **coin** pour l'objet **boite**. Ensuite, pour accéder à cet objet qui se trouve lui-même à l'intérieur d'un autre objet, nous utilisons la *qualification des noms hiérarchisée* (à l'aide de points) que nous avons déjà rencontrée à plusieurs reprises.

[55] Concernant ce phénomène d'aliasing, voir également page 145 : copie d'une liste

Ainsi l'expression `boite.coin.y` signifie « Aller à l'objet référencé dans la variable **boite**. Dans cet objet, repérer l'attribut **coin**, puis aller à l'objet référencé dans cet attribut. Une fois cet autre objet trouvé, sélectionner son attribut **y**. »

Vous pourrez peut-être mieux vous représenter tout cela à l'aide d'un diagramme tel que celui-ci :



Le nom **boite** se trouve dans *l'espace de noms principal*. Il référence un autre *espace de noms* réservé à l'objet correspondant, dans lequel sont mémorisés les noms **largeur**, **hauteur** et **coin**. Ceux-ci référencent à leur tour, soit *d'autres espaces de noms* (cas du nom « **coin** »), soit *des valeurs* bien déterminées, lesquelles sont mémorisées ailleurs.

Python réserve des espaces de noms différents pour chaque module, chaque classe, chaque instance, chaque fonction. Vous pouvez tirer parti de tous ces espaces de noms bien compartimentés afin de réaliser des *programmes robustes*, c'est-à-dire des programmes dont les différents composants ne peuvent pas facilement interférer.

Objets comme valeurs de retour d'une fonction

Nous avons vu plus haut que les fonctions peuvent utiliser des objets comme paramètres. Elles peuvent également transmettre une instance comme valeur de retour. Par exemple, la fonction **trouveCentre()** ci-dessous doit être appelée avec un argument de type **Rectangle()** et elle renvoie un objet de type **Point()**, lequel contiendra les coordonnées du centre du rectangle.

```
>>> def trouveCentre(box):
        p = Point()
        p.x = box.coin.x + box.largeur/2.0
        p.y = box.coin.y + box.hauteur/2.0
        return p
```

Vous pouvez par exemple appeler cette fonction, en utilisant comme argument l'objet **boite** défini plus haut :

```
>>> centre = trouveCentre(boite)
>>> print centre.x, centre.y
37.0  44.5
```

Modification des objets

Nous pouvons changer les propriétés d'un objet en assignant de nouvelles valeurs à ses attributs. Par exemple, nous pouvons modifier la taille d'un rectangle (sans modifier sa position), en réassignant ses attributs hauteur et largeur :

```
>>> boite.hauteur = boite.hauteur + 20
>>> boite.largeur = boite.largeur - 5
```

Nous pouvons faire cela sous Python, parce que dans ce langage les propriétés des objets sont toujours publiques (du moins jusqu'à la version actuelle 2.5). D'autres langages établissent une distinction nette entre attributs publics (accessibles de l'extérieur de l'objet) et attributs privés (qui sont accessibles seulement aux algorithmes inclus dans l'objet lui-même).

Cependant, comme nous l'avons déjà signalé plus haut (à propos de la définition des attributs par assignation simple, depuis l'extérieur de l'objet), modifier de cette façon les attributs d'une instance *n'est pas une pratique recommandable*, parce qu'elle contredit l'un des objectifs fondamentaux de la programmation orientée objet, qui vise à établir une séparation stricte entre la fonctionnalité d'un objet (telle qu'elle a été déclarée au monde extérieur) et la manière dont cette fonctionnalité est réellement implémentée dans l'objet (et que le monde extérieur n'a pas à connaître).

Concrètement, cela signifie que nous devons maintenant étudier comment faire fonctionner les objets à l'aide d'outils vraiment appropriés, que nous appellerons des *méthodes*.

Ensuite, lorsque nous aurons bien compris le maniement de celles-ci, nous nous fixerons pour règle de ne plus modifier les attributs d'un objet par assignation directe depuis le monde extérieur, comme nous l'avons fait jusqu'à présent. Nous veillerons au contraire à toujours utiliser pour cela des méthodes mises en place spécifiquement dans ce but, comme nous allons l'expliquer dans le chapitre suivant. L'ensemble de ces méthodes constituera ce que nous appellerons désormais l'*interface* de l'objet.

12

Classes, méthodes, héritage

Les classes que nous avons définies dans le chapitre précédent peuvent être considérées comme des espaces de noms particuliers, dans lesquels nous n'avons placé jusqu'ici que des variables (les attributs d'instance).

Il nous faut à présent doter ces classes d'une fonctionnalité. L'idée de base de la programmation orientée objet consiste en effet à regrouper dans un même ensemble (l'objet), à la fois un certain nombre de données (ce sont les *attributs d'instance*), et les algorithmes destinés à effectuer divers traitements sur ces données (ce sont les *méthodes*, à savoir des fonctions particulières encapsulées dans l'objet).

Objet = [attributs + méthodes]

Cette façon d'associer dans une même « capsule » les propriétés d'un objet et les fonctions qui permettent d'agir sur elles, correspond chez les concepteurs de programmes à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel qui nous entoure.

Considérons par exemple un *widget* « bouton » dans une application graphique. Il nous paraît raisonnable de souhaiter que l'objet informatique que nous appelons ainsi, ait un comportement qui ressemble à celui d'un bouton d'appareil quelconque dans le monde réel. Or nous savons que la fonctionnalité d'un bouton réel (sa capacité de fermer ou d'ouvrir un circuit électrique) est bien intégrée dans l'objet lui-même (au même titre que d'autres propriétés, telles que sa taille, sa couleur, etc.). De la même manière, nous souhaiterons donc que les différentes caractéristiques de notre bouton logiciel (sa taille, son emplacement, sa couleur, le texte qu'il supporte), mais aussi la définition de ce qui se passe lorsque l'on effectue différentes actions de la souris sur ce bouton, soient regroupés dans une entité bien précise à l'intérieur du programme, de telle sorte qu'il n'y ait pas de confusion entre ce bouton et un autre, ou *à fortiori* entre ce bouton et d'autres entités.

Définition d'une méthode

Pour illustrer notre propos, nous allons définir une nouvelle classe **Time()**, laquelle devrait nous permettre d'effectuer toute une série d'opérations sur des instants, des durées, etc. :

```
>>> class Time(object):
        "Définition d'une classe temporelle"
```

Créons à présent un objet de ce type, et ajoutons-lui des variables d'instance pour mémoriser les heures, minutes et secondes :

```
>>> instant = Time()
>>> instant.heure = 11
>>> instant.minute = 34
>>> instant.seconde = 25
```

A titre d'exercice, écrivez maintenant vous-même une fonction **affiche_heure()** , qui serve à visualiser le contenu d'un objet de classe **Time()** sous la forme conventionnelle « heure:minute:seconde ».
Appliquée à l'objet instant créé ci-dessus, cette fonction devrait donc afficher `11:34:25` :

```
>>> print affiche_heure(instant)
11:34:25
```

Votre fonction ressemblera probablement à ceci :

```
>>> def affiche_heure(t):
        print str(t.heure) + ":" + str(t.minute) + ":" + str(t.seconde)
```

... ou mieux encore, à :

```
>>> def affiche_heure(t):
        print "%s:%s:%s" % (t.heure, t.minute, t.seconde)
```

(en application de la technique de formatage des chaînes décrite à la page 138).

Si par la suite vous deviez utiliser fréquemment des objets de la classe **Time()**, cette fonction d'affichage vous serait probablement fort utile.

Il serait donc judicieux d'arriver à *encapsuler* cette fonction **affiche_heure()** dans la classe **Time()** elle-même, de manière à s'assurer qu'elle soit toujours automatiquement disponible, chaque fois que l'on aura à manipuler des objets de la classe **Time()**.

Une fonction ainsi encapsulée dans une classe s'appelle préférentiellement une *méthode*.

Vous avez évidemment déjà rencontré des méthodes à de nombreuses reprises dans les chapitres précédents de ce cours, et vous savez donc déjà qu'une méthode est bien une fonction associée à une classe particulière d'objets. Il vous reste seulement à apprendre comment construire une telle fonction.

Définition concrète d'une méthode dans un script

On définit une méthode comme on définit une fonction, c'est-à-dire en écrivant un bloc d'instructions à la suite du mot réservé **def**, mais cependant avec deux différences :

- La définition d'une méthode est toujours placée *à l'intérieur de la définition d'une classe*, de manière à ce que la relation qui lie la méthode à la classe soit clairement établie.
- La définition d'une méthode doit toujours comporter au moins un paramètre, lequel doit être *une référence d'instance*, et ce paramètre particulier doit toujours être listé en premier.

Vous pourriez en principe utiliser un nom de variable quelconque pour ce paramètre, mais il est vivement conseillé de respecter la convention qui consiste à toujours lui donner le nom : **self**.

Ce paramètre **self** est nécessaire, parce qu'il faut pouvoir désigner *l'instance à laquelle la méthode sera associée*, dans les instructions faisant partie de sa définition. (Vous comprendrez cela plus facilement avec les exemples ci-après).

> *Remarquons que la définition d'une méthode comporte toujours au moins un paramètre : self, alors que la définition d'une fonction peut n'en comporter aucun*

Voyons comment cela se passe en pratique :

Pour faire en sorte que la fonction **affiche_heure()** devienne une méthode de la classe **Time()**, il nous suffit de déplacer sa définition à l'intérieur de celle de la classe :

```
>>> class Time(object):
        "Nouvelle classe temporelle"
        def affiche_heure(t):
            print "%s:%s:%s" % (t.heure, t.minute, t.seconde)
```

Techniquement, c'est tout à fait suffisant, car le paramètre **t** peut parfaitement désigner l'instance à laquelle seront attachés les attributs **heure**, **minute** et **seconde**. Étant donné son rôle particulier, il est cependant fortement recommandé de changer son nom en **self** :

```
>>> class Time(object):
        "Nouvelle classe temporelle"
        def affiche_heure(self):
            print "%s:%s:%s" % (self.heure, self.minute, self.seconde)
```

La définition de la méthode **affiche_heure()** fait maintenant partie du bloc d'instructions indentées suivant l'instruction **class** (et dont fait partie aussi la chaîne documentaire "Nouvelle classe temporelle").

Essai de la méthode, dans une instance quelconque

Nous disposons donc dès à présent d'une classe **Time()**, dotée d'une méthode **affiche_heure()**. En principe, nous devons maintenant pouvoir créer des objets de cette classe, et leur appliquer cette méthode. Voyons si ça marche. Pour ce faire, commençons par instancier un objet :

```
>>> maintenant = Time()
```

Si nous essayons un peu trop vite de tester notre nouvelle méthode sur cet objet, ça ne marche pas :

```
>>> maintenant.affiche_heure()
AttributeError: 'Time' instance has no attribute 'heure'
```

C'est normal : nous n'avons pas encore créé les attributs d'instance. Il faudrait faire par exemple :

```
>>> maintenant.heure = 13
>>> maintenant.minute = 34
>>> maintenant.seconde = 21
```

... et réessayer. À présent, ça marche :

```
>>> maintenant.affiche_heure()
13:34:21
```

À plusieurs reprises, nous avons cependant déjà signalé qu'il n'est pas recommandable de créer ainsi des attributs d'instance par assignation directe en dehors de l'objet lui-même. Entre autres désagréments, cela conduirait fréquemment à des erreurs comme celle que nous venons de rencontrer. Voyons donc à présent comment nous pouvons mieux faire.

La méthode « constructeur »

L'erreur que nous avons rencontrée au paragraphe précédent est-elle évitable ?

Elle ne se produirait effectivement pas, si nous nous étions arrangés pour que la méthode **affiche_heure()** puisse toujours afficher quelque chose, sans qu'il ne soit nécessaire d'effectuer au préalable aucune manipulation sur l'objet nouvellement créé. En d'autres termes, il serait judicieux que *les variables d'instance soient prédéfinies* elles aussi à l'intérieur de la classe, avec pour chacune d'elles une valeur « par défaut ».

Pour obtenir cela, nous allons faire appel à une méthode particulière, que l'on désignera par la suite sous le nom de *constructeur*. Une méthode constructeur a ceci de particulier qu'*elle est exécu-*

tée automatiquement lorsque l'on instancie un nouvel objet à partir de la classe. On peut donc y placer tout ce qui semble nécessaire pour initialiser automatiquement l'objet que l'on crée.

Afin qu'elle soit reconnue comme telle par Python, la méthode constructeur devra obligatoirement s'appeler **__init__** (deux caractères « souligné », le mot **init**, puis encore deux caractères « souligné »).

Exemple :

```
>>> class Time(object):
        "Encore une nouvelle classe temporelle"

        def __init__(self):
            self.heure =0
            self.minute =0
            self.seconde =0

        def affiche_heure(self):
            print "%s:%s:%s" % (self.heure, self.minute, self.seconde)
```

Comme précédemment, créons un objet de cette classe et testons-en la méthode **affiche_heure()** :

```
>>> tstart = Time()
>>> tstart.affiche_heure()
0:0:0
```

Nous n'obtenons plus aucune erreur, cette fois. En effet : lors de son instanciation, l'objet **tstart** s'est vu attribuer automatiquement les trois attributs **heure**, **minute** et **seconde** par la méthode constructeur, avec zéro comme valeur par défaut pour chacun d'eux. Dès lors qu'un objet de cette classe existe, on peut donc tout de suite demander l'affichage de ces attributs.

L'intérêt de cette technique apparaîtra plus clairement si nous ajoutons encore quelque chose.

Comme toute méthode qui se respecte, la méthode **__init__()** peut être dotée de paramètres. Et dans le cas de cette méthode particulière qu'est le constructeur, les paramètres peuvent jouer un rôle très intéressant, parce qu'ils vont permettre d'initialiser certaines de ses variables d'instance au moment même de l'instanciation de l'objet.

Veuillez donc reprendre l'exemple précédent, en modifiant la définition de la méthode **__init__()** comme suit :

```
        def __init__(self, hh =0, mm =0, ss =0):
            self.heure = hh
            self.minute = mm
            self.seconde = ss
```

Notre nouvelle méthode **__init__()** comporte à présent 3 paramètres, avec pour chacun une valeur par défaut. Nous obtenons ainsi une classe encore plus perfectionnée. Lorsque nous instancions un objet de cette classe, nous pouvons maintenant initialiser ses principaux attributs à l'aide d'arguments, au sein même de l'instruction d'instanciation. Et si nous omettons tout ou partie d'entre eux, les attributs reçoivent de toute manière des valeurs par défaut.

Pour lui transmettre des arguments, lorsque l'on écrit l'instruction d'instanciation d'un nouvel objet, il suffit de placer ceux-ci dans les parenthèses qui accompagnent le nom de la classe. On procède donc exactement de la même manière que lorsqu'on invoque une fonction quelconque.

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet **Time()** :

```
>>> recreation = Time(10, 15, 18)
>>> recreation.affiche_heure()
10:15:18
```

Puisque les variables d'instance possèdent maintenant des valeurs par défaut, nous pouvons aussi bien créer de tels objets **Time()** en omettant un ou plusieurs arguments :

```
>>> rentree = Time(10, 30)
>>> rentree.affiche_heure()
10:30:0
```

ou encore :

```
>>> rendezVous = Time(hh =18)
>>> rendezVous.affiche_heure()
18:0:0
```

12 *Exercices :*

12.1 Définissez une classe **Domino()** qui permette d'instancier des objets simulant les pièces d'un jeu de dominos. Le constructeur de cette classe initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0).
Deux autres méthodes seront définies :
une méthode **affiche_points()** qui affiche les points présents sur les deux faces
une méthode **valeur()** qui renvoie la somme des points présents sur les 2 faces.

Exemples d'utilisation de cette classe :

```
>>> d1 = Domino(2,6)
>>> d2 = Domino(4,3)
>>> d1.affiche_points()
face A : 2  face B : 6
>>> d2.affiche_points()
face A : 4  face B : 3
>>> print "total des points :", d1.valeur() + d2.valeur()
15
>>> liste_dominos = []
>>> for i in range(7):
 liste_dominos.append(Domino(6, i))
>>> print liste_dominos
```

etc., etc.

12.2 Définissez une classe **CompteBancaire()**, qui permette d'instancier des objets tels que **compte1**, **compte2**, etc. Le constructeur de cette classe initialisera deux attributs d'instance **nom** et **solde**, avec les valeurs par défaut 'Dupont' et 1000.
Trois autres méthodes seront définies :
- **depot(somme)** permettra d'ajouter une certaine somme au solde
- **retrait(somme)** permettra de retirer une certaine somme du solde
- **affiche()** permettra d'afficher le nom du titulaire et le solde de son compte.

Exemples d'utilisation de cette classe :

```
>>> compte1 = CompteBancaire('Duchmol', 800)
>>> compte1.depot(350)
>>> compte1.retrait(200)
>>> compte1.affiche()
Le solde du compte bancaire de Duchmol est de 950 euros.
>>> compte2 = CompteBancaire()
>>> compte2.depot(25)
>>> compte2.affiche()
Le solde du compte bancaire de Dupont est de 1025 euros.
```

12.3 Définissez une classe **Voiture()** qui permette d'instancier des objets reproduisant le comportement de voitures automobiles. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :

```
marque = 'Ford',  couleur = 'rouge',  pilote = 'personne',  vitesse = 0.
```

Lorsque l'on instanciera un nouvel objet **Voiture()**, on pourra choisir sa marque et sa couleur, mais pas sa vitesse, ni le nom de son conducteur.
Les méthodes suivantes seront définies :
- **choix_conducteur(nom)** permettra de désigner (ou changer) le nom du conducteur
- **accelerer(taux, duree)** permettra de faire varier la vitesse de la voiture. La variation de vitesse obtenue sera égale au produit : **taux** x **duree**. Par exemple, si la voiture accélère au taux de 1,3 m/s2 pendant 20 secondes, son gain de vitesse doit être égal à 26 m/s. Des taux négatifs seront acceptés (ce qui permettra de décélérer). La variation de vitesse ne sera pas autorisée si le conducteur est 'personne'.
- **affiche_tout()** permettra de faire apparaître les propriétés présentes de la voiture, c'est-à-dire sa marque, sa couleur, le nom de son conducteur, sa vitesse.

Exemples d'utilisation de cette classe :

```
>>> a1 = Voiture('Peugeot', 'bleue')
>>> a2 = Voiture(couleur = 'verte')
>>> a3 = Voiture('Mercedes')
>>> a1.choix_conducteur('Roméo')
>>> a2.choix_conducteur('Juliette')
>>> a2.accelerer(1.8, 12)
>>> a3.accelerer(1.9, 11)
Cette voiture n'a pas de conducteur !
>>> a2.affiche_tout()
Ford verte pilotée par Juliette, vitesse = 21.6 m/s.
>>> a3.affiche_tout()
Mercedes rouge pilotée par personne, vitesse = 0 m/s.
```

12.4 Définissez une classe **Satellite()** qui permette d'instancier des objets simulant des satellites artificiels lancés dans l'espace, autour de la terre. Le constructeur de cette classe initialisera les attributs d'instance suivants, avec les valeurs par défaut indiquées :
```
masse = 100,  vitesse = 0.
```
Lorsque l'on instanciera un nouvel objet **Satellite()**, on pourra choisir son nom, sa masse et sa vitesse.
Les méthodes suivantes seront définies :
- **impulsion(force, duree)** permettra de faire varier la vitesse du satellite.
Pour savoir comment, rappelez-vous votre cours de physique : la variation de vitesse **Δv** subie par un objet de masse **m** soumis à l'action d'une force **F** pendant un temps **t** vaut $\Delta v = \frac{F \times t}{m}$. Par exemple : un satellite de 300 kg qui subit une force de 600 Newtons pendant 10 secondes voit sa vitesse augmenter (ou diminuer) de 20 m/s.
- **affiche_vitesse()** affichera le nom du satellite et sa vitesse courante.
- **energie()** renverra au programme appelant la valeur de l'énergie cinétique du satellite.

Rappel : l'énergie cinétique se calcule à l'aide de la formule $E_c = \frac{m \times v^2}{2}$

Exemples d'utilisation de cette classe :

```
>>> s1 = Satellite('Zoé', masse =250, vitesse =10)
```

```
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 40 m/s.
>>> print s1.energie()
200000
>>> s1.impulsion(500, 15)
>>> s1.affiche_vitesse()
vitesse du satellite Zoé = 70 m/s.
>>> print s1.energie()
612500
```

Espaces de noms des classes et instances

Vous avez appris précédemment (voir page 71) que les variables définies à l'intérieur d'une fonction sont des variables locales, inaccessibles aux instructions qui se trouvent à l'extérieur de la fonction. Cela vous permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence.

Pour décrire la même chose en d'autres termes, nous pouvons dire que chaque fonction possède son propre *espace de noms*, indépendant de l'espace de noms principal.

Vous avez appris également que les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais *en consultation seulement* : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier (à moins de faire appel à l'instruction **global**).

Il existe donc une sorte de hiérarchie entre les espaces de noms. Nous allons constater la même chose à propos des classes et des objets. En effet :

- Chaque classe possède son propre espace de noms. Les variables qui en font partie sont appelées *variables de classe* ou *attributs de classe*.
- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées *variables d'instance* ou *attributs d'instance*.
- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal.
- Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

Considérons par exemple la classe **Time()** définie précédemment. A la page 171, nous avons instancié trois objets de cette classe : **recreation**, **rentree** et **rendezVous**. Chacun a été initialisé avec des valeurs différentes, indépendantes. Nous pouvons modifier et réafficher ces valeurs à volonté dans chacun de ces trois objets, sans que l'autre n'en soit affecté :

```
>>> recreation.heure = 12
>>> rentree.affiche_heure()
10:30:0
>>> recreation.affiche_heure()
12:15:18
```

Veuillez à présent encoder et tester l'exemple ci-dessous :

```
>>> class Espaces(object):                  # 1
...     aa = 33                             # 2
...     def affiche(self):                  # 3
...         print aa, Espaces.aa, self.aa   # 4
...
>>> aa = 12                                 # 5
>>> essai = Espaces()                       # 6
>>> essai.aa = 67                           # 7
```

```
>>> essai.affiche()                          # 8
12 33 67
>>> print aa, Espaces.aa, essai.aa           # 9
12 33 67
```

Dans cet exemple, le même nom **aa** est utilisé pour définir trois variables différentes : une dans l'espace de noms de la classe (à la ligne 2), une autre dans l'espace de noms principal (à la ligne 5), et enfin une dernière dans l'espace de nom de l'instance (à la ligne 7).
La ligne 4 et la ligne 9 montrent comment vous pouvez accéder à ces trois espaces de noms (de l'intérieur d'une classe, ou au niveau principal), en utilisant la qualification par points. Notez encore une fois l'utilisation de **self** pour désigner l'instance à l'intérieur de la définition d'une classe.

Héritage

Les classes constituent le principal outil de la programmation orientée objet (*Object Oriented Programming ou OOP*), qui est considérée de nos jours comme la technique de programmation la plus performante. L'un des principaux atouts de ce type de programmation réside dans le fait que l'on peut toujours se servir d'une classe préexistante pour en créer une nouvelle, qui *héritera* toutes ses propriétés mais pourra modifier certaines d'entre elles et/ou y ajouter les siennes propres. Le procédé s'appelle *dérivation*. Il permet de créer toute une hiérarchie de classes allant du général au particulier.

Nous pouvons par exemple définir une classe **Mammifere()**, qui contienne un ensemble de caractéristiques propres à ce type d'animal. A partir de cette classe *parente*, nous pouvons dériver une ou plusieurs classes *filles*, comme par exemple : une classe **Primate()**, une classe **Rongeur()**, une classe **Carnivore()**, etc., qui hériteront toutes les caractéristiques de la classe **Mammifere()**, en y ajoutant leurs spécificités.

Au départ de la classe **Carnivore()**, nous pouvons ensuite dériver une classe **Belette()**, une classe **Loup()**, une classe **Chien()**, etc., qui hériteront encore une fois toutes les caractéristiques de la classe parente avant d'y ajouter les leurs. Exemple :

```
>>> class Mammifere(object):
        caract1 = "il allaite ses petits ;"

>>> class Carnivore(Mammifere):
        caract2 = "il se nourrit de la chair de ses proies ;"

>>> class Chien(Carnivore):
        caract3 = "son cri s'appelle aboiement ;"

>>> mirza = Chien()
>>> print mirza.caract1, mirza.caract2, mirza.caract3
il allaite ses petits ; il se nourrit de la chair de ses proies ;
son cri s'appelle aboiement ;
```

Dans cet exemple, nous voyons que l'objet **mirza** , qui est une instance de la classe **Chien()**, hérite non seulement l'attribut défini pour cette classe, mais également les attributs définis pour les classes parentes.

Vous voyez également dans cet exemple comment il faut procéder pour dériver une classe à partir d'une classe parente : on utilise l'instruction **class**, suivie comme d'habitude du nom que l'on veut attribuer à la nouvelle classe, et on place entre parenthèses le nom de la classe parente. Les classes les plus fondamentales dérivent quant à elles de l'objet « ancêtre » **object**.

Notez bien que les attributs utilisés dans cet exemple sont des attributs des classes (et non des attributs d'instances). L'instance mirza peut accéder à ces attributs, mais pas les modifier :

```
>>> mirza.caract2 = "son corps est couvert de poils ;"   # 1
>>> print mirza.caract2                                  # 2
son corps est couvert de poils ;                        # 3
>>> fido = Chien()                                       # 4
>>> print fido.caract2                                   # 5
il se nourrit de la chair de ses proies ;               # 6
```

Dans ce nouvel exemple, la ligne 1 ne modifie pas l'attribut **caract2** de la classe **Carnivore()**, contrairement à ce que l'on pourrait penser au vu de la ligne 3. Nous pouvons le vérifier en créant une nouvelle instance **fido** (lignes 4 à 6) .

Si vous avez bien assimilé les paragraphes précédents, vous aurez compris que l'instruction de la ligne 1 crée une nouvelle variable d'instance associée seulement à l'objet **mirza**. Il existe donc dès ce moment deux variables avec le même nom **caract2** : l'une dans l'espace de noms de l'objet **mirza**, et l'autre dans l'espace de noms de la classe **Carnivore()**.

Comment faut-il alors interpréter ce qui s'est passé aux lignes 2 et 3 ?

Comme nous l'avons vu plus haut, l'instance **mirza** peut accéder aux variables situées dans son propre espace de noms, mais aussi à celles qui sont situées dans les espaces de noms de toutes les classes parentes. S'il existe des variables aux noms identiques dans plusieurs de ces espaces, laquelle sera-t-elle sélectionnée lors de l'exécution d'une instruction comme celle de la ligne 2 ?

Pour résoudre ce conflit, Python respecte une règle de priorité fort simple. Lorsqu'on lui demande d'utiliser la valeur d'une variable nommée **alpha**, par exemple, il commence par rechercher ce nom dans l'espace local (le plus « interne », en quelque sorte). Si une variable **alpha** est trouvée dans l'espace local, c'est celle-là qui est utilisée, et la recherche s'arrête. Sinon, Python examine l'espace de noms de la structure parente, puis celui de la structure grand-parente, et ainsi de suite jusqu'au niveau principal du programme.

A la ligne 2 de notre exemple, c'est donc la variable d'instance qui sera utilisée. A la ligne 5, par contre, c'est seulement au niveau de la classe grand-parente qu'une variable répondant au nom **caract2** peut être trouvée. C'est donc celle-là qui est affichée.

Héritage et polymorphisme

Analysez soigneusement le script de la page suivante. Il met en œuvre plusieurs concepts décrits précédemment, en particulier le concept d'héritage.

Pour bien comprendre ce script, il faut cependant d'abord vous rappeler quelques notions élémentaires de *chimie*. Dans votre cours de chimie, vous avez certainement dû apprendre que les *atomes* sont des entités constitués d'un certain nombre de *protons* (particules chargées d'électricité positive), *d'électrons* (chargés négativement) et de *neutrons* (neutres).

Le type d'atome (ou élément) est déterminé par le nombre de protons, que l'on appelle également *numéro atomique*. Dans son état fondamental, un atome contient autant d'électrons que de protons, et par conséquent il est électriquement neutre. Il possède également un nombre variable de neutrons, mais ceux-ci n'influencent en aucune manière la charge électrique globale.

Dans certaines circonstances, un atome peut gagner ou perdre des électrons. Il acquiert de ce fait une charge électrique globale, et devient alors un *ion* (il s'agit d'un *ion négatif* si l'atome a gagné un ou plusieurs électrons, et d'un *ion positif* s'il en a perdu). La charge électrique d'un ion est égale à la différence entre le nombre de protons et le nombre d'électrons qu'il contient.

Le script reproduit à la page suivante génère des objets **Atome()** et des objets **Ion()**. Nous avons rappelé ci-dessus qu'un ion est simplement un atome modifié. Dans notre programmation, la

classe qui définit les objets **Ion()** sera donc une *classe dérivée* de la classe **Atome()** : elle héritera d'elle tous ses attributs et toutes ses méthodes, en y ajoutant les siennes propres.

L'une de ces méthodes ajoutées (la méthode **affiche()**) remplace une méthode de même nom héritée de la classe **Atome()**. Les classes **Atome()** et **Ion()** possèdent donc chacune une méthode de même nom, mais qui effectuent un travail différent. On parle dans ce cas de *polymorphisme*. On pourra dire également que la méthode **affiche()** de la classe **Atome()** a été *surchargée*.

Il sera évidemment possible d'instancier un nombre quelconque d'atomes et d'ions à partir de ces deux classes. Or l'une d'entre elles, la classe **Atome()**, doit contenir une version simplifiée du tableau périodique des éléments (tableau de *Mendeléev*), de façon à pouvoir attribuer un nom d'élément chimique, ainsi qu'un nombre de neutrons, à chaque objet généré. Comme il n'est pas souhaitable de recopier tout ce tableau dans chacune des instances, nous le placerons dans un *attribut de classe*. Ainsi ce tableau n'existera qu'en un seul endroit en mémoire, tout en restant accessible à tous les objets qui seront produits à partir de cette classe.

Voyons concrètement comment toutes ces idées s'articulent :

```
class Atome(object):
    """atomes simplifiés, choisis parmi les 10 premiers éléments du TP"""
    table =[None, ('hydrogène',0), ('hélium',2), ('lithium',4),
            ('béryllium',5), ('bore',6), ('carbone',6), ('azote',7),
            ('oxygène',8), ('fluor',10), ('néon',10)]

    def __init__(self, nat):
        "le n° atomique détermine le n. de protons, d'électrons et de neutrons"
        self.np, self.ne = nat, nat       # nat = numéro atomique
        self.nn = Atome.table[nat][1]     # nb. de neutrons trouvés dans table

    def affiche(self):
        print
        print "Nom de l'élément :", Atome.table[self.np][0]
        print "%s protons, %s électrons, %s neutrons" % \
                  (self.np, self.ne, self.nn)

class Ion(Atome):
    """les ions sont des atomes qui ont gagné ou perdu des électrons"""

    def __init__(self, nat, charge):
        "le n° atomique et la charge électrique déterminent l'ion"
        Atome.__init__(self, nat)
        self.ne = self.ne - charge
        self.charge = charge

    def affiche(self):
        "cette méthode remplace celle héritée de la classe parente"
        Atome.affiche(self)         # ... tout en l'utilisant elle-même !
        print "Particule électrisée. Charge =", self.charge

### Programme principal : ###

a1 = Atome(5)
a2 = Ion(3, 1)
a3 = Ion(8, -2)
a1.affiche()
a2.affiche()
a3.affiche()
```

L'exécution de ce script provoque l'affichage suivant :

```
Nom de l'élément : bore
5 protons, 5 électrons, 6 neutrons

Nom de l'élément : lithium
3 protons, 2 électrons, 4 neutrons
```

```
Particule électrisée. Charge = 1

Nom de l'élément : oxygène
8 protons, 10 électrons, 8 neutrons
Particule électrisée. Charge = -2
```

Au niveau du programme principal, vous pouvez constater que l'on instancie les objets **Atome()** en fournissant leur numéro atomique (lequel doit être compris entre 1 et 10). Pour instancier des objets **Ion()**, par contre, on doit fournir un numéro atomique et une charge électrique globale (positive ou négative). La même méthode **affiche()** fait apparaître les propriétés de ces objets, qu'il s'agisse d'atomes ou d'ions, avec dans le cas de l'ion une ligne supplémentaire (*polymorphisme*).

Commentaires :

La définition de la classe **Atome()** commence par l'assignation de la variable **table**. Une variable définie à cet endroit fait partie de l'espace de noms de la classe. C'est donc un *attribut de classe*, dans lequel nous plaçons une liste d'informations concernant les 10 premiers éléments du tableau périodique de *Mendeléev.*

Pour chacun de ces éléments, la liste contient un tuple : (nom de l'élément, nombre de neutrons), à l'indice qui correspond au numéro atomique. Comme il n'existe pas d'élément de numéro atomique zéro, nous avons placé à l'indice zéro dans la liste, l'objet spécial *None*. (A priori, nous aurions pu placer à cet endroit n'importe quelle autre valeur, puisque cet indice ne sera pas utilisé. L'objet *None* de Python nous semble cependant particulièrement explicite).

Viennent ensuite les définitions de deux méthodes :

- Le constructeur **__init__()** sert essentiellement ici à générer trois *attributs d'instance*, destinés à mémoriser respectivement les nombres de protons, d'électrons et de neutrons pour chaque objet atome construit à partir de cette classe (Rappelez-vous que les attributs d'instance sont des variables liées au paramètre **self**).
  Notez au passage la technique utilisée pour obtenir le nombre de neutrons à partir de l'attribut de classe, en mentionnant le nom de la classe elle-même dans une qualification par points, comme par exemple dans l'instruction : `self.nn = Atome.table[nat][1]`.
- La méthode **affiche()** utilise à la fois les attributs d'instance, pour retrouver les nombres de protons, d'électrons et de neutrons de l'objet courant, et l'attribut de classe (lequel est commun à tous les objets) pour en extraire le nom d'élément correspondant.

La définition de la classe **Ion()** inclut dans ses parenthèses le nom de la classe **Atome** qui précède. Il s'agit donc d'une classe dérivée.

Les méthodes de cette classe sont des variantes de celles de la classe **Atome()**. Elles devront donc vraisemblablement faire appel à celles-ci. Cette remarque est importante : Comment peut-on, à l'intérieur de la définition d'une classe, faire appel à une méthode définie dans une autre classe ?

Il ne faut pas perdre de vue, en effet, qu'une méthode se rattache toujours à l'instance qui sera générée à partir de la classe (instance représentée par **self** dans la définition). Si une méthode doit faire appel à une autre méthode définie dans une autre classe, il faut pouvoir lui transmettre la référence de l'instance à laquelle elle doit s'associer. Comment faire ? C'est très simple :

> *Lorsque dans la définition d'une classe, on souhaite faire appel à une méthode définie dans une autre classe, il suffit de l'invoquer directement, via cette autre classe, en lui transmettant la référence de l'instance comme premier argument.*

C'est ainsi que dans notre script, par exemple, la méthode **affiche()** de la classe **Ion()** peut faire appel à la méthode **affiche()** de la classe **Atome()** : les informations affichées seront bien celles de l'objet-ion courant, puisque sa référence a été transmise dans l'instruction d'appel :

```
        Atome.affiche(self)
```

(dans cette instruction, **self** est bien entendu la référence de l'instance courante).

De la même manière (vous en verrez de nombreux autres exemples plus loin), la méthode constructeur de la classe **Ion()** fait appel à la méthode constructeur de sa classe parente, dans :

```
        Atome.__init__(self, nat)
```

Cet appel est nécessaire, afin que les objets de la classe **Ion()** soient initialisés de la même manière que les objets de la classe **Atome()**. Si nous n'effectuons pas cet appel, les objets-ions n'hériteront pas automatiquement les attributs **ne**, **np** et **nn**, car ceux ci sont des *attributs d'instance* créés par la méthode constructeur de la classe **Atome()**, et celle-ci *n'est pas invoquée automatiquement* lorsqu'on instancie des objets d'une classe dérivée.

Comprenez donc bien que l'héritage ne concerne que les classes, et non les instances de ces classes. Lorsque nous disons qu'une classe dérivée hérite toutes les propriétés de sa classe parente, cela ne signifie pas que les propriétés des instances de la classe parente sont automatiquement transmises aux instances de la classe fille. En conséquence, retenez bien que :

> *Dans la méthode constructeur d'une classe dérivée, il faut presque toujours prévoir un appel à la méthode constructeur de sa classe parente.*

Résumé : Définition et utilisation d'une classe

```
###################################
# Programme Python type           #
# auteur : G.Swinnen, Liège, 2003 #
# licence : GPL                   #
###################################

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

class Rectangle:
    """rectangle"""
    def __init__(self, ang, lar, hau):
        self.ang = ang
        self.lar = lar
        self.hau = hau

    def trouveCentre(self):
        xc = self.ang.x + self.lar /2
        yc = self.ang.y + self.hau /2
        return Point(xc, yc)

class Carre(Rectangle):
    """carré = rectangle particulier"""
    def __init__(self, coin, cote):
        Rectangle.__init__(self,
               coin, cote, cote)
        self.cote = cote

    def surface(self):
        return self.cote**2

#########################
## Programme principal : ##

# coord. de 2 coins sup. gauches :
csgR = Point(40,30)
csgC = Point(10,25)

# "boîtes" rectangulaire et carrée :
boiteR = Rectangle(csgR, 100, 50)
boiteC = Carre(csgC, 40)

# Coordonnées du centre pour chacune :
cR = boiteR.trouveCentre()
cC = boiteC.trouveCentre()

print "centre du rect. :", cR.x, cR.y
print "centre du carré :", cC.x, cC.y

print "surf. du carré :",
print boiteC.surface()
```

La classe est un moule servant à produire des objets. Chacun d'eux sera une instance de la classe considérée.

Les instances de la classe Point() seront des objets très simples qui posséderont seulement un attribut 'x' et un attribut 'y' ; ils ne seront dotés d'aucune méthode.

Le paramètre self désigne toutes les instances qui seront produites par cette classe

Les instances de la classe Rectangle() posséderont 3 attributs : le premier ('ang') doit être lui-même un objet de classe Point(). Il servira à mémoriser les coordonnées de l'angle supérieur gauche du rectangle.

La classe Rectangle() comporte une méthode, qui renverra un objet de classe Point() au programme appelant.

Carre() est une classe dérivée, qui hérite les attributs et méthodes de la classe Rectangle(). Son constructeur doit faire appel au constructeur de la classe parente, en lui transmettant la référence de l'instance (self) comme premier argument.

La classe Carre() comporte une méthode de plus que sa classe parente.

Pour créer (ou instancier) un objet, il suffit d'affecter une classe à une variable. Les instructions ci-contre créent donc deux objets de la classe Point()...

... et celles-ci, encore deux autres objets. Note : par convention, le nom d'une classe commence par une lettre majuscule

La méthode trouveCentre() fonctionne pour les objets des deux types, puisque la classe Carre() a hérité de classe Rectangle().

Par contre, la méthode surface() ne peut être invoquée que pour les objets carrés.

Modules contenant des bibliothèques de classes

Vous connaissez déjà depuis longtemps l'utilité des modules Python (cf. pages 60 & 76). Vous savez qu'ils servent à regrouper des bibliothèques de classes et de fonctions. A titre d'exercice de révision, vous allez créer vous-même un nouveau module de classes, en encodant les lignes d'instruction ci-dessous dans un fichier-module que vous nommerez **formes.py** :

```
class Rectangle:
    "Classe de rectangles"
    def __init__(self, longueur =30, largeur =15):
        self.L = longueur
        self.l = largeur
        self.nom ="rectangle"

    def perimetre(self):
        return "(%s + %s) * 2 = %s" % (self.L, self.l,
                                       (self.L + self.l)*2)
    def surface(self):
        return "%s * %s = %s" % (self.L, self.l, self.L*self.l)

    def mesures(self):
        print "Un %s de %s sur %s" % (self.nom, self.L, self.l)
        print "a une surface de %s" % (self.surface(),)
        print "et un périmètre de %s\n" % (self.perimetre(),)

class Carre(Rectangle):
    "Classe de carrés"
    def __init__(self, cote =10):
        Rectangle.__init__(self, cote, cote)
        self.nom ="carré"

if __name__ == "__main__":
    r1 = Rectangle(15, 30)
    r1.mesures()
    c1 = Carre(13)
    c1.mesures()
```

Une fois ce module enregistré, vous pouvez l'utiliser de deux manières : Soit vous en lancez l'exécution comme celle d'un programme ordinaire, soit vous l'importez dans un script quelconque ou depuis la ligne de commande, pour en utiliser les classes. Exemple :

```
>>> import formes
>>> f1 = formes.Rectangle(27, 12)
>>> f1.mesures()
Un rectangle de 27 sur 12
a une surface de 27 * 12 = 324
et un périmètre de (27 + 12) * 2 = 78

>>> f2 = formes.Carre(13)
>>> f2.mesures()
Un carré de 13 sur 13
a une surface de 13 * 13 = 169
et un périmètre de (13 + 13) * 2 = 52
```

On voit dans ce script que la classe **Carre()** est construite par dérivation à partir de la classe **Rectangle()** dont elle hérite toutes les caractéristiques. En d'autres termes, la classe **Carre()** est une classe fille de la classe **Rectangle()**.

Vous pouvez remarquer encore une fois que le constructeur de la classe **Carre()** doit faire appel au constructeur de sa classe parente (`Rectangle.__init__(self, ...)`), en lui transmettant la référence de l'instance (**self**) comme premier argument.

Quant à l'instruction :

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

placée à la fin du module, elle sert à déterminer si le module est « lancé » en tant que programme autonome (auquel cas les instructions qui suivent doivent être exécutées), ou au contraire utilisé comme une bibliothèque de classes importée ailleurs. Dans ce cas cette partie du code est sans effet.

Exercices :

12.5 Définissez une classe **Cercle()**. Les objets construits à partir de cette classe seront des cercles de tailles variées. En plus de la méthode constructeur (qui utilisera donc un paramètre **rayon**), vous définirez une méthode **surface()**, qui devra renvoyer la surface du cercle.

Définissez ensuite une classe **Cylindre()** dérivée de la précédente. Le constructeur de cette nouvelle classe comportera les deux paramètres **rayon** et **hauteur**. Vous y ajouterez une méthode **volume()** qui devra renvoyer le volume du cylindre.
(Rappel : Volume d'un cylindre = surface de section x hauteur).

Exemple d'utilisation de cette classe :

```
>>> cyl = Cylindre(5, 7)
>>> print cyl.surface()
78.54
>>> print cyl.volume()
549.78
```

12.6 Complétez l'exercice précédent en lui ajoutant encore une classe **Cone()**, qui devra dériver cette fois de la classe **Cylindre()**, et dont le constructeur comportera lui aussi les deux paramètres **rayon** et **hauteur**. Cette nouvelle classe possédera sa propre méthode **volume()**, laquelle devra renvoyer le volume du cône.
(Rappel : Volume d'un cône = volume du cylindre correspondant divisé par 3).

Exemple d'utilisation de cette classe :

```
>>> co = Cone(5,7)
>>> print co.volume()
183.26
```

12.7 Définissez une classe **JeuDeCartes()** permettant d'instancier des objets dont le comportement soit similaire à celui d'un vrai jeu de cartes. La classe devra comporter au moins les quatre méthodes suivantes :

- méthode constructeur : création et remplissage d'une liste de 52 éléments, qui sont eux-mêmes des tuples de 2 entiers. Cette liste de tuples contiendra les caractéristiques de chacune des 52 cartes. Pour chacune d'elles, il faut en effet mémoriser séparément un entier indiquant la valeur (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, les 4 dernières valeurs étant celles des valet, dame, roi et as), et un autre entier indiquant la couleur de la carte (c'est-à-dire 3,2,1,0 pour Cœur, Carreau, Trèfle & Pique).
Dans une telle liste, l'élément (11,2) désigne donc le valet de Trèfle, et la liste terminée doit être du type : `[(2, 0), (3,0), (3,0), (4,0), ..... (12,3), (13,3), (14,3)]`
- méthode **nom_carte()** : cette méthode doit renvoyer, sous la forme d'une chaîne, l'identité d'une carte quelconque dont on lui a fourni le tuple descripteur en argument.
Par exemple, l'instruction : `print jeu.nom_carte((14, 3))` doit provoquer l'affichage de : `As de pique`

- méthode **battre()** : comme chacun sait, battre les cartes consiste à les mélanger.
Cette méthode sert donc à mélanger les éléments de la liste contenant les cartes, quel qu'en soit le nombre.
- méthode **tirer()** : lorsque cette méthode est invoquée, une carte est retirée du jeu. Le tuple contenant sa valeur et sa couleur est renvoyé au programme appelant. On retire toujours la première carte de la liste. Si cette méthode est invoquée alors qu'il ne reste plus aucune carte dans la liste, il faut alors renvoyer l'objet spécial None au programme appelant.

Exemple d'utilisation de la classe **JeuDeCartes()** :

```
jeu = JeuDeCartes()                 # instanciation d'un objet
jeu.battre()                        # mélange des cartes
for n in range(53):                 # tirage des 52 cartes :
   c = jeu.tirer()
   if c == None:                    # il ne reste plus aucune carte
      print 'Terminé !'             # dans la liste
   else:
      print jeu.nom_carte(c)        # valeur et couleur de la carte
```

12.8 Complément de l'exercice précédent : Définir deux joueurs A et B. Instancier deux jeux de cartes (un pour chaque joueur) et les mélanger. Ensuite, à l'aide d'une boucle, tirer 52 fois une carte de chacun des deux jeux et comparer leurs valeurs. Si c'est la première des 2 qui a la valeur la plus élevée, on ajoute un point au joueur A. Si la situation contraire se présente, on ajoute un point au joueur B. Si les deux valeurs sont égales, on passe au tirage suivant. Au terme de la boucle, comparer les comptes de A et B pour déterminer le gagnant.

12.9 Écrivez un nouveau script qui récupère le code de l'exercice 12.2 (Compte bancaire) en l'important comme un module. Définissez-y une nouvelle classe **CompteEpargne()**, dérivant de la classe **CompteBancaire()** importée, qui permette de créer des comptes d'épargne rapportant un certain intérêt au cours du temps.
Pour simplifier, nous admettrons que ces intérêts sont calculés tous les mois.
Le constructeur de votre nouvelle classe devra initialiser un taux d'intérêt mensuel par défaut égal à **0,3 %**.
Une méthode **changeTaux(valeur)** devra permettre de modifier ce taux à volonté.
Une méthode **capitalisation(nombreMois)** devra :
- afficher le nombre de mois et le taux d'intérêt pris en compte ;
- calculer le solde atteint en capitalisant les intérêts composés, pour le taux et le nombre de mois qui auront été choisis.
Exemple d'utilisation de la nouvelle classe :

```
>>> c1 = CompteEpargne('Duvivier', 600)
>>> c1.depot(350)
>>> c1.affiche()
Le solde du compte bancaire de Duvivier est de 950 euros.
>>> c1.capitalisation(12)
Capitalisation sur 12 mois au taux mensuel de 0.3 %.
>>> c1.affiche()
Le solde du compte bancaire de Duvivier est de 984.769981274 euros.
>>> c1.changeTaux(.5)
>>> c1.capitalisation(12)
Capitalisation sur 12 mois au taux mensuel de 0.5 %.
>>> c1.affiche()
Le solde du compte bancaire de Duvivier est de 1045.50843891 euros.
```

13

Classes & Interfaces graphiques

La programmation orientée objet convient particulièrement bien au développement d'applications avec interface graphique. Des bibliothèques de classes comme *Tkinter* ou *wxPython* fournissent une base de *widgets* très étoffée, que nous pouvons adapter à nos besoins par dérivation. Dans ce chapitre, nous allons utiliser à nouveau la bibliothèque *Tkinter*, mais en appliquant les concepts décrits dans les pages précédentes, et en nous efforçant de mettre en évidence les avantages qu'apporte l'orientation objet dans nos programmes.

« Code des couleurs » : un petit projet bien encapsulé

Nous allons commencer par un petit projet qui nous a été inspiré par le cours d'initiation à l'électronique. L'application que nous décrivons ci-après permet de retrouver rapidement le code de trois couleurs qui correspond à une résistance électrique de valeur bien déterminée.

Pour rappel, la fonction des résistances électriques consiste à s'opposer (à résister) plus ou moins bien au passage du courant. Les résistances se présentent concrètement sous la forme de petites pièces tubulaires cerclées de bandes de couleur (en général 3). Ces bandes de couleur indiquent la valeur numérique de la résistance, en fonction du code suivant :

Chaque couleur correspond conventionnellement à l'un des chiffres de zéro à neuf :

Noir = 0 ; Brun = 1 ; Rouge = 2 ; Orange = 3 ; Jaune = 4 ;
Vert = 5 ; Bleu = 6 ; Violet = 7 ; Gris = 8 ; Blanc = 9.

On oriente la résistance de manière telle que les bandes colorées soient placées à gauche. La valeur de la résistance – exprimée en ohms (Ω) - s'obtient en lisant ces bandes colorées également à partir de la gauche : les deux premières bandes indiquent les deux premiers chiffres de la valeur numérique ; il faut ensuite accoler à ces deux chiffres un nombre de zéros égal à l'indication fournie par la troisième bande. Exemple concret :

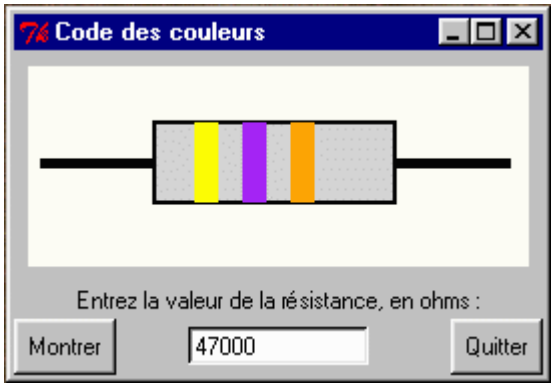
Supposons qu'à partir de la gauche, les bandes colorées soient jaune, violette et verte.
La valeur de cette résistance est 4700000 Ω, ou 4700 kΩ , ou encore 4,7 MΩ .

Ce système ne permet évidemment de préciser une valeur numérique qu'avec deux chiffres significatifs seulement. Il est toutefois considéré comme largement suffisant pour la plupart des applications électroniques « ordinaires » (radio, TV, etc.)

Cahier des charges de notre programme :

Notre application doit faire apparaître une fenêtre comportant un dessin de la résistance, ainsi qu'un champ d'entrée dans lequel l'utilisateur peut encoder une valeur numérique. Un bouton « Montrer » déclenche la modification du dessin de la résistance, de telle façon que les trois bandes de couleur se mettent en accord avec la valeur numérique introduite.

Contrainte : Le programme doit accepter toute entrée numérique fournie sous forme entière ou réelle, dans les limites de 10 à 10^{11} Ω. Par exemple, une valeur telle que 4.78e6 doit être acceptée et arrondie correctement, c'est-à-dire convertie en 4800000 Ω.



Mise en œuvre concrète

Nous construisons le corps de cette application simple sous la forme d'une classe. Nous voulons vous montrer ainsi comment une classe peut servir d'*espace de noms commun*, dans lequel vous pouvez *encapsuler* vos variables et nos fonctions. Le principal intérêt de procéder ainsi est que cela vous permet de vous passer de variables globales. En effet :

- Mettre en route l'application se résumera à instancier un objet de cette classe.
- Les fonctions que l'on voudra y mettre en oeuvre seront les *méthodes* de cet objet-application.
- À l'intérieur de ces méthodes, il suffira de rattacher un nom de variable au paramètre **self**, pour que cette variable soit accessible de partout à l'intérieur de l'objet. Une telle *variable d'instance* est donc tout à fait l'équivalent d'une variable globale (mais seulement à l'intérieur de l'objet), puisque toutes les autres méthodes de cet objet peuvent y accéder par l'intermédiaire de **self**.

```
 1# class Application(object):
 2#     def __init__(self):
 3#         """Constructeur de la fenêtre principale"""
 4#         self.root =Tk()
 5#         self.root.title('Code des couleurs')
 6#         self.dessineResistance()
 7#         Label(self.root, text ="Entrez la valeur de la résistance, en ohms :").\
 8#                 grid(row =2, column =1, columnspan =3)
 9#         Button(self.root, text ='Montrer', command =self.changeCouleurs).\
10#                 grid(row =3, column =1)
11#         Button(self.root, text ='Quitter', command =self.root.quit).\
12#                 grid(row =3, column =3)
13#         self.entree = Entry(self.root, width =14)
14#         self.entree.grid(row =3, column =2)
15#         # Code des couleurs pour les valeurs de zéro à neuf :
16#         self.cc = ['black','brown','red','orange','yellow',
17#                    'green','blue','purple','grey','white']
18#         self.root.mainloop()
19#
20#     def dessineResistance(self):
21#         """Canevas avec un modèle de résistance à trois lignes colorées"""
22#         self.can = Canvas(self.root, width=250, height =100, bg ='light blue')
23#         self.can.grid(row =1, column =1, columnspan =3, pady =5, padx =5)
24#         self.can.create_line(10, 50, 240, 50, width =5)         # fils
25#         self.can.create_rectangle(65, 30, 185, 70, fill ='beige', width =2)
26#         # Dessin des trois lignes colorées (noires au départ) :
27#         self.ligne =[]            # on mémorisera les trois lignes dans 1 liste
28#         for x in range(85,150,24):
29#             self.ligne.append(self.can.create_rectangle(x,30,x+12,70,
30#                                                         fill='black',width=0))
31#
32#     def changeCouleurs(self):
33#         """Affichage des couleurs correspondant à la valeur entrée"""
34#         self.v1ch = self.entree.get()       # cette méthode renvoie une chaîne
35#         try:
36#             v = float(self.v1ch)            # conversion en valeur numérique
37#         except:
38#             err =1                          # erreur : entrée non numérique
```

```
39#          else:
40#              err =0
41#          if err ==1 or v < 10 or v > 1e11 :
42#              self.signaleErreur()            # entrée incorrecte ou hors limites
43#          else:
44#              li =[0]*3                       # liste des 3 codes à afficher
45#              logv = int(log10(v))            # partie entière du logarithme
46#              ordgr = 10**logv                # ordre de grandeur
47#              # extraction du premier chiffre significatif :
48#              li[0] = int(v/ordgr)            # partie entière
49#              decim = v/ordgr - li[0]         # partie décimale
50#              # extraction du second chiffre significatif :
51#              li[1] = int(decim*10 +.5)       # +.5 pour arrondir correctement
52#              # nombre de zéros à accoler aux 2 chiffres significatifs :
53#              li[2] = logv -1
54#              # Coloration des 3 lignes :
55#              for n in range(3):
56#                  self.can.itemconfigure(self.ligne[n], fill =self.cc[li[n]])
57#
58#      def signaleErreur(self):
59#          self.entree.configure(bg ='red')            # colorer le fond du champ
60#          self.root.after(1000, self.videEntree)      # après 1 seconde, effacer
61#
62#      def videEntree(self):
63#          self.entree.configure(bg ='white')          # rétablir le fond blanc
64#          self.entree.delete(0, len(self.v1ch))       # enlever les car. présents
65#
66#  # Programme principal :
67#  if __name__ == '__main__':
68#      from Tkinter import *
69#      from math import log10              # logarithmes en base 10
70#      f = Application()                   # instanciation de l'objet application
```

Commentaires :

- Ligne 1 : La classe est définie comme une nouvelle classe indépendante (elle ne dérive d'aucune classe parente préexistante, mais seulement de **object**, « ancêtre » de toutes les classes).
- Lignes 2 à 14 : Le constructeur de la classe instancie les widgets nécessaires : espace graphique, libellés et boutons. Afin d'améliorer la lisibilité du programme, cependant, nous avons placé l'instanciation du canevas (avec le dessin de la résistance) dans une méthode distincte : **dessineResistance()**. Veuillez remarquer aussi que pour obtenir un code plus compact, nous ne mémorisons pas les boutons et le libellé dans des variables (comme cela a été expliqué à la page 104), parce que nous ne souhaitons pas y faire référence ailleurs dans le programme.
  Le positionnement des widgets dans la fenêtre utilise la méthode **grid()** décrite à la page 101.
- Lignes 15-17 : Le code des couleurs est mémorisé dans une simple liste.
- Ligne 18 : La dernière instruction du constructeur démarre l'application. Si vous préférez démarrer l'application indépendamment de sa création, vous devez supprimer cette ligne, et reporter l'appel à **mainloop()** au niveau principal du programme, en ajoutant une instruction : `f.root.mainloop()` à la ligne 71.
- Lignes 20 à 30 : Le dessin de la résistance se compose d'une ligne et d'un premier rectangle gris clair, pour le corps de la résistance et ses deux fils. Trois autres rectangles figureront les bandes colorées que le programme devra modifier en fonction des entrées de l'utilisateur. Ces bandes sont noires au départ ; elles sont référencées dans la liste **self.ligne**.
- Lignes 32 à 53 : Ces lignes contiennent l'essentiel de la fonctionnalité du programme.
  L'entrée brute fournie par l'utilisateur est acceptée sous la forme d'une chaîne de caractères.
  A la ligne 36, on essaie de convertir cette chaîne en une valeur numérique de type *float*. Si la conversion échoue, on mémorise l'erreur. Si l'on dispose bien d'une valeur numérique, on vé-

rifie ensuite qu'elle se situe effectivement dans l'intervalle autorisé (de 10 Ω à 10^{11} Ω). Si une erreur est détectée, on signale à l'utilisateur que son entrée est incorrecte en colorant de rouge le fond du champ d'entrée, qui est ensuite vidé de son contenu (lignes 55 à 61).

- Lignes 45-46 : Les mathématiques viennent à notre secours pour extraire de la valeur numérique son ordre de grandeur (c'est-à-dire l'exposant de 10 le plus proche). Veuillez consulter votre cours de mathématiques pour de plus amples explications concernant les logarithmes.
- Lignes 47-48 : Une fois connu l'ordre de grandeur, il devient relativement facile d'extraire du nombre traité ses deux premiers chiffres significatifs. Exemple : Supposons que la valeur entrée soit 31687. Le logarithme de ce nombre est 4,50088... dont la partie entière (4) nous donne l'ordre de grandeur de la valeur entrée (soit 10^4). Pour extraire de celle-ci son premier chiffre significatif, il suffit de la diviser par 10^4, soit 10000, et de conserver seulement la partie entière du résultat (3).
- Lignes 49 à 51 : Le résultat de la division effectuée dans le paragraphe précédent est 3,1687.
  Nous récupérons la partie décimale de ce nombre à la ligne 49, soit 0,1687 dans notre exemple.
  Si nous le multiplions par dix, ce nouveau résultat comporte une partie entière qui n'est rien d'autre que notre second chiffre significatif (1 dans notre exemple).
  Nous pourrions facilement extraire ce dernier chiffre, mais puisque c'est le dernier, nous souhaitons encore qu'il soit correctement arrondi. Pour ce faire, il suffit d'ajouter une demi unité au produit de la multiplication par dix, avant d'en extraire la valeur entière. Dans notre exemple, en effet, ce calcul donnera donc 1,687 + 0,5 = 2,187 , dont la partie entière (2) est bien la valeur arrondie recherchée.
- Ligne 53 : Le nombre de zéros à accoler aux deux chiffres significatifs correspond au calcul de l'ordre de grandeur. Il suffit de retirer une unité au logarithme.
- Ligne 56 : Pour attribuer une nouvelle couleur à un objet déjà dessiné dans un canevas, on utilise la méthode **itemconfigure()**. Nous utilisons donc cette méthode pour modifier l'option **fill** de chacune des bandes colorées, en utilisant les noms de couleur extraits de la liste **self.cc** grâce à aux trois indices **li[1]**, **li[2]** et **li[3]** qui contiennent les 3 chiffres correspondants.

13 *Exercices :*

13.1 Modifiez le script ci-dessus de telle manière que le fond d'image devienne bleu clair (*light blue*), que le corps de la résistance devienne beige (*beige*), que le fil de cette résistance soit plus fin, et que les bandes colorées indiquant la valeur soient plus larges.

13.2 Modifiez le script ci-dessus de telle manière que l'image dessinée soit deux fois plus grande.

13.3 Modifiez le script ci-dessus de telle manière qu'il devienne possible d'entrer aussi des valeurs de résistances comprises entre 1 et 10 Ω. Pour ces valeurs, le premier anneau coloré devra rester noir, les deux autres indiqueront la valeur en Ω et dixièmes d' Ω.

13.4 Modifiez le script ci-dessus de telle façon que le bouton « Montrer » ne soit plus nécessaire. Dans votre script modifié, il suffira de frapper <*Enter*> après avoir entré la valeur de la résistance, pour que l'affichage s'active.

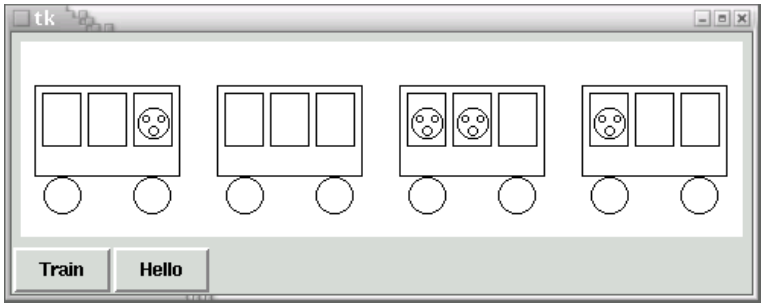
13.5 Modifiez le script ci-dessus de telle manière que les trois bandes colorées redeviennent noires dans les cas où l'utilisateur fournit une entrée inacceptable.

« Petit train » : héritage, échange d'informations entre classes

Dans l'exercice précédent, nous n'avons exploité qu'une seule caractéristique des classes : *l'encapsulation*. Celle-ci nous a permis d'écrire un programme dans lequel les différentes fonctions (qui sont donc devenues des *méthodes*) peuvent chacune accéder à un même *pool* de variables : toutes celles qui sont définies comme étant attachées à **self**. Toutes ces variables peuvent être considérées en quelque sorte comme des variables globales à l'intérieur de l'objet.

Comprenez bien toutefois qu'il ne s'agit pas de véritables variables globales. Elles restent en effet strictement confinées à l'intérieur de l'objet, et il est déconseillé de vouloir y accéder de l'extérieur[56]. D'autre part, tous les objets que vous instancierez à partir d'une même classe posséderont chacun leur propre jeu de ces variables, qui sont donc bel et bien *encapsulées* dans ces objets. On les appelle pour cette raison des *attributs d'instance*.

Nous allons à présent passer à la vitesse supérieure, et réaliser une petite application sur la base de plusieurs classes, afin d'examiner comment différents objets peuvent *s'échanger des informations par l'intermédiaire de leurs méthodes*. Nous allons également profiter de cet exercice pour vous montrer comment vous pouvez définir la classe principale de votre application graphique par dérivation d'une classe Tkinter préexistante, mettant ainsi à profit le mécanisme d'*héritage*.



Le projet développé ici très simple, mais il pourrait constituer une première étape dans la réalisation d'un logiciel de jeu : nous en fournissons d'ailleurs des exemples plus loin (voir page 235). Il s'agit d'une fenêtre contenant un canevas et deux boutons. Lorsque l'on actionne le premier de ces deux boutons, un petit train apparaît dans le canevas. Lorsque l'on actionne le second bouton, quelques petits personnages apparaissent à certaines fenêtres des wagons.

Cahier des charges :

L'application comportera deux classes :

- La classe **Application()** sera obtenue par dérivation d'une des classes de base de Tkinter : elle mettra en place la fenêtre principale, son canevas et ses deux boutons.
- Une classe **Wagon()**, indépendante, permettra d'instancier dans le canevas 4 objets-wagons similaires, dotés chacun d'une méthode **perso()**. Celle-ci sera destinée à provoquer l'apparition d'un petit personnage à l'une quelconque des trois fenêtres du wagon. L'application principale invoquera cette méthode différemment pour différents objets-wagons, afin de faire apparaître un choix de quelques personnages.

[56] Comme nous l'avons déjà signalé précédemment, Python vous permet d'accéder aux attributs d'instance en utilisant la qualification des noms par points. D'autres langages de programmation l'interdisent, ou bien ne l'autorisent que moyennant une déclaration particulière de ces attributs (distinction entre attributs privés et publics).
Sachez en tous cas que ce n'est pas recommandé : le bon usage de la programmation orientée objet stipule en effet que vous ne devez pouvoir accéder aux attributs des objets que par l'intermédiaire de méthodes spécifiques (l'interface).

Implémentation :

```
 1#  from Tkinter import *
 2#
 3#  def cercle(can, x, y, r):
 4#      "dessin d'un cercle de rayon <r> en <x,y> dans le canevas <can>"
 5#      can.create_oval(x-r, y-r, x+r, y+r)
 6#
 7#  class Application(Tk):
 8#      def __init__(self):
 9#          Tk.__init__(self)        # constructeur de la classe parente
10#          self.can =Canvas(self, width =475, height =130, bg ="white")
11#          self.can.pack(side =TOP, padx =5, pady =5)
12#          Button(self, text ="Train", command =self.dessine).pack(side =LEFT)
13#          Button(self, text ="Hello", command =self.coucou).pack(side =LEFT)
14#
15#      def dessine(self):
16#          "instanciation de 4 wagons dans le canevas"
17#          self.w1 = Wagon(self.can, 10, 30)
18#          self.w2 = Wagon(self.can, 130, 30)
19#          self.w3 = Wagon(self.can, 250, 30)
20#          self.w4 = Wagon(self.can, 370, 30)
21#
22#      def coucou(self):
23#          "apparition de personnages dans certaines fenêtres"
24#          self.w1.perso(3)        # 1er wagon, 3e fenêtre
25#          self.w3.perso(1)        # 3e wagon, 1e fenêtre
26#          self.w3.perso(2)        # 3e wagon, 2e fenêtre
27#          self.w4.perso(1)        # 4e wagon, 1e fenêtre
28#
29#  class Wagon(object):
30#      def __init__(self, canev, x, y):
31#          "dessin d'un petit wagon en <x,y> dans le canevas <canev>"
32#          # mémorisation des paramètres dans des variables d'instance :
33#          self.canev, self.x, self.y = canev, x, y
34#          # rectangle de base : 95x60 pixels :
35#          canev.create_rectangle(x, y, x+95, y+60)
36#          # 3 fenêtres de 25x40 pixels, écartées de 5 pixels :
37#          for xf in range(x+5, x+90, 30):
38#              canev.create_rectangle(xf, y+5, xf+25, y+40)
39#          # 2 roues de rayon égal à 12 pixels  :
40#          cercle(canev, x+18, y+73, 12)
41#          cercle(canev, x+77, y+73, 12)
42#
43#      def perso(self, fen):
44#          "apparition d'un petit personnage à la fenêtre <fen>"
45#          # calcul des coordonnées du centre de chaque fenêtre :
46#          xf = self.x + fen*30 -12
47#          yf = self.y + 25
48#          cercle(self.canev, xf, yf, 10)      # visage
49#          cercle(self.canev, xf-5, yf-3, 2)   # oeil gauche
50#          cercle(self.canev, xf+5, yf-3, 2)   # oeil droit
51#          cercle(self.canev, xf, yf+5, 3)     # bouche
52#
53#  app = Application()
54#  app.mainloop()
```

Commentaires :

- Lignes 3 à 5 : Nous projetons de dessiner une série de petits cercles. Cette petite fonction nous facilitera le travail en nous permettant de définir ces cercles à partir de leur centre et leur rayon.

- Lignes 7 à 13 : La classe principale de notre application est construite par dérivation de la classe de fenêtres **Tk()** importée du module **Tkinter**.[57] Comme nous l'avons expliqué au chapitre précédent, le constructeur d'une classe dérivée doit activer lui-même le constructeur de la classe parente, en lui transmettant la référence de l'instance comme premier argument.
  Les lignes 10 à 13 servent à mettre en place le canevas et les boutons.
- Lignes 15 à 20 : Ces lignes instancient les 4 objets-wagons, produits à partir de la classe correspondante. Ceci pourrait être programmé plus élégamment à l'aide d'une boucle et d'une liste, mais nous le laissons ainsi pour ne pas alourdir inutilement les explications qui suivent.
  Nous voulons placer nos objets-wagons dans le canevas, à des emplacements bien précis : il nous faut donc transmettre quelques informations au constructeur de ces objets : au moins la référence du canevas, ainsi que les coordonnées souhaitées. Ces considérations nous font également entrevoir que lorsque nous définirons la classe **Wagon()**, un peu plus loin, nous devrons associer à sa méthode constructeur un nombre égal de paramètres afin de réceptionner ces arguments.
- Lignes 22 à 27 : Cette méthode est invoquée lorsque l'on actionne le second bouton. Elle invoque elle-même la méthode **perso()** de certains objets-wagons, avec des arguments différents, afin de faire apparaître les personnages aux fenêtres indiquées. Ces quelques lignes de code vous montrent donc comment un objet peut communiquer avec un autre, en faisant appel à ses méthodes. Il s'agit là du mécanisme central de la programmation par objets :

> *Les objets sont des entités programmées qui s'échangent des messages et interagissent par l'intermédiaire de leurs méthodes.*

- Idéalement, la méthode **coucou()** devrait comporter quelques instructions complémentaires, lesquelles vérifieraient d'abord si les objets-wagons concernés existent bel et bien, avant d'autoriser l'activation d'une de leurs méthodes. Nous n'avons pas inclus ce genre de garde-fou afin que l'exemple reste aussi simple que possible, mais cela entraîne la conséquence que vous ne pouvez pas actionner le second bouton avant le premier.
- Lignes 29-30 : La classe **Wagon()** ne dérive d'aucune autre classe préexistante. Étant donné qu'il s'agit d'une classe d'objets graphiques, nous devons cependant munir sa méthode constructeur de paramètres, afin de recevoir la référence du canevas auquel les dessins sont destinés, ainsi que les coordonnées de départ de ces dessins. Dans vos expérimentations éventuelles autour de cet exercice, vous pourriez bien évidemment ajouter encore d'autres paramètres : taille du dessin, orientation, couleur, vitesse, etc.
- Lignes 31 à 51 : Ces instructions ne nécessitent guère de commentaires. La méthode **perso()** est dotée d'un paramètre qui indique celle des 3 fenêtres où il faut faire apparaître un petit personnage. Ici aussi nous n'avons pas prévu de garde-fou : vous pouvez invoquer cette méthode avec un argument égal à 4 ou 5, par exemple, ce qui produira des effets incorrects.
- Lignes 53-54 : Pour cette application, contrairement à la précédente, nous avons préféré séparer la création de l'objet **app**, et son démarrage par invocation de **mainloop()**, dans deux instructions distinctes (en guise d'exemple). Vous pourriez également condenser ces deux instructions en une seule, laquelle serait alors : `Application().mainloop()` , et faire ainsi l'économie d'une variable.

Exercice :

[57] Nous verrons plus loin que *Tkinter* autorise également de construire la fenêtre principale d'une application par dérivation d'une classe de *widget* (le plus souvent, il s'agira d'un *widget* **Frame()**). La fenêtre englobant ce *widget* sera automatiquement ajoutée. (Voir page 199).

13.6 Perfectionnez le script décrit ci-dessus, en ajoutant un paramètre **couleur** au constructeur de la classe **Wagon()**, lequel déterminera la couleur de la cabine du wagon. Arrangez-vous également pour que les fenêtres soient noires au départ, et les roues grises (pour réaliser ce dernier objectif, ajoutez aussi un paramètre **couleur** à la fonction **cercle()**).
À cette même classe **Wagon()**, ajoutez encore une méthode **allumer()**, qui servira à changer la couleur des 3 fenêtres (initialement noires) en jaune, afin de simuler l'allumage d'un éclairage intérieur.
Ajoutez un bouton à la fenêtre principale, qui puisse déclencher cet allumage. Profitez de l'amélioration de la fonction **cercle()** pour teinter le visage des petits personnages en rose (*pink*), leurs yeux et leurs bouches en noir, et instanciez les objets-wagons avec des couleurs différentes.

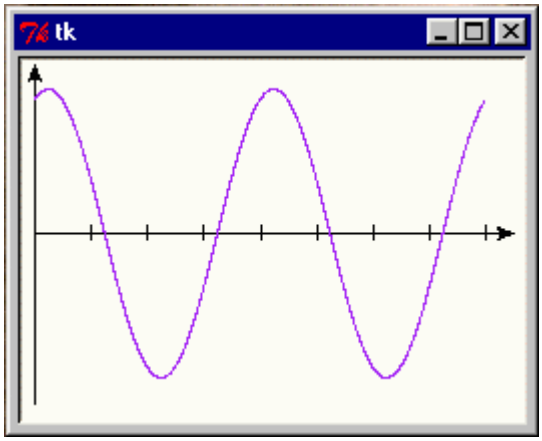
13.7 Ajoutez des correctifs au programme précédent, afin que l'on puisse utiliser n'importe quel bouton dans le désordre, sans que cela ne déclenche une erreur ou un effet bizarre.

« OscilloGraphe » : un widget personnalisé

Le projet qui suit va nous entraîner encore un petit peu plus loin. Nous allons y construire *une nouvelle classe de widget*, qu'il sera possible d'intégrer dans nos projets futurs comme n'importe quel widget standard. Comme la classe principale de l'exercice précédent, cette nouvelle classe sera construite par dérivation d'une classe Tkinter préexistante.

Le sujet concret de cette application nous est inspiré par le cours de physique. Pour rappel :

Un mouvement vibratoire harmonique se définit comme étant la projection d'un mouvement circulaire uniforme sur une droite. Les positions successives d'un mobile qui effectue ce type de mouvement sont traditionnellement repérées par rapport à une position centrale : on les appelle alors des *élongations*. L'équation qui décrit l'évolution de l'élongation d'un tel mobile au cours du temps est toujours de la forme $e = A \sin(2\pi f t + \varphi)$, dans laquelle e représente l'élongation du mobile à tout instant t . Les constantes A, f et φ désignent respectivement l'*amplitude*, la *fréquence* et la *phase* du mouvement vibratoire.



Le but du présent projet est de fournir un instrument de visualisation simple de ces différents concepts, à savoir un système d'affichage automatique de graphiques élongation/temps. L'utilisateur pourra choisir librement les valeurs des paramètres A, f et φ , et observer les courbes qui en résultent.

Le widget que nous allons construire d'abord s'occupera de l'affichage proprement dit. Nous construirons ensuite d'autres widgets pour faciliter l'entrée des paramètres A, f et φ .

Veuillez donc encoder le script ci-dessous et le sauvegarder dans un fichier, auquel vous donnerez le nom **oscillo.py** . Vous réaliserez ainsi *un véritable module contenant une classe* (vous pourrez par la suite ajouter d'autres classes dans ce même module, si le cœur vous en dit).

```
1#  from Tkinter import *
2#  from math import sin, pi
3#
4#  class OscilloGraphe(Canvas):
5#      "Canevas spécialisé, pour dessiner des courbes élongation/temps"
6#      def __init__(self, boss =None, larg=200, haut=150):
```

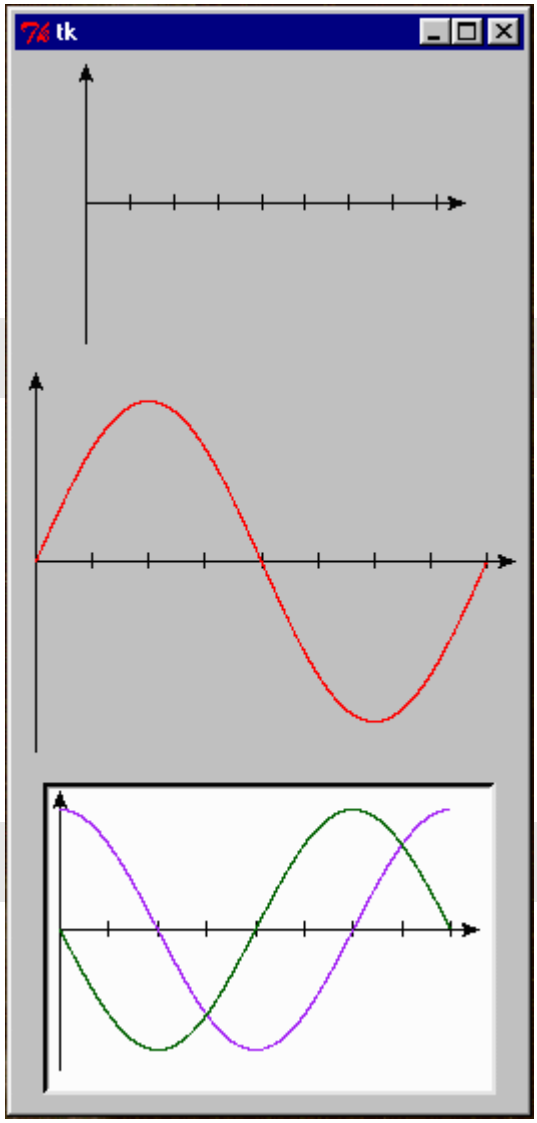
```
 7#         "Constructeur du graphique : axes et échelle horiz."
 8#         # construction du widget parent :
 9#         Canvas.__init__(self)                          # appel au constructeur
10#         self.configure(width=larg, height=haut)        # de la classe parente
11#         self.larg, self.haut = larg, haut                      # mémorisation
12#         # tracé des axes de référence :
13#         self.create_line(10, haut/2, larg, haut/2, arrow=LAST)   # axe X
14#         self.create_line(10, haut-5, 10, 5, arrow=LAST)          # axe Y
15#         # tracé d'une échelle avec 8 graduations :
16#         pas = (larg-25)/8.          # intervalles de l'échelle horizontale
17#         for t in range(1, 9):
18#             stx = 10 + t*pas        # +10 pour partir de l'origine
19#             self.create_line(stx, haut/2-4, stx, haut/2+4)
20#
21#     def traceCourbe(self, freq=1, phase=0, ampl=10, coul='red'):
22#         "tracé d'un graphique élongation/temps sur 1 seconde"
23#         curve =[]                       # liste des coordonnées
24#         pas = (self.larg-25)/1000.      # l'échelle X correspond à 1 seconde
25#         for t in range(0,1001,5):       # que l'on divise en 1000 ms.
26#             e = ampl*sin(2*pi*freq*t/1000 - phase)
27#             x = 10 + t*pas
28#             y = self.haut/2 - e*self.haut/25
29#             curve.append((x,y))
30#         n = self.create_line(curve, fill=coul, smooth=1)
31#         return n                        # n = numéro d'ordre du tracé
32#
33# #### Code pour tester la classe : ####
34#
35# if __name__ == '__main__':
36#     root = Tk()
37#     gra = OscilloGraphe(root, 250, 180)
38#     gra.pack()
39#     gra.configure(bg ='ivory', bd =2, relief=SUNKEN)
40#     gra.traceCourbe(2, 1.2, 10, 'purple')
41#     root.mainloop()
```

Le niveau principal du script est constitué par les lignes 35 à 41. Comme nous l'avons déjà expliqué à la page 180, les lignes de code situées après l'instruction `if __name__ == '__main__':` ne sont pas exécutées si le script est importé en tant que module. Si on lance le script comme application principale, par contre, ces instructions s'exécutent.

Nous disposons ainsi d'un mécanisme intéressant, qui nous permet d'intégrer des instructions de test à l'intérieur des modules, même si ceux-ci sont destinés à être importés dans d'autres scripts.

Lancez donc l'exécution du script de la manière habituelle. Vous devriez obtenir un affichage similaire à celui qui est reproduit à la page précédente.

Expérimentation :



Nous commenterons les lignes importantes du script un peu plus loin dans ce texte. Mais commençons d'abord par expérimenter quelque peu la classe que nous venons de construire.

Ouvrez donc votre terminal, et entrez les instructions ci-dessous directement à la ligne de commande :

```
>>> from oscillo import *
>>> g1 = OscilloGraphe()
>>> g1.pack()
```

Après importation des classes du module **oscillo**, nous instancions un premier objet **g1** , de la classe **OscilloGraphe()**.

Puisque nous ne fournissons aucun argument, l'objet possède les dimensions par défaut, définies dans le constructeur de la classe. Remarquons au passage que nous n'avons même pas pris la peine de définir d'abord une fenêtre maître pour y placer ensuite notre widget. Tkinter nous pardonne cet oubli et nous en fournit une automatiquement !

```
>>> g2 = OscilloGraphe(haut=200, larg=250)
>>> g2.pack()
>>> g2.traceCourbe()
```

Par ces instructions, nous créons un second widget de la même classe, en précisant cette fois ses dimensions (hauteur et largeur, dans n'importe quel ordre).

Ensuite, nous activons la méthode **traceCourbe()** associée à ce widget. Étant donné que nous ne lui fournissons aucun argument, la sinusoïde qui apparaît correspond aux valeurs prévues par défaut pour les paramètres A, f et φ .

```
>>> g3 = OscilloGraphe(larg=220)
>>> g3.configure(bg='white', bd=3, relief=SUNKEN)
>>> g3.pack(padx=5,pady=5)
>>> g3.traceCourbe(phase=1.57, coul='purple')
>>> g3.traceCourbe(phase=3.14, coul='dark green')
```

Pour comprendre la configuration de ce troisième widget, il faut nous rappeler que la classe **OscilloGraphe()** a été construite par dérivation de la classe **Canvas()**. Elle hérite donc toutes les propriétés de celle-ci, ce qui nous permet de choisir la couleur de fond, la bordure, etc., en utilisant les mêmes arguments que ceux qui sont à notre disposition lorsque nous configurons un canevas.

Nous faisons ensuite apparaître deux tracés successifs, en faisant appel deux fois à la méthode **traceCourbe()**, à laquelle nous fournissons des arguments pour la phase et la couleur.

Exercice :

13.8 Créez un quatrième widget, de taille : 400 x 300, couleur de fond : jaune, et faites-y apparaître plusieurs courbes correspondant à des fréquences et des amplitudes différentes.

Il est temps à présent que nous analysions la structure de la classe qui nous a permis d'instancier tous ces widgets. Nous avons donc enregistré cette classe dans le module **oscillo.py** (voir page 190).

Cahier des charges :

Nous souhaitons définir une nouvelle classe de widget, capable d'afficher automatiquement les graphiques élongation/temps correspondant à divers mouvements vibratoires harmoniques.

Ce widget doit pouvoir être dimensionné à volonté au moment de son instanciation. Il doit faire apparaître deux axes cartésiens X et Y munis de flèches. L'axe X représentera l'écoulement du temps pendant une seconde au total, et il sera muni d'une échelle comportant 8 intervalles.

Une méthode **traceCourbe()** sera associée à ce widget. Elle provoquera le tracé du graphique élongation/temps pour un mouvement vibratoire, dont on aura fourni la *fréquence* (entre 0.25 et 10 Hz), la *phase* (entre 0 et 2π radians) et *l'amplitude* (entre 1 et 10 ; échelle arbitraire).

Implémentation :

- Ligne 4 : La classe **OscilloGraphe()** est créée par dérivation de la classe **Canvas()**.
  Elle hérite donc toutes les propriétés de celle-ci : on pourra configurer les objets de cette nouvelle classe en utilisant les nombreuses options déjà disponibles pour la classe **Canvas()**.
- Ligne 6 : La méthode constructeur utilise 3 paramètres, qui sont tous optionnels puisque chacun d'entre eux possède une valeur par défaut. Le paramètre **boss** ne sert qu'à réceptionner la référence d'une fenêtre maîtresse éventuelle (voir exemples suivants). Les paramètres **larg** et **haut** (largeur et hauteur) servent à assigner des valeurs aux options **width** et **height** du canevas parent, au moment de l'instanciation.
- Lignes 9 & 10 : Comme nous l'avons déjà dit à plusieurs reprises, le constructeur d'une classe dérivée doit presque toujours commencer par activer le constructeur de sa classe parente. Nous ne pouvons en effet hériter toute la fonctionnalité de la classe parente, que si cette fonctionnalité a été effectivement mise en place et initialisée.
  Nous activons donc le constructeur de la classe **Canvas()** à la ligne 9 , et nous ajustons deux de ses options à la ligne 10. Notez au passage que nous pourrions condenser ces deux lignes en une seule, qui deviendrait en l'occurrence : `Canvas.__init__(self, width=larg, height=haut)`
  ... et comme cela a également déjà été expliqué (cf. page 177), nous devons transmettre à ce constructeur la référence de l'instance présente (self) comme premier argument.
- Ligne 11 : Il est nécessaire de mémoriser les paramètres **larg** et **haut** dans des variables d'instance, parce que nous devrons pouvoir y accéder aussi dans la méthode **traceCourbe()**.
- Lignes 13 & 14 : Pour tracer les axes X et Y, nous utilisons les paramètres **larg** et **haut**, ainsi ces axes sont automatiquement mis à dimension. L'option `arrow=LAST` permet de faire apparaître une petite flèche à l'extrémité de chaque ligne.
- Lignes 16 à 19 : Pour tracer l'échelle horizontale, on commence par réduire de 25 pixels la largeur disponible, de manière à ménager des espaces aux deux extrémités. On divise ensuite en 8 intervalles, que l'on visualise sous la forme de 8 petits traits verticaux.
- Ligne 21 : La méthode **traceCourbe()** pourra être invoquée avec quatre arguments.
  Chacun d'entre eux pourra éventuellement être omis, puisque chacun des paramètres correspondants possède une valeur par défaut. Il sera également possible de fournir les arguments dans n'importe quel ordre, comme nous l'avons déjà expliqué à la page 81.

- Lignes 23 à 31 : Pour le tracé de la courbe, la variable **t** prend successivement toutes les valeurs de 0 à 1000, et on calcule à chaque fois l'élongation e correspondante, à l'aide de la formule théorique (ligne 26). Les couples de valeurs **t** & **e** ainsi trouvées sont mises à l'échelle et transformées en coordonnées **x, y** aux lignes 27 & 28, puis accumulées dans la liste **curve**.
- Lignes 30 & 31 : La méthode **create_line()** trace alors la courbe correspondante en une seule opération, et elle renvoie le numéro d'ordre du nouvel objet ainsi instancié dans le canevas (ce numéro d'ordre nous permettra d'y accéder encore par après : pour l'effacer, par exemple). L'option `smooth =1` améliore l'aspect final, par lissage.

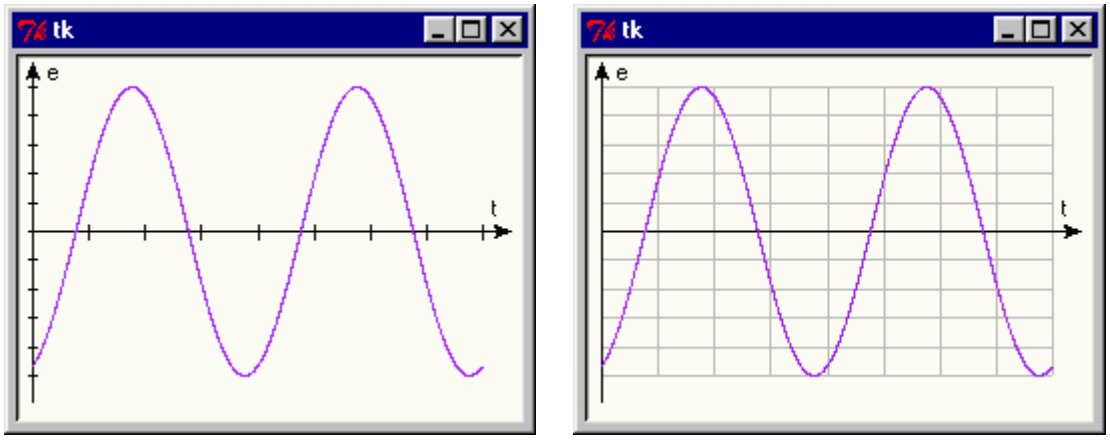
Exercices :

13.9 Modifiez le script de manière à ce que l'axe de référence vertical comporte lui aussi une échelle, avec 5 tirets de part et d'autre de l'origine.

13.10 Comme les widgets de la classe **Canvas()** dont il dérive, votre widget peut intégrer des indications textuelles. Il suffit pour cela d'utiliser la méthode **create_text()**. Cette méthode attend au moins trois arguments : les coordonnées **x** et **y** de l'emplacement où vous voulez faire apparaître votre texte, et puis le texte lui-même, bien entendu. D'autres arguments peuvent être transmis sous forme d'options, pour préciser par exemple la police de caractères et sa taille. Afin de voir comment cela fonctionne, ajoutez provisoirement la ligne suivante dans le constructeur de la classe **OscilloGraphe()**, puis relancez le script :

```
self.create_text(130, 30, text = "Essai", anchor =CENTER)
```

Utilisez cette méthode pour ajouter au widget les indications suivantes aux extrémités des axes de référence : **e** (pour « élongation ») le long de l'axe vertical, et **t** (pour « temps ») le long de l'axe horizontal. Le résultat pourrait ressembler à ceci (figure de gauche) :



13.11 Vous pouvez compléter encore votre widget, en y faisant apparaître une grille de référence, plutôt que de simples tirets le long des axes. Pour éviter que cette grille ne soit trop visible, vous pouvez colorer ses traits en gris (option `fill = 'grey'`), comme dans la figure de droite.

13.12 Complétez encore votre widget en y faisant apparaître des repères numériques.

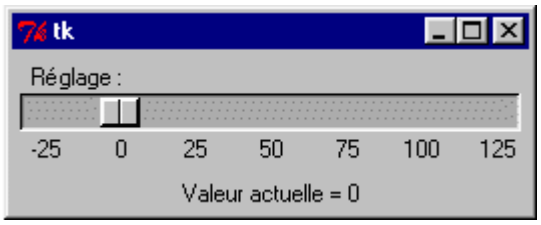
« Curseurs » : un widget composite

Dans l'exercice précédent, vous avez construit un nouveau type de widget que vous avez sauvegardé dans le module **oscillo.py**. Conservez soigneusement ce module, car vous l'intégrerez bientôt dans un projet plus complexe.

Pour l'instant, vous allez construire encore un autre widget, plus interactif cette fois. Il s'agira d'une sorte de panneau de contrôle comportant trois curseurs de réglage et une case à cocher. Comme le précédent, ce widget est destiné à être réutilisé dans une application de synthèse.

Présentation du widget « Scale »

Commençons d'abord par découvrir un widget de base, que nous n'avions pas encore utilisé jusqu'ici :
Le widget Scale se présente comme un curseur qui coulisse devant une échelle. Il permet à l'utilisateur de choisir rapidement la valeur d'un paramètre quelconque, d'une manière très attrayante.



Le petit script ci-dessous vous montre comment le paramétrer et l'utiliser dans une fenêtre :

```
from Tkinter import *

def updateLabel(x):
    lab.configure(text='Valeur actuelle = ' + str(x))

root = Tk()
Scale(root, length=250, orient=HORIZONTAL, label ='Réglage :',
      troughcolor ='dark grey', sliderlength =20,
      showvalue =0, from_=-25, to=125, tickinterval =25,
      command=updateLabel).pack()
lab = Label(root)
lab.pack()

root.mainloop()
```

Ces lignes ne nécessitent guère de commentaires.

Vous pouvez créer des widgets **Scale** de n'importe quelle taille (option length), en orientation horizontale (comme dans notre exemple) ou verticale (option **orient = VERTICAL**).
Les options **from_** (attention : n'oubliez pas le caractère 'souligné', lequel est nécessaire afin d'éviter la confusion avec le mot réservé **from** !) et **to** définissent la plage de réglage. L'intervalle entre les repères numériques est défini dans l'option **tickinterval**, etc.

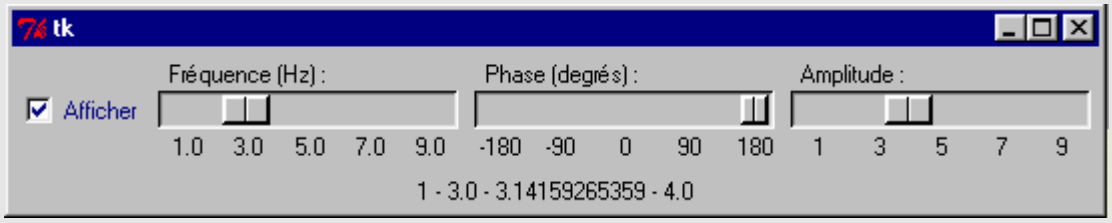
La fonction désignée dans l'option **command** est appelée automatiquement chaque fois que le curseur est déplacé, et la position actuelle du curseur par rapport à l'échelle lui est transmise en argument. Il est donc très facile d'utiliser cette valeur pour effectuer un traitement quelconque. Considérez par exemple le paramètre **x** de la fonction **updateLabel()**, dans notre exemple.

Le widget **Scale** constitue une interface très intuitive et attrayante pour proposer différents réglages aux utilisateurs de vos programmes. Nous allons à présent l'incorporer en plusieurs exemplaires dans une nouvelle classe de widget : un panneau de contrôle destiné à choisir la fréquence, la phase et l'amplitude pour un mouvement vibratoire, dont nous afficherons ensuite le graphique élongation/temps à l'aide du widget **oscilloGraphe** construit dans les pages précédentes.

Construction d'un panneau de contrôle à trois curseurs

Comme le précédent, le script que nous décrivons ci-dessous est destiné à être sauvegardé dans un module, que vous nommerez cette fois **curseurs.py**. Les classes que vous sauvegardez ainsi seront réutilisées (par importation) dans une application de synthèse que nous décrirons un peu plus loin[58]. Nous attirons votre attention sur le fait que le code ci-dessous peut être raccourci de différentes manières (Nous y reviendrons). Nous ne l'avons pas optimisé d'emblée, parce que cela nécessiterait d'y incorporer un concept supplémentaire (les *expressions lambda*), ce que nous préférons éviter pour l'instant.

Vous savez déjà que les lignes de code placées à la fin du script permettent de tester son fonctionnement. Vous devriez obtenir une fenêtre semblable à celle-ci :




```
 1#  from Tkinter import *
 2#  from math import pi
 3#
 4#  class ChoixVibra(Frame):
 5#      """Curseurs pour choisir fréquence, phase & amplitude d'une vibration"""
 6#      def __init__(self, boss =None, coul ='red'):
 7#          Frame.__init__(self)        # constructeur de la classe parente
 8#          # Initialisation de quelques attributs d'instance :
 9#          self.freq, self.phase, self.ampl, self.coul = 0, 0, 0, coul
10#          # Variable d'état de la case à cocher :
11#          self.chk = IntVar()                # 'objet-variable' Tkinter
12#          Checkbutton(self, text='Afficher', variable=self.chk,
13#                      fg = self.coul, command = self.setCurve).pack(side=LEFT)
14#          # Définition des 3 widgets curseurs :
15#          Scale(self, length=150, orient=HORIZONTAL, sliderlength =25,
16#                label ='Fréquence (Hz) :', from_=1., to=9., tickinterval =2,
17#                resolution =0.25,
18#                showvalue =0, command = self.setFrequency).pack(side=LEFT)
19#          Scale(self, length=150, orient=HORIZONTAL, sliderlength =15,
20#                label ='Phase (degrés) :', from_=-180, to=180, tickinterval =90,
21#                showvalue =0, command = self.setPhase).pack(side=LEFT)
22#          Scale(self, length=150, orient=HORIZONTAL, sliderlength =25,
23#                label ='Amplitude :', from_=1, to=9, tickinterval =2,
24#                showvalue =0, command = self.setAmplitude).pack(side=LEFT)
25#
26#      def setCurve(self):
27#          self.event_generate('<Control-Z>')
28#
29#      def setFrequency(self, f):
30#          self.freq = float(f)
31#          self.event_generate('<Control-Z>')
32#
33#      def setPhase(self, p):
34#          pp =float(p)
35#          self.phase = pp*2*pi/360        # conversion degrés -> radians
36#          self.event_generate('<Control-Z>')
37#
38#      def setAmplitude(self, a):
39#          self.ampl = float(a)
40#          self.event_generate('<Control-Z>')
41#
42#  #### Code pour tester la classe : ###
```

[58] Vous pourriez bien évidemment aussi enregistrer plusieurs classes dans un même module.

```
43#
44# if __name__ == '__main__':
45#     def afficherTout(event=None):
46#         lab.configure(text = '%s - %s - %s - %s' %
47#                               (fra.chk.get(), fra.freq, fra.phase, fra.ampl))
48#     root = Tk()
49#     fra = ChoixVibra(root,'navy')
50#     fra.pack(side =TOP)
51#     lab = Label(root, text ='test')
52#     lab.pack()
53#     root.bind('<Control-Z>', afficherTout)
54#     root.mainloop()
```

Ce panneau de contrôle permettra à vos utilisateurs de régler aisément la valeur des paramètres indiqués (fréquence, phase & amplitude), lesquels pourront alors servir à commander l'affichage de graphiques élongation/temps dans un widget de la classe **OscilloGraphe()** construite précédemment, comme nous le montrerons dans l'application de synthèse.

Commentaires :

- Ligne 6 : La méthode « constructeur » utilise un paramètre optionnel **coul**. Ce paramètre permettra de choisir une couleur pour le graphique soumis au contrôle du widget. Le paramètre **boss** sert à réceptionner la référence d'une fenêtre maîtresse éventuelle (voir plus loin).
- Ligne 7 : Activation du constructeur de la classe parente (pour hériter sa fonctionnalité).
- Ligne 9 : Déclaration de quelques variables d'instance. Leurs vraies valeurs seront déterminées par les méthodes des lignes 29 à 40 (gestionnaires d'événements).
- Ligne 11 : Cette instruction instancie un objet de la classe **IntVar()**, laquelle fait partie du module Tkinter au même titre que les classes similaires **DoubleVar()**, **StringVar()** et **BooleanVar()**. Toutes ces classes permettent de définir des *variables Tkinter*, lesquels sont en fait des objets, mais qui se se comportent comme des variables à l'intérieur des widgets Tkinter (voir ci-après).
  Ainsi l'objet référencé dans **self.chk** contient l'équivalent d'une variable de type entier, dans un format utilisable par Tkinter. Pour accéder à sa valeur depuis Python, il faut utiliser des méthodes spécifiques de cette classe d'objets : la méthode **set()** permet de lui assigner une valeur, et la méthode **get()** permet de la récupérer (ce que l'on mettra en pratique à la ligne 47).
- Ligne 12 : L'option **variable** de l'objet **checkbutton** est associée à la *variable Tkinter* définie à la ligne précédente. (Nous ne pouvons pas référencer directement une variable ordinaire dans la définition d'un widget Tkinter, parce que Tkinter lui-même est écrit dans un langage qui n'utilise pas les mêmes conventions que Python pour formater ses variables. Les objets construits à partir des classes de *variables Tkinter* sont donc nécessaires pour assurer l'interface).
- Ligne 13 : L'option **command** désigne la méthode que le système doit invoquer lorsque l'utilisateur effectue un clic de souris dans la case à cocher.
- Lignes 14 à 24 : Ces lignes définissent les trois widgets curseurs, en trois instructions similaires. Il serait plus élégant de programmer tout ceci en une seule instruction, répétée trois fois à l'aide d'une boucle. Cela nécessiterait cependant de faire appel à un concept que nous n'avons pas encore expliqué (les *fonctions ou expressions lamdba*), et la définition du gestionnaire d'événements associé à ces widgets deviendrait elle aussi plus complexe. Conservons donc pour cette fois des instructions séparées : nous nous efforcerons d'améliorer tout cela plus tard.
- Lignes 26 à 40 : Les 4 widgets définis dans les lignes précédentes possèdent chacun une option **command**. Pour chacun d'eux, la méthode invoquée dans cette option **command** est diffé-

rente : la case à cocher active la méthode **setCurve()**, le premier curseur active la méthode **setFrequency()**, le second curseur active la méthode **setPhase()**, et le troisième curseur active la méthode **setAmplitude()**. Remarquez bien au passage que l'option **command** des widgets **Scale** transmet un argument à la méthode associée (la position actuelle du curseur), alors que la même option **command** ne transmet rien dans le cas du widget **Checkbutton.**

Ces 4 méthodes (qui sont donc les gestionnaires des événements produits par la case à cocher et les trois curseurs) provoquent elles-mêmes chacune l'émission d'un nouvel *événement*[59], en faisant appel à la méthode **event_generate()**.

Lorsque cette méthode est invoquée, Python envoie au système d'exploitation exactement le même message-événement que celui qui se produirait si l'utilisateur enfonçait simultanément les touches <Ctrl>, <Maj> et <Z> de son clavier.
Nous produisons ainsi un message-événement bien particulier, qui peut être détecté et traité par un gestionnaire d'événement associé à un autre widget (voir page suivante). De cette manière, nous mettons en place un véritable système de communication entre widgets : chaque fois que l'utilisateur exerce une action sur notre panneau de contrôle, celui-ci génère un événement spécifique, qui signale cette action à l'attention des autres widgets présents.

Note : nous aurions pu choisir une autre combinaison de touches (ou même carrément un autre type d'événement). Notre choix s'est porté sur celle-ci parce qu'il y a vraiment très peu de chances que l'utilisateur s'en serve alors qu'il examine notre programme. Nous pourrons cependant produire nous-mêmes un tel événement au clavier à titre de test, lorsque le moment sera venu de vérifier le gestionnaire de cet événement, que nous mettrons en place par ailleurs.

- Lignes 42 à 54 : Comme nous l'avions déjà fait pour **oscillo.py**, nous complétons ce nouveau module par quelques lignes de code au niveau principal. Ces lignes permettent de tester le bon fonctionnement de la classe : elles ne s'exécutent que si on lance le module directement, comme une application à part entière. Veillez à utiliser vous-même cette technique dans vos propres modules, car elle constitue une bonne pratique de programmation : l'utilisateur de modules construits ainsi peut en effet (re)découvrir très aisément leur fonctionnalité (en les exécutant) et la manière de s'en servir (en analysant ces quelques lignes de code).

  Dans ces lignes de test, nous construisons une fenêtre principale **root** qui contient deux widgets : un widget de la nouvelle classe **ChoixVibra()** et un widget de la classe **Label()**.

  A la ligne 53, nous associons à la fenêtre principale un gestionnaire d'événement : tout événement du type spécifié déclenche désormais un appel de la fonction **afficherTout()**.
  Cette fonction est donc notre gestionnaire d'événement spécialisé, qui est sollicité chaque fois qu'un événement de type <Maj-Ctrl-Z> est détecté par le système d'exploitation.

  Comme nous l'avons déjà expliqué plus haut, nous avons fait en sorte que de tels événements soient produits par les objets de la classe **ChoixVibra()**, chaque fois que l'utilisateur modifie l'état de l'un ou l'autre des trois curseurs, ou celui de la case à cocher.
- Conçue seulement pour effectuer un test, la fonction **afficherTout()** ne fait rien d'autre que provoquer l'affichage des valeurs des variables associées à chacun de nos quatre widgets, en

[59] En fait, on devrait plutôt appeler cela un message (qui est lui-même la notification d'un événement). Veuillez relire à ce sujet les explications de la page 89 : ***Programmes pilotés par des événements***.

(re)configurant l'option **text** d'un widget de classe **Label()**.

- Ligne 47, expression `fra.chk.get()` : nous avons vu plus haut que la variable mémorisant l'état de la case à cocher est un objet-variable Tkinter. Python ne peut pas lire directement le contenu d'une telle variable, qui est en réalité un objet-interface. Pour en extraire la valeur, il faut donc faire usage d'une méthode spécifique de cette classe d'objets : la méthode **get()**.

Propagation des événements

Le mécanisme de communication décrit ci-dessus respecte la hiérarchie de classes des widgets. Vous aurez noté que la méthode qui déclenche l'événement est associée au widget dont nous sommes en train de définir la classe, par l'intermédiaire de self. En général, un message-événement est en effet associé à un widget particulier (par exemple, un clic de souris sur un bouton est associé à ce bouton), ce qui signifie que le système d'exploitation va d'abord examiner s'il existe un gestionnaire pour ce type d'événement, qui soit lui aussi associé à ce widget. S'il en existe un, c'est celui-là qui est activé, et la propagation du message s'arrête. Sinon, le message-événement est « présenté » successivement aux widgets maîtres, dans l'ordre hiérarchique, jusqu'à ce qu'un gestionnaire d'événement soit trouvé, ou bien jusqu'à ce que la fenêtre principale soit atteinte.

Les événements correspondant à des frappes sur le clavier (telle la combinaison de touches `<Maj-Ctrl-Z>` utilisée dans notre exercice) sont cependant toujours expédiés directement à la fenêtre principale de l'application. Dans notre exemple, le gestionnaire de cet événement doit donc être associé à la fenêtre **root**.

Exercices :

13.13 Votre nouveau widget hérite des propriétés de la classe **Frame()**. Vous pouvez donc modifier son aspect en modifiant les options par défaut de cette classe, à l'aide de la méthode **configure()**. Essayez par exemple de faire en sorte que le panneau de contrôle soit entouré d'une bordure de 4 pixels ayant l'aspect d'un sillon (`bd = 4, relief = GROOVE`). Si vous ne comprenez pas bien ce qu'il faut faire, inspirez-vous du script **oscillo.py** (ligne 10).

13.14 Si l'on assigne la valeur 1 à l'option **showvalue** des widgets **Scale()**, la position précise du curseur par rapport à l'échelle est affichée en permanence. Activez donc cette fonctionnalité pour le curseur qui contrôle le paramètre « phase ».

13.15 L'option **troughcolor** des widgets **Scale()** permet de définir la couleur de leur glissière. Utilisez cette option pour faire en sorte que la couleur des glissières des 3 curseurs soit celle qui est utilisée comme paramètre lors de l'instanciation de votre nouveau widget.

13.16 Modifiez le script de telle manière que les widgets curseurs soient écartés davantage les uns des autres (options **padx** et **pady** de la méthode **pack()**).

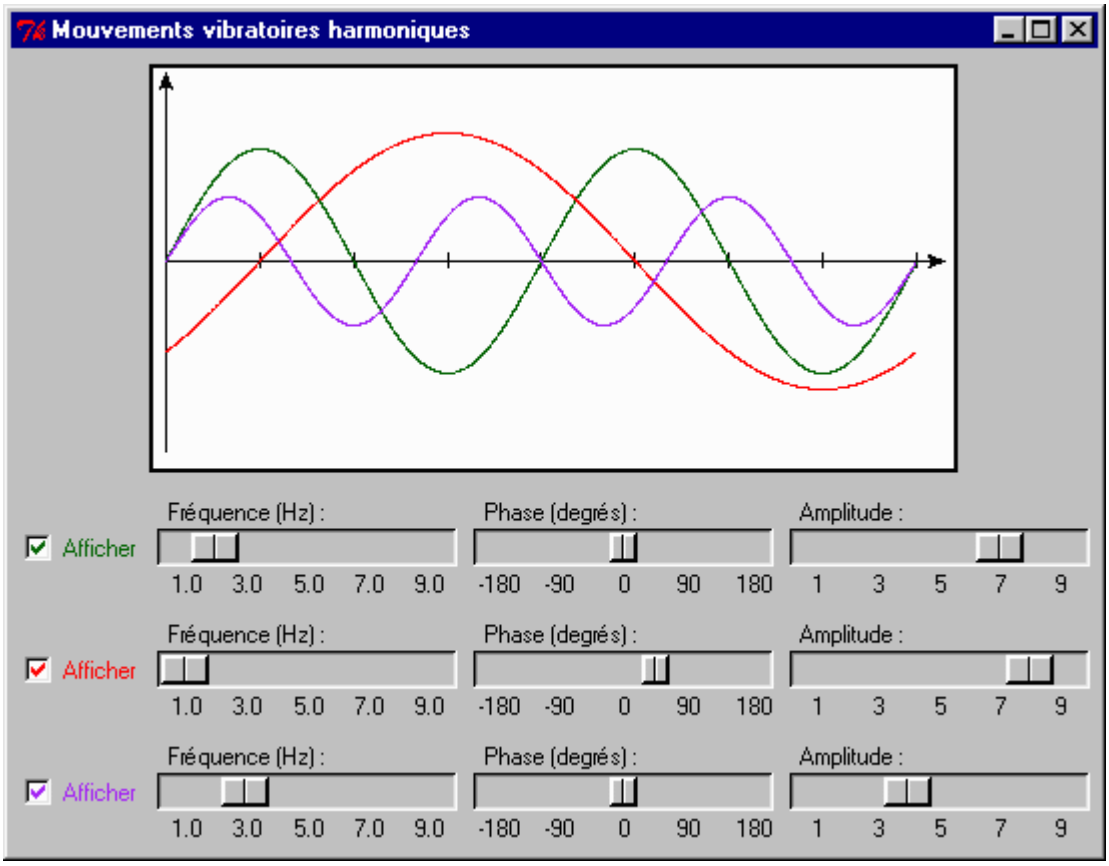
Intégration de widgets composites dans une application synthèse

Dans les exercices précédents, nous avons construit deux nouvelles classes de widgets : le widget **OscilloGraphe()**, canevas spécialisé pour le dessin de sinusoïdes, et le widget **ChoixVibra()**, panneau de contrôle à trois curseurs permettant de choisir les paramètres d'une vibration.

Ces widgets sont désormais disponibles dans les modules **oscillo.py** et **curseurs.py**[60]

[60] Il va de soi que nous pourrions aussi rassembler toutes les classes que nous construisons dans un seul module.

Nous allons à présent les utiliser dans une application synthèse, qui pourrait illustrer votre cours de physique : un widget **OscilloGraphe()** y affiche un, deux, ou trois graphiques superposés, de couleurs différentes, chacun d'entre eux étant soumis au contrôle d'un widget **ChoixVibra()** :



Le script correspondant est reproduit ci-après.

Nous attirons votre attention sur la technique mise en œuvre pour provoquer un rafraîchissement de l'affichage dans le canevas par l'intermédiaire d'un événement, chaque fois que l'utilisateur effectue une action quelconque au niveau de l'un des panneaux de contrôle.

Rappelez-vous que les applications destinées à fonctionner dans une interface graphique doivent être conçues comme des « programmes pilotés par les événements » (voir page 89).

En préparant cet exemple, nous avons arbitrairement décidé que l'affichage des graphiques serait déclenché par un événement particulier, tout à fait similaire à ceux que génère le système d'exploitation lorsque l'utilisateur accomplit une action quelconque. Dans la gamme (très étendue) d'événements possibles, nous en avons choisi un qui ne risque guère d'être utilisé pour d'autres raisons, pendant que notre application fonctionne : la combinaison de touches <Maj-Ctrl-Z>.

Lorsque nous avons construit la classe de widgets **ChoixVibra()**, nous y avons donc incorporé les instructions nécessaires pour que de tels événements soient générés, chaque fois que l'utilisateur actionne l'un des curseurs ou modifie l'état de la case à cocher. Nous allons à présent définir le gestionnaire de cet événement et l'inclure dans notre nouvelle classe : nous l'appellerons **montreCourbes()** et il se chargera de rafraîchir l'affichage. Étant donné que l'événement concerné est du type <enfoncement d'une touche>, nous devrons cependant le détecter au niveau de la fenêtre principale de l'application.

```
1#  from oscillo import *
2#  from curseurs import *
```

```
 3#
 4# class ShowVibra(Frame):
 5#     """Démonstration de mouvements vibratoires harmoniques"""
 6#     def __init__(self, boss =None):
 7#         Frame.__init__(self)        # constructeur de la classe parente
 8#         self.couleur = ['dark green', 'red', 'purple']
 9#         self.trace = [0]*3          # liste des tracés (courbes à dessiner)
10#         self.controle = [0]*3       # liste des panneaux de contrôle
11#
12#         # Instanciation du canevas avec axes X et Y :
13#         self.gra = OscilloGraphe(self, larg =400, haut=200)
14#         self.gra.configure(bg ='white', bd=2, relief=SOLID)
15#         self.gra.pack(side =TOP, pady=5)
16#
17#         # Instanciation de 3 panneaux de contrôle (curseurs) :
18#         for i in range(3):
19#             self.controle[i] = ChoixVibra(self, self.couleur[i])
20#             self.controle[i].pack()
21#
22#         # Désignation de l'événement qui déclenche l'affichage des tracés :
23#         self.master.bind('<Control-Z>', self.montreCourbes)
24#         self.master.title('Mouvements vibratoires harmoniques')
25#         self.pack()
26#
27#     def montreCourbes(self, event):
28#         """(Ré)Affichage des trois graphiques élongation/temps"""
29#         for i in range(3):
30#
31#             # D'abord, effacer le tracé précédent (éventuel) :
32#             self.gra.delete(self.trace[i])
33#
34#             # Ensuite, dessiner le nouveau tracé :
35#             if self.controle[i].chk.get():
36#                 self.trace[i] = self.gra.traceCourbe(
37#                                     coul = self.couleur[i],
38#                                     freq = self.controle[i].freq,
39#                                     phase = self.controle[i].phase,
40#                                     ampl = self.controle[i].ampl)
41#
42# #### Code pour tester la classe : ###
43#
44# if __name__ == '__main__':
45#     ShowVibra().mainloop()
```

Commentaires :

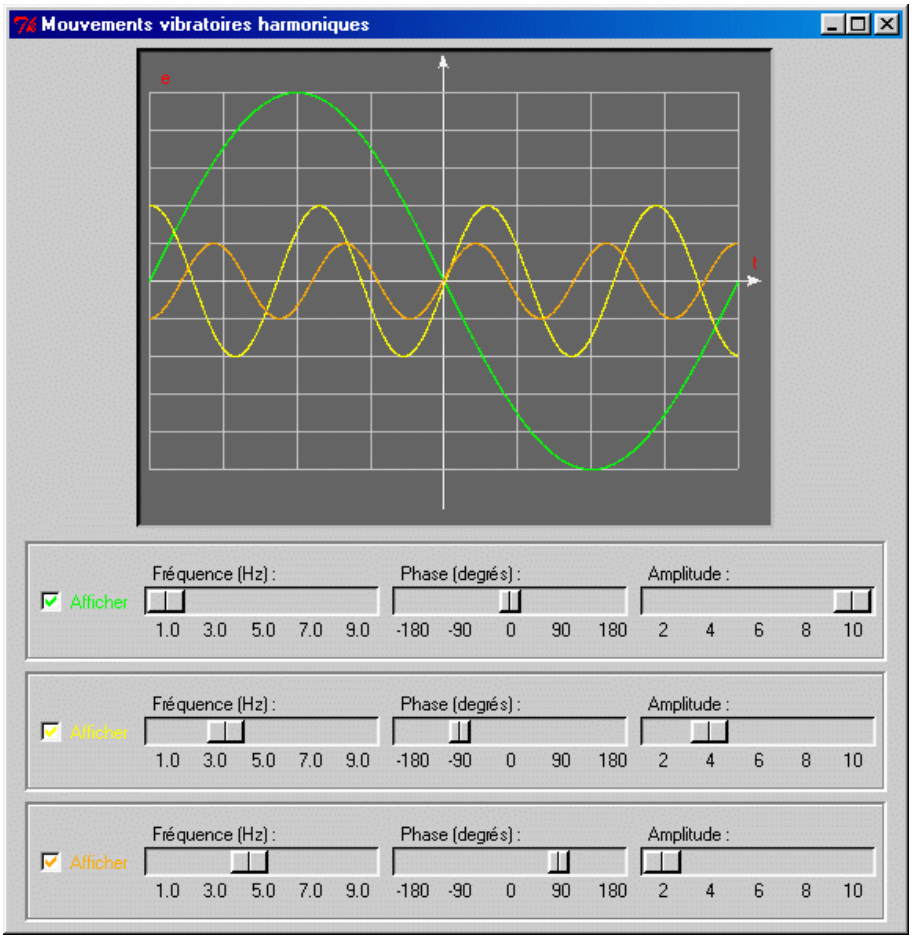
- Lignes 1-2 : Nous pouvons nous passer d'importer le module Tkinter : chacun de ces deux modules s'en charge déjà.
- Ligne 4 : Puisque nous commençons à connaître les bonnes techniques, nous décidons de construire l'application elle-même sous la forme d'une nouvelle classe de widget, dérivée de la classe **Frame()** : ainsi nous pourrons plus tard l'intégrer toute entière dans d'autres projets, si le cœur nous en dit.
- Lignes 8-10 : Définition de quelques variables d'instance (3 listes) : les trois courbes tracées seront des objets graphiques, dont les couleurs sont pré-définies dans la liste **self.couleur** ; nous devons préparer également une liste **self.trace** pour mémoriser les références de ces objets graphiques, et enfin une liste **self.controle** pour mémoriser les références des trois panneaux de contrôle.
- Lignes 13 à 15 : Instanciation du widget d'affichage. Étant donné que la classe **OscilloGraphe()** a été obtenue par dérivation de la classe **Canvas()**, il est toujours possible de configurer ce widget en redéfinissant les options spécifiques de cette classe (ligne 13).

- Lignes 18 à 20 : Pour instancier les trois widgets « panneau de contrôle », on utilise une boucle. Leurs références sont mémorisées dans la liste **self.controle** préparée à la ligne 10. Ces panneaux de contrôle sont instanciés comme esclaves du présent widget, par l'intermédiaire du paramètre **self**. Un second paramètre leur transmet la couleur du tracé à contrôler.
- Lignes 23-24 : Au moment de son instanciation, chaque widget Tkinter reçoit automatiquement un attribut **master** qui contient la référence de la fenêtre principale de l'application. Cet attribut se révèle particulièrement utile si la fenêtre principale a été instanciée implicitement par Tkinter, comme c'est le cas ici.
  Rappelons en effet que lorsque nous démarrons une application en instanciant directement un widget tel que **Frame()**, par exemple (c'est ce que nous avons fait à la ligne 4), Tkinter instancie automatiquement une fenêtre maîtresse pour ce widget (un objet de la classe **Tk()**).
  Comme cet objet a été créé automatiquement, nous ne disposons d'aucune référence dans notre code pour y accéder, si ce n'est par l'intermédiaire de cet attribut **master** que Tkinter associe automatiquement à chaque widget.
  Nous nous servons de cette référence pour redéfinir le bandeau-titre de la fenêtre principale (à la ligne 24), et pour y attacher un gestionnaire d'événement (à la ligne 23).
- Lignes 27 à 40 : La méthode décrite ici est le gestionnaire des événements <Maj-Ctrl-Z> spécifiquement déclenchés par nos widgets **ChoixVibra()** (ou « panneaux de contrôle »), chaque fois que l'utilisateur exerce une action sur un curseur ou une case à cocher. Dans tous les cas, les graphiques éventuellement présents sont d'abord effacés (ligne 28) à l'aide de la méthode **delete()** : le widget **OscilloGraphe()** a hérité cette méthode de sa classe parente **Canvas()**.
  Ensuite, de nouvelles courbes sont retracées, pour chacun des panneaux de contrôle dont on a coché la case « Afficher ». Chacun des objets ainsi dessinés dans le canevas possède un numéro de référence, renvoyé par la méthode **traceCourbe()** de notre widget **OscilloGraphe()**.
  Les numéros de référence de nos dessins sont mémorisés dans la liste **self.trace**.
  Ils permettent d'effacer individuellement chacun d'entre eux (cfr. instruction de la ligne 28).
- Lignes 38-40 : Les valeurs de fréquence, phase & amplitude que l'on transmet à la méthode **traceCourbe()** sont les attributs d'instance correspondants de chacun des trois panneaux de contrôle, eux-mêmes mémorisés dans la liste **self.controle**. Nous pouvons récupérer ces attributs en utilisant la qualification des noms par points.

Exercices :

13.17 Modifiez le script, de manière à obtenir l'aspect ci-dessous (écran d'affichage avec grille de référence, panneaux de contrôle entourés d'un sillon) :

13.18 Modifiez le script, de manière à faire apparaître et contrôler 4 graphiques au lieu de trois. Pour la couleur du quatrième graphique, choisissez par exemple : 'blue', 'navy', 'maroon', ...



13.19 Aux lignes 33-35, nous récupérons les valeurs des fréquence, phase & amplitude choisies par l'utilisateur sur chacun des trois panneaux de contrôle, en accédant directement aux attributs d'instance correspondants. Python autorise ce raccourci - et c'est bien pratique – mais cette technique est dangereuse. Elle enfreint l'une des recommandations de la théorie générale de la « programmation orientée objet », qui préconise que l'accès aux propriétés des objets soit toujours pris en charge par des méthodes spécifiques. Pour respecter cette recommandation, ajoutez à la classe **ChoixVibra()** une méthode supplémentaire que vous appellerez **valeurs()**, et qui renverra un tuple contenant les valeurs de la fréquence, la phase et l'amplitude choisies. Les lignes 33 à 35 du présent script pourront alors être remplacées par quelque chose comme :

```
freq, phase, ampl = self.control[i].valeurs()
```

13.20 Écrivez une petite application qui fait apparaître une fenêtre avec un canevas et un widget curseur (**Scale**). Dans le canevas, dessinez un cercle, dont l'utilisateur pourra faire varier la taille à l'aide du curseur.

13.21 Écrivez un script qui créera deux classes : une classe "Application", dérivée de **Frame()**, dont le constructeur instanciera un canevas de 400x400 pixels, ainsi que deux boutons. Dans le canevas, vous instancierez un objet de la classe "Visage" décrite ci-après.
La classe "Visage" servira à définir des objets graphiques censés représenter des visages humains simplifiés. Ces visages seront constitués d'un cercle principal dans lequel trois ovales plus petits représenteront deux yeux et une bouche (ouverte). Une méthode "fermer" permettra de remplacer l'ovale de la bouche par une ligne horizontale. Une méthode "ouvrir" permettra de restituer la bouche de forme ovale.
Les deux boutons définis dans la classe "Application" serviront respectivement à fermer et à ouvrir la bouche de l'objet Visage installé dans le canevas.
(Vous pouvez vous inspirer de l'exemple de la page 94 pour composer une partie du code).

13.22 Exercice de synthèse : élaboration d'un dictionnaire de couleurs.

But : réaliser un petit programme utilitaire, qui puisse vous aider à construire facilement et rapidement un nouveau dictionnaire de couleurs, lequel permettrait l'accès technique à une couleur quelconque par l'intermédiaire de son nom usuel en français.

Contexte : En manipulant divers objets colorés avec Tkinter, vous avez constaté que cette bibliothèque graphique accepte qu'on lui désigne les couleurs les plus fondamentales sous la forme de chaînes de caractères contenant leur nom en anglais : *red, blue, yellow,* etc.

Vous savez cependant qu'un ordinateur ne peut traiter que des informations numérisées. Cela implique que la désignation d'une couleur quelconque doit nécessairement tôt ou tard être encodée sous la forme d'un nombre. Il faut bien entendu adopter pour cela une une convention, et celle-ci peut varier d'un système à un autre. L'une de ces conventions, parmi les plus courantes, consiste à représenter une couleur à l'aide de trois octets, qui indiqueront respectivement les intensités des trois composantes rouge, verte et bleue de cette couleur.

Cette convention peut être utilisée avec Tkinter pour accéder à n'importe quelle nuance colorée. Vous pouvez en effet lui indiquer la couleur d'un élément graphique quelconque, à l'aide d'une chaîne de 7 caractères telle que `'#00FA4E'`. Dans cette chaîne, le premier caractère (#) signifie que ce qui suit est une valeur hexadécimale. Les six caractères suivants représentent les 3 valeurs hexadécimales des 3 composantes R, V & B.
Pour visualiser concrètement la correspondance entre une couleur quelconque et son code, vous pouvez essayer le petit programme utilitaire **tkColorChooser.py (**qui se trouve généralement dans le sous-répertoire **/lib-tk** de votre installation de Python).

Étant donné qu'il n'est pas facile pour les humains que nous sommes de mémoriser de tels codes hexadécimaux, Tkinter est également doté d'un dictionnaire de conversion, qui autorise l'utilisation de noms communs pour un certain nombre de couleurs parmi les plus courantes, mais cela ne marche que pour des noms de couleurs exprimés en anglais.

Le but du présent exercice est de réaliser un logiciel qui facilitera la construction d'un dictionnaire équivalent en français, lequel pourrait ensuite être incorporé à l'un ou l'autre de vos propres programmes. Une fois construit, ce dictionnaire serait donc de la forme :
`{'vert':'#00FF00', 'bleu':'#0000FF', ... etc ...}`.

Cahier des charges :

L'application à réaliser sera une application graphique, construite autour d'une classe.

Elle comportera une fenêtre avec un certain nombre de champs d'entrée et de boutons, afin que l'utilisateur puisse aisément encoder de nouvelles couleurs en indiquant à chaque fois son nom français dans un champ, et son code hexadécimal dans un autre.

Lorsque le dictionnaire contiendra déjà un certain nombre de données, il devra être possible de le tester, c'est-à-dire d'entrer un nom de couleur en français et de retrouver le code hexadécimal correspondant à l'aide d'un bouton (avec affichage éventuel d'une zone colorée).

Un bouton provoquera l'enregistrement du dictionnaire dans un fichier texte. Un autre permettra de reconstruire le dictionnaire à partir du fichier.

13.23 Le script ci-dessous correspond à une ébauche de projet dessinant des ensembles de dés à jouer disposés à l'écran de plusieurs manières différentes (cette ébauche pourrait être une première étape dans la réalisation d'un logiciel de jeu).
L'exercice consistera à analyser ce script et à le compléter. Vous vous placerez ainsi dans la

situation d'un programmeur chargé de continuer le travail commencé par quelqu'un d'autre, ou encore dans celle de l'informaticien prié de participer à un travail d'équipe.

A) Commencez par analyser ce script, et ajoutez-y des commentaires, en particulier aux lignes marquées : **#***** , afin de montrer que vous comprenez ce que doit faire le programme à ces emplacements :

```
from Tkinter import *

class FaceDom:
    def __init__(self, can, val, pos, taille =70):
        self.can =can
        # ***
        x, y, c = pos[0], pos[1], taille/2
        can.create_rectangle(x -c, y-c, x+c, y+c, fill ='ivory', width =2)
        d = taille/3
        # ***
        self.pList =[]
        # ***
        pDispo = [((0,0),), ((-d,d),(d,-d)), ((-d,-d), (0,0), (d,d))]
        disp = pDispo[val -1]
        # ***
        for p in disp:
            self.cercle(x +p[0], y +p[1], 5, 'red')

    def cercle(self, x, y, r, coul):
        # ***
        self.pList.append(self.can.create_oval(x-r, y-r, x+r, y+r, fill=coul))

    def effacer(self):
        # ***
        for p in self.pList:
            self.can.delete(p)

class Projet(Frame):
    def __init__(self, larg, haut):
        Frame.__init__(self)
        self.larg, self.haut = larg, haut
        self.can = Canvas(self, bg='dark green', width =larg, height =haut)
        self.can.pack(padx =5, pady =5)
        # ***
        bList = [("A", self.boutA), ("B", self.boutB),
                 ("C", self.boutC), ("D", self.boutD),
                 ("Quitter", self.boutQuit)]
        for b in bList:
            Button(self, text =b[0], command =b[1]).pack(side =LEFT)
        self.pack()

    def boutA(self):
        self.d3 = FaceDom(self.can, 3, (100,100), 50)

    def boutB(self):
        self.d2 = FaceDom(self.can, 2, (200,100), 80)

    def boutC(self):
        self.d1 = FaceDom(self.can, 1, (350,100), 110)

    def boutD(self):
        # ***
        self.d3.effacer()

    def boutQuit(self):
        self.master.destroy()

Projet(500, 300).mainloop()
```

B) Modifiez ensuite ce script, afin qu'il corresponde au cahier des charges suivant :

Le canevas devra être plus grand : 600 x 600 pixels.
Les boutons de commande devront être déplacés à droite et espacés davantage.
La taille des points sur une face de dé devra varier proportionnellement à la taille de cette face

Variante 1 :

Ne conservez que les 2 boutons A et B. Chaque utilisation du bouton A fera apparaître 3 nouveaux dés (de même taille, plutôt petits) disposés sur une colonne (verticale), les valeurs de ces dés étant tirées au hasard entre 1 et 6. Chaque nouvelle colonne sera disposée à la droite de la précédente. Si l'un des tirages de 3 dés correspond à 4, 2, 1 (dans n'importe quel ordre), un message « gagné » sera affiché dans la fenêtre (ou dans le canevas). Le bouton B provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Variante 2 :

Ne conservez que les 2 boutons A et B. Le bouton A fera apparaître 5 dés disposés en quinconce (c.à.d. comme les points d'une face de valeur 5). Les valeurs de ces dés seront tirées au hasard entre 1 et 6, mais il ne pourra pas y avoir de doublons. Le bouton B provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Variante 3 :

Ne conservez que les 3 boutons A, B et C. Le bouton A fera apparaître 13 dés de même taille disposés en cercle. Chaque utilisation du bouton B provoquera un changement de valeur du premier dé, puis du deuxième, du troisième, etc. La nouvelle valeur d'un dé sera à chaque fois égale a sa valeur précédente augmentée d'une unité, sauf dans le cas ou la valeur précédente était 6 : dans ce cas la nouvelle valeur est 1, et ainsi de suite. Le bouton C provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

Variante 4 :

Ne conservez que les 3 boutons A, B et C. Le bouton A fera apparaître 12 dés de même taille disposés sur deux lignes de 6. Les valeurs des dés de la première ligne seront dans l'ordre 1, 2, 3, 4, 5, 6. Les valeurs des dés de la seconde ligne seront tirées au hasard entre 1 et 6. Chaque utilisation du bouton B provoquera un changement de valeur aléatoire du premier dé de la seconde ligne, tant que cette valeur restera différente de celle du dé correspondant dans la première ligne. Lorsque le 1er dé de la 2e ligne aura acquis la valeur de son correspondant, c'est la valeur du 2e dé de la seconde ligne qui sera changée au hasard, et ainsi de suite, jusqu'à ce que les 6 faces du bas soient identiques à celles du haut. Le bouton C provoquera l'effacement complet (pas seulement les points !) de tous les dés affichés.

14

Et pour quelques widgets de plus ...

Les pages qui suivent contiennent des indications et et des exemples complémentaires qui pourront vous être utiles pour le développement de vos projets personnels. Il ne s'agit évidemment pas d'une documentation de référence complète sur *Tkinter*. Pour en savoir plus, vous devrez tôt ou tard consulter des ouvrages spécialisés, comme par exemple *Python and Tkinter programming* de John E. Grayson.

Les « boutons radio »

Les widgets « boutons radio » permettent de proposer à l'utilisateur un ensemble de choix mutuellement exclusifs. On les appelle ainsi par analogie avec les boutons de sélection que l'on trouvait jadis sur les postes de radio. Ces boutons étaient conçus de telle manière qu'un seul à la fois pouvait être enfoncé : tous les autres ressortaient automatiquement.

La caractéristique essentielle de ces widgets est qu'on les utilise toujours par groupes. Tous les boutons radio faisant partie d'un même groupe sont associés à une seule et même *variable Tkinter*, mais chacun d'entre eux se voit aussi attribuer une valeur particulière.
Lorsque l'utilisateur sélectionne l'un des boutons, la valeur correspondant à ce bouton est affectée à la variable Tkinter commune.

```
 1# from Tkinter import *
 2#
 3# class RadioDemo(Frame):
 4#     """Démo : utilisation de widgets 'boutons radio'"""
 5#     def __init__(self, boss =None):
 6#         """Création d'un champ d'entrée avec 4 boutons radio"""
 7#         Frame.__init__(self)
 8#         self.pack()
 9#         # Champ d'entrée contenant un petit texte :
10#         self.texte = Entry(self, width =30, font ="Arial 14")
11#         self.texte.insert(END, "La programmation, c'est génial")
12#         self.texte.pack(padx =8, pady =8)
13#         # Nom français et nom technique des quatre styles de police :
14#         stylePoliceFr =["Normal", "Gras", "Italique", "Gras/Italique"]
15#         stylePoliceTk =["normal", "bold", "italic"  , "bold italic"]
16#         # Le style actuel est mémorisé dans un 'objet-variable' Tkinter ;
17#         self.choixPolice = StringVar()
18#         self.choixPolice.set(stylePoliceTk[0])
19#         # Création des quatre 'boutons radio' :
20#         for n in range(4):
21#             bout = Radiobutton(self,
22#                                text = stylePoliceFr[n],
23#                                variable = self.choixPolice,
```

```
24#                                 value = stylePoliceTk[n],
25#                                 command = self.changePolice)
26#             bout.pack(side =LEFT, padx =5)
27#
28#     def changePolice(self):
29#         """Remplacement du style de la police actuelle"""
30#         police = "Arial 15 " + self.choixPolice.get()
31#         self.texte.configure(font =police)
32#
33# if __name__ == '__main__':
34#     RadioDemo().mainloop()
```

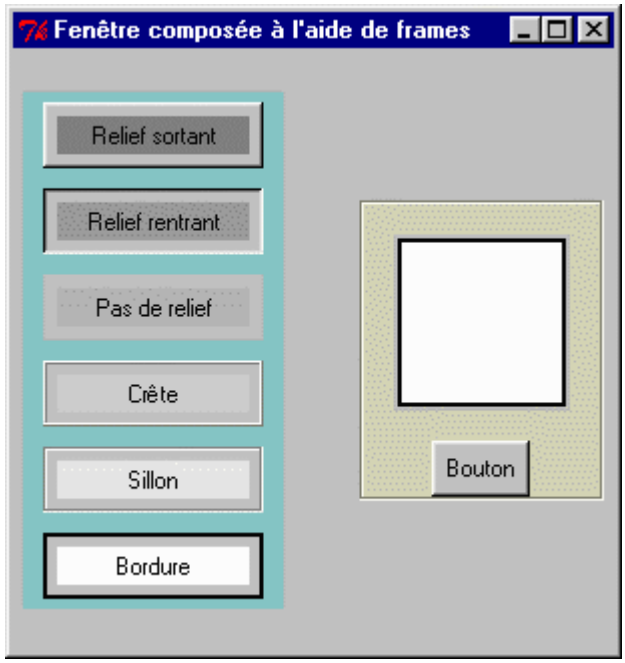
Commentaires :

- Ligne 3 : Cette fois encore, nous préférons construire notre petite application comme une classe dérivée de la classe **Frame()**, ce qui nous permettrait éventuellement de l'intégrer sans difficulté dans une application plus importante.
- Ligne 8 : En général, on applique les méthodes de positionnement des widgets (**pack()**, **grid()**, ou **place()**) après instanciation de ceux-ci, ce qui permet de choisir librement leur disposition à l'intérieur des fenêtres maîtresses. Comme nous le montrons ici, il est cependant tout à fait possible de déjà prévoir ce positionnement dans le constructeur du widget.
- Ligne 11 : Les widgets de la classe **Entry** disposent de plusieurs méthodes pour accéder à la chaîne de caractères affichée. La méthode **get()** permet de récupérer la chaîne entière (il s'agira en fait d'une chaîne unicode). La méthode **delete()** permet d'en effacer tout ou partie (cfr. projet « Code des couleurs », page 183). La méthode **insert()** permet d'insérer de nouveaux caractères à un emplacement quelconque (c'est-à-dire au début, à la fin, ou même à l'intérieur d'une chaîne préexistante éventuelle). Cette méthode s'utilise donc avec deux arguments, le premier indiquant l'emplacement de l'insertion (utilisez **0** pour insérer au début, **END** pour insérer à la fin, ou encore un indice numérique quelconque pour désigner un caractère dans la chaîne).
- Lignes 14-15 : Plutôt que de les instancier dans des instructions séparées, nous préférons créer nos quatre boutons à l'aide d'une boucle. Les options spécifiques à chacun d'eux sont d'abord préparées dans les deux listes **stylePoliceFr** et **stylePoliceTk** : la première contient les petits textes qui devront s'afficher en regard de chaque bouton, et la seconde les valeurs qui devront leur être associées.
- Lignes 17-18 : Comme expliqué à la page précédente, les quatre boutons forment un groupe autour d'une variable commune. Cette variable prendra la valeur associée au bouton radio que l'utilisateur décidera de choisir. Nous ne pouvons cependant pas utiliser une variable ordinaire pour remplir ce rôle, parce que les attributs internes des objets Tkinter ne sont accessibles qu'au travers de méthodes spécifiques. Une fois de plus, nous utilisons donc ici un *objet-variable* Tkinter, de type 'chaîne de caractères', que nous instancions à partir de la classe **StringVar()**, et auquel nous donnons une valeur par défaut à la ligne 18.
- Lignes 20 à 26 : Instanciation des quatre boutons radio. Chacun d'entre eux se voit attribuer une étiquette et une valeur différentes, mais tous sont associés à la même variable Tkinter commune (**self.choixPolice**). Tous invoquent également la même méthode **self.changePolice()**, chaque fois que l'utilisateur effectue un clic de souris sur l'un ou l'autre.
- Lignes 28 à 31 : Le changement de police s'obtient par re-configuration de l'option **font** du widget **Entry**. Cette option attend un tuple contenant le nom de la police, sa taille, et éventuellement son style. Si le nom de la police ne contient pas d'espaces, le tuple peut aussi être remplacé par une chaîne de caractères. Exemples :
  `('Arial', 12, 'italic')`

```
('Helvetica', 10)
('Times New Roman', 12, 'bold italic')
"Verdana 14 bold"
"President 18 italic"
```

Voir également les exemples de la page 229.

Utilisation de cadres pour la composition d'une fenêtre



Vous avez déjà abondamment utilisé la classe de widgets **Frame()** (« cadre », en français), notamment pour créer de nouveaux widgets complexes par dérivation.

Le petit script ci-dessous vous montre l'utilité de cette même classe pour regrouper des ensembles de widgets et les disposer d'une manière déterminée dans une fenêtre. Il vous démontre également l'utilisation de certaines options décoratives (bordures, relief, etc.).

Pour composer la fenêtre ci-contre, nous avons utilisé deux cadres **f1** et **f2**, de manière à réaliser deux groupes de widgets bien distincts, l'un à gauche et l'autre à droite. Nous avons coloré ces deux cadres pour bien les mettre en évidence, mais ce n'est évidemment pas indispensable.

Le cadre **f1** contient lui-même 6 autres cadres, qui contiennent chacun un widget de la classe **Label()**. Le cadre **f2** contient un widget **Canvas()** et un widget **Button()**. Les couleurs et garnitures sont de simples options.

```
 1#  from Tkinter import *
 2#
 3#  fen = Tk()
 4#  fen.title("Fenêtre composée à l'aide de frames")
 5#  fen.geometry("300x300")
 6#
 7#  f1 = Frame(fen, bg = '#80c0c0')
 8#  f1.pack(side =LEFT, padx =5)
 9#
10#  fint = [0]*6
11#  for (n, col, rel, txt) in [(0, 'grey50', RAISED, 'Relief sortant'),
12#                             (1, 'grey60', SUNKEN, 'Relief rentrant'),
13#                             (2, 'grey70', FLAT, 'Pas de relief'),
14#                             (3, 'grey80', RIDGE, 'Crête'),
15#                             (4, 'grey90', GROOVE, 'Sillon'),
16#                             (5, 'grey100', SOLID, 'Bordure')]:
17#      fint[n] = Frame(f1, bd =2, relief =rel)
18#      e = Label(fint[n], text =txt, width =15, bg =col)
19#      e.pack(side =LEFT, padx =5, pady =5)
20#      fint[n].pack(side =TOP, padx =10, pady =5)
21#
22#  f2 = Frame(fen, bg ='#d0d0b0', bd =2, relief =GROOVE)
23#  f2.pack(side =RIGHT, padx =5)
24#
25#  can = Canvas(f2, width =80, height =80, bg ='white', bd =2, relief =SOLID)
26#  can.pack(padx =15, pady =15)
```

```
27#  bou =Button(f2, text='Bouton')
28#  bou.pack()
29#
30#  fen.mainloop()
```

Commentaires :

- Lignes 3 à 5 : Afin de simplifier au maximum la démonstration, nous ne programmons pas cet exemple comme une nouvelle classe. Remarquez à la ligne 5 l'utilité de la méthode **geometry()** pour fixer les dimensions de la fenêtre principale.
- Ligne 7 : Instanciation du cadre de gauche. La couleur de fond (une variété de bleu cyan) est déterminée par l'argument **bg** (*background*). Cette chaîne de caractères contient en notation hexadécimale la description des trois composantes rouge, verte et bleue de la teinte que l'on souhaite obtenir : Après le caractère **#** signalant que ce qui suit est une valeur numérique hexadécimale, on trouve trois groupes de deux symboles alphanumériques. Chacun de ces groupes représente un nombre compris entre 1 et 255. Ainsi, 80 correspond à 128, et c0 correspond à 192 en notation décimale. Dans notre exemple, les composantes rouge, verte et bleue de la teinte à représenter valent donc respectivement 128, 192 & 192.
  En application de cette technique descriptive, le noir serait obtenu avec **`#000000`**, le blanc avec **`#ffffff`**, le rouge pur avec **`#ff0000`**, un bleu sombre avec **`#000050`**, etc.
- Ligne 8 : Puisque nous lui appliquons la méthode **pack()**, le cadre sera automatiquement dimensionné par son contenu. L'option **`side =LEFT`** le positionnera à gauche dans sa fenêtre maîtresse. L'option **`padx =5`** ménagera un espace de 5 pixels à sa gauche et à sa droite (nous pouvons traduire « padx » par « espacement horizontal »).
- Ligne 10 : Dans le cadre **f1** que nous venons de préparer, nous avons l'intention de regrouper 6 autres cadres similaires contenant chacun une étiquette. Le code correspondant sera plus simple et plus efficient si nous instancions ces widgets dans une liste plutôt que dans des variables indépendantes. Nous préparons donc cette liste avec 6 éléments que nous remplacerons plus loin.
- Lignes 11 à 16 : Pour construire nos 6 cadres similaires, nous allons parcourir une liste de 6 tuples contenant les caractéristiques particulières de chaque cadre. Chacun de ces tuples est constitué de 4 éléments : un indice, une constante Tkinter définissant un type de relief, et deux chaînes de caractères décrivant respectivement la couleur et le texte de l'étiquette.
  La boucle **for** effectue 6 itérations pour parcourir les 6 éléments de la liste. A chaque itération, le contenu d'un des tuples est affecté aux variables **n**, **col**, **rel** et **txt** (et ensuite les instructions des lignes 17 à 20 sont exécutées).

> *Le parcours d'une liste de tuples à l'aide d'une boucle for constitue une construction particulièrement compacte, qui permet de réaliser de nombreuses affectations avec un très petit nombre d'instructions.*

- Ligne 17 : Les 6 cadres sont instanciés comme des éléments de la liste **fint**. Chacun d'entre eux est agrémenté d'une bordure décorative de 2 pixels de large, avec un certain effet de relief.
- Lignes 18-20 : Les étiquettes ont toutes la même taille, mais leurs textes et leurs couleurs de fond diffèrent. Du fait de l'utilisation de la méthode **pack()**, c'est la dimension des étiquettes qui détermine la taille des petits cadres. Ceux-ci à leur tour déterminent la taille du cadre qui les regroupe (le cadre f1). Les options **padx** et **pady** permettent de réserver un petit espace autour de chaque étiquette, et un autre autour de chaque petit cadre. L'option **`side =TOP`** positionne les 6 petits cadres les uns en dessous des autres dans le cadre conteneur f1.

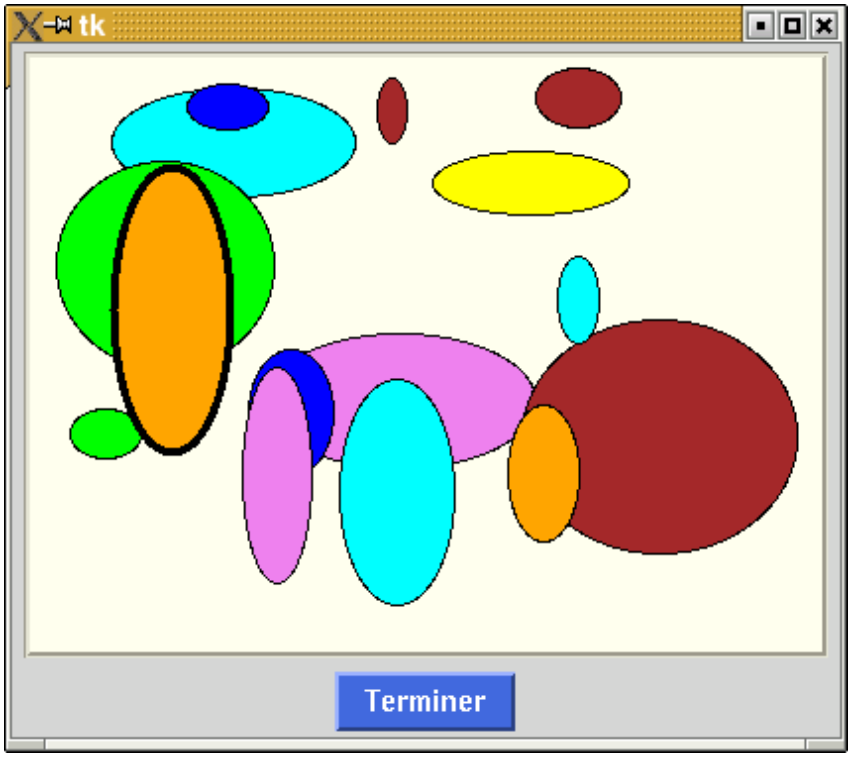
- Lignes 22-23 : Préparation du cadre **f2** (cadre de droite). Sa couleur sera une variété de jaune, et nous l'entourerons d'une bordure décorative ayant l'aspect d'un sillon.
- Lignes 25 à 28 : Le cadre **f2** contiendra un canevas et un bouton. Notez encore une fois l'utilisation des options **padx** et **pady** pour ménager des espaces autour des widgets (Considérez par exemple le cas du bouton, pour lequel cette option n'a pas été utilisée : de ce fait, il entre en contact avec la bordure du cadre qui l'entoure). Comme nous l'avons fait pour les cadres, nous avons placé une bordure autour du canevas. Sachez que d'autres widgets acceptent également ce genre de décoration : boutons, champs d'entrée, etc.

Comment déplacer des dessins à l'aide de la souris

Le widget canevas est l'un des points forts de la bibliothèque graphique Tkinter. Il intègre en effet un grand nombre de dispositifs très efficaces pour manipuler des dessins. Le script ci-après est destiné à vous montrer quelques techniques de base. Si vous voulez en savoir plus, notamment en ce qui concerne la manipulation de dessins composés de plusieurs parties, veuillez consulter l'un ou l'autre ouvrage de référence traitant de Tkinter.

Au démarrage de notre petite application, une série de dessins sont tracés au hasard dans un canevas (il s'agit en l'occurrence de simples ellipses colorées). Vous pouvez déplacer n'importe lequel de ces dessins en le « saisissant » à l'aide de votre souris.

Lorsqu'un dessin est déplacé, il passe à l'avant-plan par rapport aux autres, et sa bordure apparaît plus épaisse pendant toute la durée de sa manipulation.



Pour bien comprendre la technique utilisée, vous devez vous rappeler qu'un logiciel utilisant une interface graphique est un logiciel « piloté par les événements » (revoyez au besoin les explications de la page 89). Dans cette application, nous allons mettre en place un mécanisme qui réagit aux événements : « enfoncement du bouton gauche de la souris », « déplacement de la souris, le bouton gauche restant enfoncé », « relâchement du bouton gauche ».

Ces événements sont générés par le système d'exploitation et pris en charge par l'interface Tkinter. Notre travail de programmation consistera donc simplement à les associer à des gestionnaires différents (fonctions ou méthodes).

```
# Exemple montrant comment faire en sorte que les objets dessinés dans un
# canevas puissent être manipulés à l'aide de la souris

from Tkinter import *
from random import randrange

class Draw(Frame):
    "classe définissant la fenêtre principale du programme"
    def __init__(self):
        Frame.__init__(self)
        # mise en place du canevas - dessin de 15 ellipses colorées :
        self.c = Canvas(self, width =400, height =300, bg ='ivory')
        self.c.pack(padx =5, pady =3)
        for i in range(15):
            # tirage d'une couleur au hasard :
            coul =['brown','red','orange','yellow','green','cyan','blue',
                   'violet', 'purple'][randrange(9)]
            # tracé d'une ellipse avec coordonnées aléatoires :
            x1, y1 = randrange(300), randrange(200)
            x2, y2 = x1 + randrange(10, 150), y1 + randrange(10, 150)
            self.c.create_oval(x1, y1, x2, y2, fill =coul)
        # liaison d'événements <souris> au widget <canevas> :
        self.c.bind("<Button-1>", self.mouseDown)
        self.c.bind("<Button1-Motion>", self.mouseMove)
        self.c.bind("<Button1-ButtonRelease>", self.mouseUp)
        # mise en place d'un bouton de sortie :
        b_fin = Button(self, text ='Terminer', bg ='royal blue', fg ='white',
                       font =('Helvetica', 10, 'bold'), command =self.quit)
        b_fin.pack(pady =2)
        self.pack()

    def mouseDown(self, event):
        "Op. à effectuer quand le bouton gauche de la souris est enfoncé"
        self.currObject =None
        # event.x et event.y contiennent les coordonnées du clic effectué :
        self.x1, self.y1 = event.x, event.y
        # <find_closest> renvoie la référence du dessin le plus proche :
        self.selObject = self.c.find_closest(self.x1, self.y1)
        # modification de l'épaisseur du contour du dessin :
        self.c.itemconfig(self.selObject, width =3)
        # <lift> fait passer le dessin à l'avant-plan :
        self.c.lift(self.selObject)

    def mouseMove(self, event):
        "Op. à effectuer quand la souris se déplace, bouton gauche enfoncé"
        x2, y2 = event.x, event.y
        dx, dy = x2 -self.x1, y2 -self.y1
        if self.selObject:
            self.c.move(self.selObject, dx, dy)
            self.x1, self.y1 = x2, y2

    def mouseUp(self, event):
        "Op. à effectuer quand le bouton gauche de la souris est relâché"
        if self.selObject:
            self.c.itemconfig(self.selObject, width =1)
            self.selObject =None

if __name__ == '__main__':
    Draw().mainloop()
```

Commentaires :

Le script contient essentiellement la définition d'une classe graphique dérivée de **Frame()**.

Comme c'est souvent le cas pour les programmes exploitant les classes d'objets, le corps principal du script se résume à une seule instruction composée, dans laquelle on réalise deux opérations consécutives : instanciation d'un objet de la classe définie précédemment, et activation de sa méthode **mainloop()** (laquelle démarre l'observateur d'événements).

Le constructeur de la classe **Draw()** présente une structure qui doit vous être devenue familière, à savoir : appel au constructeur de la classe parente, puis mise en place de divers widgets.

Dans le widget canevas, nous instancions 15 dessins sans nous préoccuper de conserver leurs références dans des variables. Nous pouvons procéder ainsi parce que Tkinter conserve lui-même une référence interne pour chacun de ces objets (cf. page 104. Si vous travaillez avec d'autres bibliothèques graphiques, vous devrez probablement prévoir une mémorisation de ces références).

Les dessins sont de simples ellipses colorées. Leur couleur est choisie au hasard dans une liste de 9 possibilités, l'indice de la couleur choisie étant déterminé par la fonction **randrange()** importée du module **random**.

Le mécanisme d'interaction est installé ensuite : on associe les trois identificateurs d'événements <Button-1>, <Button1-Motion> et <Button1-ButtonRelease> concernant le widget canevas, aux noms des trois méthodes choisies comme gestionnaires d'événements.

Lorsque l'utilisateur enfonce le bouton gauche de sa souris, la méthode **mouseDown()** est donc activée, et le système d'exploitation lui transmet en argument un objet **event**, dont les attributs **x** et **y** contiennent les coordonnées du curseur souris dans le canevas, déterminées au moment du clic.

Nous mémorisons directement ces coordonnées dans les variables d'instance **self.x1** et **self.x2**, car nous en aurons besoin par ailleurs. Ensuite, nous utilisons la méthode **find_closest()** du widget canevas, qui nous renvoie la référence du dessin le plus proche. (Note : cette méthode bien pratique renvoie toujours une référence, même si le clic de souris n'a pas été effectué à l'intérieur du dessin).

Le reste est facile : la référence du dessin sélectionné est mémorisée dans une variable d'instance, et nous pouvons faire appel à d'autres méthodes du widget canevas pour modifier ses caractéristiques. En l'occurrence, nous utilisons les méthodes **itemconfig()** et **lift()** pour épaissir son contour et le faire passer à l'avant-plan.

Le « transport » du dessin est assuré par la méthode **mouseMove()**, invoquée à chaque fois que la souris se déplace alors que son bouton gauche est resté enfoncé. L'objet **event** contient cette fois encore les coordonnées du curseur souris, au terme de ce déplacement. Nous nous en servons pour calculer les différences entre ces nouvelles coordonnées et les précédentes, afin de pouvoir les transmettre à la méthode **move()** du widget canevas, qui effectuera le transport proprement dit.

Nous ne pouvons cependant faire appel à cette méthode que s'il existe effectivement un objet sélectionné, et il nous faut veiller également à mémoriser les nouvelles coordonnées acquises.

La méthode **mouseUp()** termine le travail. Lorsque le dessin transporté est arrivé à destination, il reste à annuler la sélection et rendre au contour son épaisseur initiale. Ceci ne peut être envisagé que s'il existe effectivement une sélection, bien entendu.

Python Mega Widgets

Les modules **Pmw** constituent une extension intéressante de Tkinter. Entièrement écrits en Python, ils contiennent toute une bibliothèque de widgets composites, construits à partir des classes

de base de Tkinter. Dotés de fonctionnalités très étendues, ces widgets peuvent se révéler fort précieux pour le développement rapide d'applications complexes. Si vous souhaitez les utiliser, sachez cependant que les modules Pmw ne font pas partie de l'installation standard de Python : vous devrez donc toujours vérifier leur présence sur les machines cibles de vos programmes[61].

Il existe un grand nombre de ces méga-widgets. Nous n'en présenterons ici que quelques-uns parmi les plus utiles. Vous pouvez rapidement vous faire une idée plus complète de leurs multiples possibilités, en essayant les scripts de démonstration qui les accompagnent (lancez par exemple le script **all.py** , situé dans le sous-répertoire `.../Pmw/demos`).

« Combo Box »

Les méga-widgets s'utilisent aisément. La petite application ci-après vous montre comment mettre en œuvre un widget de type **ComboBox** (boîte de liste combinée à un champ d'entrée). Nous l'avons configuré de la manière la plus habituelle (avec une boîte de liste déroulante).

Lorsque l'utilisateur de notre petit programme choisit une couleur dans la liste déroulante (il peut aussi entrer un nom de couleur directement dans le champ d'entrée), cette couleur devient automatiquement la couleur de fond pour la fenêtre maîtresse.

Dans cette fenêtre maîtresse, nous avons ajouté un libellé et un bouton, afin de vous montrer comment vous pouvez accéder à la sélection opérée précédemment dans le **ComboBox** lui-même (le bouton provoque l'affichage du nom de la dernière couleur choisie).

```
 1#  from Tkinter import *
 2#  import Pmw
 3#
 4#  def changeCoul(col):
 5#      fen.configure(background = col)
 6#
 7#  def changeLabel():
 8#      lab.configure(text = combo.get())
 9#
10#  couleurs = ('navy', 'royal blue', 'steelblue1', 'cadet blue',
11#              'lawn green', 'forest green', 'dark red',
12#              'grey80','grey60', 'grey40', 'grey20')
13#
14#  fen = Pmw.initialise()
15#  bou = Button(fen, text ="Test", command =changeLabel)
16#  bou.grid(row =1, column =0, padx =8, pady =6)
17#  lab = Label(fen, text ='néant', bg ='ivory')
18#  lab.grid(row =1, column =1, padx =8)
19#
20#  combo = Pmw.ComboBox(fen, labelpos = NW,
21#                       label_text = 'Choisissez la couleur :',
22#                       scrolledlist_items = couleurs,
23#                       listheight = 150,
24#                       selectioncommand = changeCoul)
25#  combo.grid(row =2, columnspan =2, padx =10, pady =10)
26#
27#  fen.mainloop()
```

[61] L'installation de la bibliothèque Pmw est décrite dans les annexes. Notez qu'il existe d'autres bibliothèques d'extension de Tkinter : **Tix**, par exemple, est peut-être plus intéressante désormais (mais elle n'était pas encore disponible à l'époque de la rédaction de ce texte).

Commentaires :

- Lignes 1 & 2 : On commence par importer les composants habituels de Tkinter, ainsi que le module **Pmw**.
- Ligne 14 : Pour créer la fenêtre maîtresse, il faut utiliser de préférence la méthode **Pmw.initialise()**, plutôt que d'instancier directement un objet de la classe **Tk()**. Cette méthode veille en effet à mettre en place tout ce qui est nécessaire afin que les widgets esclaves de cette fenêtre puissent être détruits correctement lorsque la fenêtre elle-même sera détruite. Cette méthode installe également un meilleur gestionnaire des messages d'erreurs.
- Ligne 12 : L'option **labelpos** détermine l'emplacement du libellé qui accompagne le champ d'entrée. Dans notre exemple, nous l'avons placé au-dessus, mais vous pourriez préférer le placer ailleurs, à gauche par exemple (`labelpos = W`). Notez que cette option est indispensable si vous souhaitez un libellé (pas de valeur par défaut).
- Ligne 14 : L'option **selectioncommand** transmet un argument à la fonction invoquée : l'item sélectionné dans la boîte de liste. Vous pourrez également retrouver cette sélection à l'aide de la méthode **get()**, comme nous le faisons à la ligne 8 pour actualiser le libellé.

Remarque concernant l'entrée de caractères accentués

Nous vous avons déjà signalé précédemment que Python est tout à fait capable de prendre en charge les alphabets du monde entier (grec, cyrillique, arabe, japonais, etc.). Il en va de même pour Tkinter. En tant que francophone, vous souhaiterez certainement que les utilisateurs de vos scripts puissent entrer des caractères accentués dans les widgets **Entry**, **Text** et leurs dérivés (**ComboBox**, **ScrolledText**).

Veuillez donc prendre bonne note que lorsque vous entrez dans l'un de ces widgets une chaîne contenant un ou plusieurs caractères non-ASCII (tel qu'une lettre accentuée, par exemple), Tkinter encode cette chaîne comme un objet *unicode*. Suivant que votre ordinateur utilise l'encodage *Latin-1* par défaut (ce qui est encore souvent le cas sous *Windows*), ou bien *Utf-8* (cas de tous les OS récents prenant en charge les normes internationales), vous devrez peut-être convertir la chaîne en objet *string* pour certaines de vos opérations d'entrée-sortie.

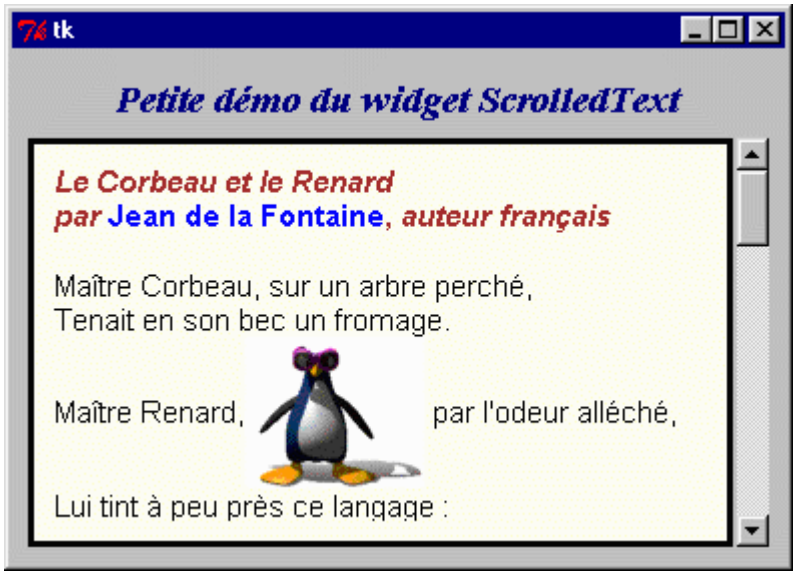
Comme expliqué en détail aux pages 123 et suivantes, cela peut se faire très aisément en utilisant la fonction intégrée **encode()**. Exemple :

```
 1#  from Tkinter import *
 2#
 3#  def imprimer():
 4#      ch1 = e.get()                 # le widget Entry renvoie un objet unicode
 5#      ch2 = ch1.encode('Utf-8')     # conversion unicode -> string
 6#      print ch1, type(ch1)
 7#      print ch2 , type(ch2)
 8#
 9#  f = Tk()
10#  e = Entry(f)
11#  e.pack()
12#  Button(f, text ="afficher", command =imprimer).pack()
13#  f.mainloop()
```

Essayez ce petit script en entrant des chaînes avec caractères accentués dans le champ d'entrée.

Essayez encore, mais en remplaçant 'Utf-8' par 'Latin-1' à ligne 5. Concluez.

« Scrolled Text »



Ce méga-widget étend les possibilités du widget **Text** standard, en lui associant un cadre, un libellé (titre) et des barres de défilement.

Comme le démontrera le petit script ci-dessous, il sert fondamentalement à afficher des textes, mais ceux-ci peuvent être mis en forme et intégrer des images.

Vous pouvez également rendre « cliquables » les éléments affichés (textes ou images), et vous en servir pour déclencher toutes sortes de mécanismes.

Dans l'application qui génère la figure ci-dessus, par exemple, le fait de cliquer sur le nom « Jean de la Fontaine » provoque le défilement automatique du texte (*scrolling*), jusqu'à ce qu'une rubrique décrivant cet auteur devienne visible dans le widget (Voir page suivante le script correspondant).

D'autres fonctionnalités sont présentes, mais nous ne présenterons ici que les plus fondamentales. Veuillez donc consulter les démos et exemples accompagnant Pmw pour en savoir davantage.

Gestion du texte affiché : Vous pouvez accéder à n'importe quelle portion du texte pris en charge par le widget grâce à deux concepts complémentaires, les index et les balises :

- Chaque caractère du texte affiché est référencé par un *index*, lequel doit être une chaîne de caractères contenant deux valeurs numériques reliées par un point (ex : `"5.2"`). Ces deux valeurs indiquent respectivement le numéro de ligne et le numéro de colonne où se situe le caractère.
- N'importe quelle portion du texte peut être associée à une ou plusieurs balise(s), dont vous choisissez librement le nom et les propriétés. Celles-ci vous permettent de définir la police, les couleurs d'avant- et d'arrière-plan, les événements associés, etc.

Note : Pour la bonne compréhension du script ci-dessous, veuillez considérer que le texte de la fable traitée doit être accessible, dans un fichier nommé **CorbRenard.txt**.

```
 1#  from Tkinter import *
 2#  import Pmw
 3#
 4#  def action(event=None):
 5#      """défilement du texte jusqu'à la balise <cible>"""
 6#      index = st.tag_nextrange('cible', '0.0', END)
 7#      st.see(index[0])
 8#
 9#  # Instanciation d'une fenêtre contenant un widget ScrolledText :
10#  fen = Pmw.initialise()
11#  st = Pmw.ScrolledText(fen,
12#                        labelpos =N,
13#                        label_text ="Petite démo du widget ScrolledText",
14#                        label_font ='Times 14 bold italic',
15#                        label_fg = 'navy', label_pady =5,
16#                        text_font='Helvetica 11 normal', text_bg ='ivory',
17#                        text_padx =10, text_pady =10, text_wrap ='none',
```

```
18#                       borderframe =1,
19#                       borderframe_borderwidth =3,
20#                       borderframe_relief =SOLID,
21#                       usehullsize =1,
22#                       hull_width =370, hull_height =240)
23#  st.pack(expand =YES, fill =BOTH, padx =8, pady =8)
24#
25#  # Définition de balises, liaison d'un gestionnaire d'événement au clic souris :
26#  st.tag_configure('titre', foreground ='brown', font ='Helvetica 11 bold italic')
27#  st.tag_configure('lien', foreground ='blue', font ='Helvetica 11 bold')
28#  st.tag_configure('cible', foreground ='forest green', font ='Times 11 bold')
29#  st.tag_bind('lien', '<Button-1>', action)
30#
31#  titre ="""Le Corbeau et le Renard
32#  par Jean de la Fontaine, auteur français
33#  \n"""
34#  auteur ="""
35#  Jean de la Fontaine
36#  écrivain français (1621-1695)
37#  célèbre pour ses Contes en vers,
38#  et surtout ses Fables, publiées
39#  de 1668 à 1694."""
40#
41#  # Remplissage du widget Text (2 techniques) :
42#  st.importfile('CorbRenard.txt')
43#  st.insert('0.0', titre, 'titre')
44#  st.insert(END, auteur, 'cible')
45#  # Insertion d'une image :
46#  photo =PhotoImage(file= 'Penguin.gif')
47#  st.image_create('6.14', image =photo)
48#  # Mise en oeuvre dynamique d'une balise :
49#  st.tag_add('lien', '2.4', '2.23')
50#
51#  fen.mainloop()
```

Commentaires :

- Lignes 4-7 : Cette fonction est un gestionnaire d'événement, qui est appelé lorsque l'utilisateur effectue un clic de souris sur le nom de l'auteur (cfr. lignes 27 & 29). A la ligne 6, on utilise la méthode **tag_nextrange()** du widget pour trouver les index de la portion de texte associée à la balise « cible ». La recherche de ces index est limitée au domaine défini par les 2e & 3e arguments (dans notre exemple, on recherche du début à la fin du texte entier). La méthode **tag_nextrange()** renvoie une liste de deux index (ceux des premier et dernier caractères de la portion de texte associée à la balise « cible »). A la ligne 7, nous nous servons d'un seul de ces index (le premier) pour activer la méthode **see()**. Celle-ci provoque un défilement automatique du texte (*scrolling*), de telle manière que le caractère correspondant à l'index transmis devienne visible dans le widget (avec en général un certain nombre des caractères qui suivent).
- Lignes 9 à 23 : Construction classique d'une fenêtre destinée à afficher un seul widget. Dans le code d'instanciation du widget, nous avons inclus un certain nombre d'options destinées à vous montrer une petite partie des nombreuses possibilités de configuration.
- Ligne 12 : L'option **labelpos** détermine l'emplacement du libellé (titre) par rapport à la fenêtre de texte. Les valeurs acceptées s'inspirent des lettres utilisées pour désigner les points cardinaux (**N**, **S**, **E**, **W**, ou encore **NE**, **NW**, **SE**, **SW**). Si vous ne souhaitez pas afficher un libellé, il vous suffit tout simplement de ne pas utiliser cette option.
- Lignes 13 à 15 : Le libellé n'est rien d'autre qu'un widget **Label** standard, intégré dans le widget composite **ScrolledText**. On peut accéder à toutes ses options de configuration, en utilisant la syntaxe qui est présentée dans ces lignes : on y voit qu'il suffit d'associer le préfixe **label_** au nom de l'option que l'on souhaite activer, pour définir aisément les couleurs d'avant-

et d'arrière-plans, la police, la taille, et même l'espacement à réserver autour du widget (option **pady**).

- Lignes 16-17 : En utilisant une technique similaire à celle qui est décrite ci-dessus pour le libellé, on peut accéder aux options de configuration du widget **Text** intégré dans **ScrolledText**. Il suffit cette fois d'associer aux noms d'option le préfixe **text_**.
- Lignes 18 à 20 : Il est prévu un cadre (un widget **Frame**) autour du widget **Text**. L'option **borderframe = 1** permet de le faire apparaître. On accède ensuite à ses options de configuration d'une manière similaire à celle qui a été décrite ci-dessus pour **label_** et **text_**.
- Lignes 21-22 : Ces options permettent de fixer globalement les dimensions du widget. Une autre possibilité serait de définir plutôt les dimensions de son composant **Text** (par exemple à l'aide d'options telles que **text_width** et **text_height**), mais alors les dimensions globales du widget risqueraient de changer en fonction du contenu (apparition/disparition automatique de barres de défilement). Note : le mot **hull** désigne le contenant global, c.à.d. le méga-widget lui-même.
- Ligne 23 : Les options `expand = YES` et `fill = BOTH` de la méthode **pack()** indiquent que le widget concerné pourra être redimensionné à volonté, dans ses 2 dimensions horizontale et verticale.
- Lignes 26 à 29 : Ces lignes définissent les trois balises **titre**, **lien** et **cible** ainsi que le formatage du texte qui leur sera associé. La ligne 29 précise en outre que le texte associé à la balise **lien** sera « cliquable », avec indication du gestionnaire d'événement correspondant.
- Ligne 42 : Importation de texte à partir d'un fichier. Note : Il est possible de préciser l'endroit exact où devra se faire l'insertion, en fournissant un index comme second argument.
- Lignes 43-44 : Ces instructions insèrent des fragments de texte (respectivement au début et à la fin du texte préexistant), en associant une balise à chacun d'eux.
- Ligne 49 : L'association des balises au texte est dynamique. A tout moment, vous pouvez activer une nouvelle association (comme nous le faisons ici en rattachant la balise « lien » à une portion de texte préexistante). Note : pour « détacher » une balise, utilisez la méthode **tag_delete()**.

« Scrolled Canvas »

Le script ci-après vous montre comment vous pouvez exploiter le méga-widget **ScrolledCanvas**, lequel étend les possibilités du widget **Canvas** standard en lui associant des barres de défilement, un libellé et un cadre. Notre exemple constitue en fait un petit jeu d'adresse, dans lequel l'utilisateur doit réussir à cliquer sur un bouton qui s'esquive sans cesse. (Note : si vous éprouvez vraiment des difficultés pour l'attraper, commencez d'abord par dilater la fenêtre).

Le widget **Canvas** est très versatile : il vous permet de combiner à volonté des dessins, des images bitmap, des fragments de texte, et même d'autres widgets, dans un espace parfaitement extensible. Si vous souhaitez développer l'un ou l'autre jeu graphique, c'est évidemment le widget qu'il vous faut apprendre à maîtriser en priorité.

Comprenez bien cependant que les indications que nous vous fournissons à ce sujet dans les présentes notes sont forcément très incomplètes. Leur objectif est seulement de vous aider à comprendre quelques concepts de base, afin que vous puissiez ensuite consulter les ouvrages de référence spécialisés dans de bonnes conditions.

Notre petite application se présente comme une nouvelle classe **FenPrinc()**, obtenue par dérivation à partir de la classe de méga-widgets **Pmw.ScrolledCanvas()**. Elle contient donc un grand canevas muni de barres de défilement, dans lequel nous commençons par planter un décor constitué de 80 ellipses de couleur dont l'emplacement et les dimensions sont tirés au hasard.

Nous y ajoutons également un petit clin d'œil sous la forme d'une image bitmap, destinée avant tout à vous rappeler comment vous pouvez gérer ce type de ressource.

Nous y installons enfin un véritable widget : un simple bouton, en l'occurrence, mais la technique mise en œuvre pourrait s'appliquer à n'importe quel autre type de widget, y compris un gros widget composite comme ceux que nous avons développés précédemment. Cette grande souplesse dans le développement d'applications complexes est l'un des principaux bénéfices apportés par le mode de programmation « orientée objet ».

Le bouton s'anime dès qu'on l'a enfoncé une première fois. Dans votre analyse du script ci-après, soyez attentifs aux méthodes utilisées pour modifier les propriétés d'un objet existant.

```
 1# from Tkinter import *
 2# import Pmw
 3# from random import randrange
 4#
 5# Pmw.initialise()
 6# coul =['sienna','maroon','brown','pink','tan','wheat','gold','orange','plum',
 7#        'red','khaki','indian red','thistle','firebrick','salmon','coral']
 8#
 9# class FenPrinc(Pmw.ScrolledCanvas):
10#     """Fenêtre principale : canevas extensible avec barres de défilement"""
11#     def __init__(self):
12#         Pmw.ScrolledCanvas.__init__(self,
13#                 usehullsize =1, hull_width =500, hull_height =300,
14#                 canvas_bg ='grey40', canvasmargin =10,
15#                 labelpos =N, label_text ='Attrapez le bouton !',
16#                 borderframe =1,
17#                 borderframe_borderwidth =3)
18#         # Les options ci-dessous doivent être précisées après initialisation :
19#         self.configure(vscrollmode ='dynamic', hscrollmode ='dynamic')
20#         self.pack(padx =5, pady =5, expand =YES, fill =BOTH)
21#
22#         self.can = self.interior()        # accès au composant canevas
23#         # Décor : tracé d'une série d'ellipses aléatoires :
24#         for r in range(80):
25#             x1, y1 = randrange(-800,800), randrange(-800,800)
26#             x2, y2 = x1 + randrange(40,300), y1 + randrange(40,300)
27#             couleur = coul[randrange(0,16)]
28#             self.can.create_oval(x1, y1, x2, y2, fill=couleur, outline='black')
29#         # Ajout d'une petite image GIF :
30#         self.img = PhotoImage(file ='linux2.gif')
31#         self.can.create_image(50, 20, image =self.img)
32#         # Dessin du bouton à attraper :
33#         self.x, self.y = 50, 100
34#         self.bou = Button(self.can, text ="Start", command =self.start)
35#         self.fb = self.can.create_window(self.x, self.y, window =self.bou)
36#         self.resizescrollregion()
37#
38#     def anim(self):
39#         if self.run ==0:
40#             return
41#         self.x += randrange(-60, 61)
42#         self.y += randrange(-60, 61)
43#         self.can.coords(self.fb, self.x, self.y)
44#         self.configure(label_text = 'Cherchez en %s %s' % (self.x, self.y))
45#         self.resizescrollregion()
46#         self.after(250, self.anim)
47#
48#     def stop(self):
49#         self.run =0
50#         self.bou.configure(text ="Restart", command =self.start)
51#
52#     def start(self):
53#         self.bou.configure(text ="Attrapez-moi !", command =self.stop)
54#         self.run =1
55#         self.anim()
56#
57# ##### Main Program ##############
58#
59# if __name__ == '__main__':
60#     FenPrinc().mainloop()
```

Commentaires :

- Ligne 6 : Tous ces noms de couleurs sont acceptés par Tkinter. Vous pourriez bien évidemment les remplacer par des descriptions hexadécimales, comme nous l'avons expliqué page 210.

- Lignes 12 à 17 : Ces options sont très similaires à celles que nous avons décrites plus haut pour le widget **ScrolledText**. Le présent méga-widget intègre un composant **Frame**, un composant **Label**, un composant **Canvas** et deux composants **Scrollbar**. On accède aux options de configuration de ces composants à l'aide d'une syntaxe qui relie le nom du composant et celui de l'option par l'intermédiaire d'un caractère « souligné ».
- Ligne 19 : Ces options définissent le mode d'apparition des barres de défilement. En mode **static**, elles sont toujours présentes. En mode **dynamic**, elles disparaissent si les dimensions du canevas deviennent inférieures à celles de la fenêtre de visualisation.
- Ligne 22 : La méthode **interior()** renvoie la référence du composant **Canvas** intégré dans le méga-widget **ScrolledCanvas**. Les instructions suivantes (lignes 23 à 35) installent ensuite toute une série d'éléments dans ce canevas : des dessins, une image et un bouton.
- Lignes 25 à 27 : La fonction **randrange()** permet de tirer au hasard un nombre entier compris dans un certain intervalle (Veuillez vous référer aux explications de la page 149).
- Ligne 35 : C'est la méthode **create_window()** du widget **Canvas** qui permet d'y insérer n'importe quel autre widget (y compris un widget composite). Le widget à insérer doit cependant avoir été défini lui-même au préalable comme un esclave du canevas ou de sa fenêtre maîtresse.
  La méthode **create_window()** attend trois arguments : les coordonnées **X** et **Y** du point où l'on souhaite insérer le widget, et la référence de ce widget.
- Ligne 36 : La méthode **resizescrollregion()** réajuste la situation des barres de défilement de manière à ce qu'elles soient en accord avec la portion du canevas actuellement affichée.
- Lignes 38 à 46 : Cette méthode est utilisée pour l'animation du bouton. Après avoir repositionné le bouton au hasard à une certaine distance de sa position précédente, elle se ré-appelle elle-même après une pause de 250 millisecondes. Ce bouclage s'effectue sans cesse, aussi longtemps que la variable **self.run** contient une valeur non-nulle.
- Lignes 48 à 55 : Ces deux gestionnaires d'événement sont associés au bouton en alternance. Ils servent évidemment à démarrer et à arrêter l'animation.

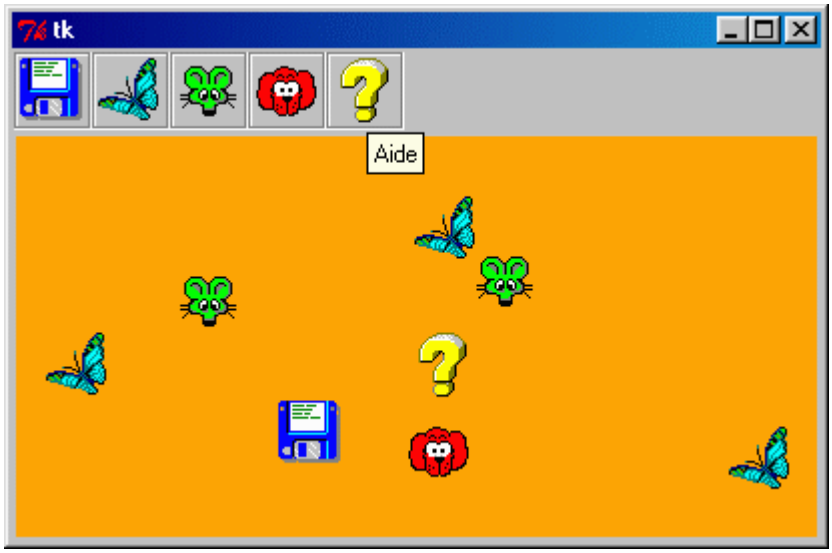
Barres d'outils avec bulles d'aide - expressions lambda

De nombreux programmes comportent une ou plusieurs « barres d'outils » (*toolbar*) constituées de petits boutons sur lesquels sont représentés des pictogrammes (icônes). Cette façon de faire permet de proposer à l'utilisateur un grand nombre de commandes spécialisées, sans que celles-ci n'occupent une place excessive à l'écran (un petit dessin vaut mieux qu'un long discours, dit-on).

La signification de ces pictogrammes n'est cependant pas toujours évidente, surtout pour les utilisateurs néophytes. Il est donc vivement conseillé de compléter les barres d'outils à l'aide d'un système de bulles d'aide (*tool tips*), qui sont des petits messages explicatifs apparaissant automatiquement lorsque la souris survole les boutons concernés.

L'application décrite ci-après comporte une barre d'outils et un canevas. Lorsque l'utilisateur clique sur l'un des boutons de la barre, le pictogramme qu'il porte est recopié dans le canevas, à un emplacement choisi au hasard :

Dans notre exemple, chaque bouton apparaît entouré d'un sillon. Vous pouvez aisément obtenir d'autres aspects en choisissant judicieusement les options **relief** et **bd** (*bordure*) dans l'instruction d'instanciation des boutons. En particulier, vous pouvez choisir `relief = FLAT` et `bd =0` pour obtenir des petits boutons « plats », sans aucun relief.



La mise en place des bulles d'aide est un jeu d'enfant. Il suffit d'instancier un seul objet Pmw.Balloon pour l'ensemble de l'application, puis d'associer un texte à chacun des widgets auxquels on souhaite associer une bulle d'aide, en faisant appel autant de fois que nécessaire à la méthode bind() de cet objet.

```
 1#  from Tkinter import *
 2#  import Pmw
 3#  from random import randrange
 4#
 5#  # noms des fichiers contenant les icônes (format GIF):
 6#  images =('floppy_2','papi2','pion_1','pion_2','help_4')
 7#  textes =('sauvegarde','papillon','joueur 1','joueur 2','Aide')
 8#
 9#  class Application(Frame):
10#      def __init__(self):
11#          Frame.__init__(self)
12#          # Création d'un objet <bulle d'aide> (un seul suffit) :
13#          tip = Pmw.Balloon(self)
14#          # Création de la barre d'outils (c'est un simple cadre) :
15#          toolbar = Frame(self, bd =1)
16#          toolbar.pack(expand =YES, fill =X)
17#          # Nombre de boutons à construire :
18#          nBou = len(images)
19#          # Les icônes des boutons doivent être placées dans des variables
20#          # persistantes. Une liste fera l'affaire :
21#          self.photoI =[None]*nBou
22#
23#          for b in range(nBou):
24#              # Création de l'icône (objet PhotoImage Tkinter) :
25#              self.photoI[b] =PhotoImage(file = images[b] +'.gif')
26#
27#              # Création du bouton.:
28#              # On utilise une expression "lambda" pour transmettre
29#              # un argument à la méthode invoquée comme commande :
30#              bou = Button(toolbar, image =self.photoI[b], relief =GROOVE,
31#                           command = lambda arg =b: self.action(arg))
32#              bou.pack(side =LEFT)
33#
34#              # association du bouton avec un texte d'aide (bulle) :
35#              tip.bind(bou, textes[b])
36#
37#          self.ca = Canvas(self, width =400, height =200, bg ='orange')
38#          self.ca.pack()
39#          self.pack()
40#
41#      def action(self, b):
```

```
42#          "l'icône du bouton b est recopiée dans le canevas"
43#          x, y = randrange(25,375), randrange(25,175)
44#          self.ca.create_image(x, y, image =self.photoI[b])
45#
46#  Application().mainloop()
```

Métaprogrammation. Expressions lambda :

Vous savez qu'en règle générale, on associe à chaque bouton une *commande*, laquelle est une méthode ou une fonction particulière qui se charge d'effectuer le travail lorsque le bouton est activé. Or dans l'application présente, tous les boutons doivent faire à peu près la même chose (recopier un dessin dans le canevas), la seule différence entre eux étant le dessin concerné.

Pour simplifier notre code, nous voudrions donc pouvoir associer l'option **command** de tous nos boutons *avec une seule et même méthode* (ce sera la méthode **action()**), mais en lui transmettant à chaque fois la référence du bouton particulier utilisé, de manière à ce que l'action accomplie puisse être différente pour chacun d'eux.

Une difficulté se présente, cependant, parce que l'option **command** du widget Button accepte seulement une *valeur* ou une *expression*, et non une *instruction*. Il est donc permis de lui indiquer la référence d'une fonction, mais pas de l'invoquer véritablement en lui transmettant des arguments éventuels (c'est la raison pour laquelle on indique le nom de cette fonction sans lui adjoindre de parenthèses).

On peut résoudre cette difficulté de deux manières :

- Du fait de son caractère *dynamique*, Python accepte qu'un programme puisse *se modifier lui-même*, par exemple en définissant de nouvelles fonctions au cours de son exécution (c'est le concept de *métaprogrammation*).

  Il est donc possible de définir *à la volée* une fonction qui utilise des paramètres, *en indiquant pour chacun de ceux-ci une valeur par défaut*, et ensuite d'invoquer cette même fonction sans arguments là où ceux-ci ne sont pas autorisés. Puisque la fonction est définie en cours d'exécution, les valeurs par défaut peuvent être les contenus de variables, et le résultat de l'opération est un véritable transfert d'arguments.

  Pour illustrer cette technique, remplacez les lignes 27 à 31 du script par les suivantes :

```
# Création du bouton.:
# On définit à la volée une fonction avec un paramètre, dont
# la valeur par défaut est l'argument à transmettre.
# Cette fonction appelle la méthode qui nécessite un argument :
    def agir(arg = b):
        self.action(arg)

    # La commande associée au bouton appelle la fonction ci-dessus :
    bou = Button(toolbar, image = self.photoI[b],
                 relief = GROOVE, command = agir)
```

- Voilà pour le principe. Mais tout ce qui précède peut être simplifié, en faisant appel à une expression **lambda**. Ce mot réservé Python désigne une *expression* qui renvoie un *objet fonction*, similaire à ceux que vous créez avec l'instruction **def**, mais avec la différence que **lambda** étant une expression et non une instruction, on peut l'utiliser comme interface afin d'invoquer une fonction (avec passage d'arguments) là où ce n'est normalement pas possible. Notez au passage qu'une telle fonction est *anonyme* (elle ne possède pas de nom).

Par exemple, l'expression :

```
lambda ar1=b, ar2=c : bidule(ar1,ar2)
```

renvoie la référence d'une fonction anonyme qui pourra elle-même invoquer la fonction **bidule()** en lui transmettant les arguments **b** et **c** , ceux-ci étant utilisés comme valeurs par défaut dans la définition des paramètres **ar1** et **ar2** de la fonction.

Cette technique utilise finalement le même principe que la précédente, mais elle présente l'avantage d'être plus concise, raison pour laquelle nous l'avons utilisée dans notre script. En revanche, elle est un peu plus difficile à comprendre :

```
command = lambda arg =b: self.action(arg)
```

Dans cette portion d'instruction, la commande associée au bouton se réfère à une fonction anonyme dont le paramètre **arg** possède une valeur par défaut : la valeur de l'argument **b**. Invoquée sans argument par la commande, cette fonction anonyme peut tout de même utiliser son paramètre **arg** (avec la valeur par défaut) pour faire appel à la méthode cible **self.action()**, et l'on obtient ainsi un véritable transfert d'argument vers cette méthode .

Nous ne détaillerons pas davantage ici la question des expressions lambda, car elle déborde du cadre que nous nous sommes fixés pour cet ouvrage d'initiation. Si vous souhaitez en savoir plus, veuillez donc consulter l'un ou l'autre des ouvrages de référence cités dans la bibliographie.

Fenêtres avec menus

Nous allons décrire à présent la construction d'une fenêtre d'application dotée de différents types de menus « déroulants », chacun de ces menus pouvant être « détaché » de l'application principale pour devenir lui-même une petite fenêtre indépendante, comme dans l'illustration ci-dessous.

Cet exercice un peu plus long nous servira également de révision, et nous le réaliserons par étapes, en appliquant une stratégie de programmation que l'on appelle *développement incrémental.*

Comme nous l'avons déjà expliqué précédemment[62], cette méthode consiste à commencer l'écriture d'un programme par une ébauche, qui ne comporte que quelques lignes seulement mais qui est déjà fonctionnelle. On teste alors cette ébauche soigneusement afin d'en éliminer les *bugs* éventuels. Lorsque l'ébauche fonctionne correctement, on y ajoute une fonctionnalité supplémentaire. On teste ce complément jusqu'à ce qu'il donne entière satisfaction, puis on en ajoute un autre, et ainsi de suite...

Cela ne signifie pas que vous pouvez commencer directement à programmer sans avoir au préalable effectué une analyse sérieuse du projet, dont au moins les grandes lignes devront être convenablement décrites dans un *cahier des charges* clairement rédigé.

> *Il reste également impératif de commenter convenablement le code produit, au fur et à mesure de son élaboration. S'efforcer de rédiger de bons commentaires est en effet nécessaire, non seulement pour que votre code soit facile à lire (et donc à maintenir plus tard, par d'autres ou par vous-même), mais aussi pour que vous soyez forcés d'exprimer ce que vous souhaitez vraiment que la machine fasse (Cfr. Erreurs sémantiques, page 17)*

Cahier des charges de l'exercice :

Notre application comportera simplement une barre de menus et un canevas. Les différentes rubriques et options des menus ne serviront qu'à faire apparaître des fragments de texte dans le canevas ou à modifier des détails de décoration, mais ce seront avant tout des exemples variés, destinés à donner un aperçu des nombreuses possibilités offertes par ce type de widget, accessoire indispensable de toute application moderne d'une certaine importance.

Nous souhaitons également que le code produit dans cet exercice soit bien structuré. Pour ce faire, nous ferons usage de deux classes : une classe pour l'application principale, et une autre pour la barre de menus. Nous voulons procéder ainsi afin de bien mettre en évidence la construction d'une application type incorporant plusieurs classes d'objets interactifs.

---

62 Voir page 18 : Recherche des erreurs et expérimentation

Première ébauche du programme :

Lorsque l'on construit l'ébauche d'un programme, il faut tâcher d'y faire apparaître le plus tôt possible la structure d'ensemble, avec les relations entre les principaux blocs qui constitueront l'application définitive. C'est ce que nous nous sommes efforcés de faire dans l'exemple ci-dessous :

```
 1# from Tkinter import *
 2#
 3# class MenuBar(Frame):
 4#     """Barre de menus déroulants"""
 5#     def __init__(self, boss =None):
 6#         Frame.__init__(self, borderwidth =2)
 7#
 8#         ##### Menu <Fichier> #####
 9#         fileMenu = Menubutton(self, text ='Fichier')
10#         fileMenu.pack(side =LEFT)
11#         # Partie "déroulante" :
12#         me1 = Menu(fileMenu)
13#         me1.add_command(label ='Effacer', underline =0,
14#                         command = boss.effacer)
15#         me1.add_command(label ='Terminer', underline =0,
16#                         command = boss.quit)
17#         # Intégration du menu :
18#         fileMenu.configure(menu = me1)
19#
20# class Application(Frame):
21#     """Application principale"""
22#     def __init__(self, boss =None):
23#         Frame.__init__(self)
24#         self.master.title('Fenêtre avec menus')
25#         mBar = MenuBar(self)
26#         mBar.pack()
27#         self.can = Canvas(self, bg='light grey', height=190,
28#                           width=250, borderwidth =2)
29#         self.can.pack()
30#         self.pack()
31#
32#     def effacer(self):
33#         self.can.delete(ALL)
34#
35# if __name__ == '__main__':
36#     app = Application()
37#     app.mainloop()
```

Veuillez donc encoder ces lignes et en tester l'exécution. Vous devriez obtenir une fenêtre avec un canevas gris clair surmonté d'une barre de menus. A ce stade, la barre de menus ne comporte encore que la seule rubrique « Fichier ».

Cliquez sur la rubrique « Fichier » pour faire apparaître le menu correspondant : l'option « Effacer » n'est pas encore fonctionnelle (elle servira plus loin à effacer le contenu du canevas), mais l'option « Terminer » devrait déjà vous permettre de fermer proprement l'application.

Comme tous les menus gérés par Tkinter, le menu que vous avez créé peut être converti en menu « flottant » : il suffit de cliquer sur la ligne pointillée apparaissant en-tête de menu. Vous obtenez ainsi une petite fenêtre satellite, que vous pouvez alors positionner où bon vous semble sur le bureau.

Analyse du script :

La structure de ce petit programme devrait vous apparaître familière : afin que les classes définies dans ce script puissent éventuellement être (ré)utilisées dans d'autres projets par importation, comme nous l'avons déjà expliqué précédemment[63], le corps principal du programme (lignes 35 à 37) comporte l'instruction désormais classique : `if __name__ == '__main__':`

Les deux instructions qui suivent consistent seulement à instancier un objet **app** et à faire fonctionner sa méthode **mainloop()**. Comme vous le savez certainement, nous aurions pu également condenser ces deux instructions en une seule.

L'essentiel du du programme se trouve cependant dans les définitions de classes qui précèdent :

La classe **MenuBar()** contient la description de la barre de menus. Dans l'état présent du script, elle se résume à une ébauche de constructeur.

- Ligne 5 : Le paramètre **boss** réceptionne la référence de la fenêtre maîtresse du widget au moment de son instanciation. Cette référence va nous permettre d'invoquer les méthodes associées à cette fenêtre maîtresse, aux lignes 14 & 16.
- Ligne 6 : Activation obligatoire du constructeur de la classe parente.
- Ligne 9 : Instanciation d'un widget de la classe **Menubutton()**, défini comme un « esclave » de **self** (c'est-à-dire l'objet composite « barre de menus » dont nous sommes occupés à définir la classe). Comme l'indique son nom, ce type de widget se comporte un peu comme un bouton : une action se produit lorsque l'on clique dessus.
- Ligne 12 : Afin que cette action consiste en l'apparition véritable d'un menu, il reste encore à définir celui-ci : ce sera encore un nouveau widget, de la classe **Menu()** cette fois, défini lui-même comme un « esclave » du widget Menubutton instancié à la ligne 9.
- Lignes 13 à 16 : On peut appliquer aux widgets de la classe **Menu()** un certain nombre de méthodes spécifiques, chacune d'elles acceptant de nombreuses options. Nous utilisons ici la méthode **add_command()** pour installer dans le menu les deux items « Effacer » et « Terminer ». Nous y intégrons tout de suite l'option **underline**, qui sert à définir un raccourci clavier : cette option indique en effet lequel des caractères de l'item doit apparaître souligné à l'écran. L'utilisateur sait alors qu'il lui suffit de frapper ce caractère au clavier pour que l'action correspondant à cet item soit activée (comme s'il avait cliqué dessus à l'aide de la souris).
  L'action à déclencher lorsque l'utilisateur sélectionne l'item est désignée par l'option **command**. Dans notre script, les commandes invoquées sont toutes les deux des méthodes de la fenêtre maîtresse, dont la référence aura été transmise au présent widget au moment de son instanciation par l'intermédiaire du paramètre **boss**. La méthode **effacer()**, que nous définissons nous-même plus loin, servira à vider le canevas. La méthode prédéfinie **quit()** provoque la sortie de la boucle **mainloop()** et donc l'arrêt du réceptionnaire d'événements associé à la fenêtre d'application.
- Ligne 18 : Lorsque les items du menu ont été définis, il reste encore à reconfigurer le widget maître **Menubutton** de manière à ce que son option « menu » désigne effectivement le **Menu** que nous venons de construire. En effet, nous ne pouvions pas déjà préciser cette option lors de la définition initiale du widget **Menubutton**, puisqu'à ce stade le **Menu** n'existait pas encore. Nous ne pouvions pas non plus définir le widget **Menu** en premier lieu, puisque celui-ci doit être défini comme un « esclave » du widget **Menubutton.** Il faut donc bien procéder en trois étapes comme nous l'avons fait, en faisant appel à la méthode **configure()**. (Cette mé-

[63] Voir page 180: Modules contenant des bibliothèques de classes

thode peut être appliquée à n'importe quel widget préexistant pour en modifier l'une ou l'autre option).

La classe **Application()** contient la description de la fenêtre principale du programme ainsi que les méthodes gestionnaires d'événements qui lui sont associées.

- Ligne 20 : Nous préférons faire dériver notre application de la classe **Frame()**, qui présente de nombreuses options, plutôt que de la classe primordiale **Tk()**. De cette manière, l'application toute entière est encapsulée dans un widget, lequel pourra éventuellement être intégré par la suite dans une application plus importante. Rappelons que de toute manière, Tkinter instanciera automatiquement une fenêtre maîtresse de type **Tk()** pour contenir cette **Frame**.
- Lignes 23-24 : Après l'indispensable activation du constructeur de la classe parente, nous utilisons l'attribut **master** que Tkinter associe automatiquement à chaque widget, pour référencer la fenêtre principale de l'application (la fenêtre maîtresse dont nous venons de parler au paragraphe précédent) et en redéfinir le bandeau-titre.
- Lignes 25 à 29 : Instanciation de deux widgets esclaves pour notre **Frame** principale.
  La « barre de menus » est évidemment le widget défini dans l'autre classe.
- Ligne 30 : Comme n'importe quel autre widget, notre **Frame** principale doit être mise en place.
- Lignes 32-33 : La méthode servant à effacer le canevas est définie dans la classe présente (puisque l'objet canevas en fait partie), mais elle est invoquée par l'option **command** d'un widget esclave défini dans l'autre classe. Comme nous l'avons expliqué plus haut, ce widget esclave reçoit la référence de son widget maître par l'intermédiaire du paramètre **boss**.
  Toutes ces références sont hiérarchisées à l'aide de la qualification des noms par points.

Ajout de la rubrique « Musiciens »

Continuez le développement de ce petit programme, en ajoutant les lignes suivantes dans le constructeur de la classe **MenuBar()** (après la ligne 18) :

```
        ##### Menu <Musiciens> #####
        self.musi = Menubutton(self, text ='Musiciens')
        self.musi.pack(side =LEFT, padx ='3')
        # Partie "déroulante" du menu <Musiciens> :
        me1 = Menu(self.musi)
        me1.add_command(label ='17e siècle', underline =1,
                        foreground ='red', background ='yellow',
                        font =('Comic Sans MS', 11),
                        command = boss.showMusi17)
        me1.add_command(label ='18e siècle', underline =1,
                        foreground='royal blue', background ='white',
                        font =('Comic Sans MS', 11, 'bold'),
                        command = boss.showMusi18)
        # Intégration du menu :
        self.musi.configure(menu = me1)
```

... ainsi que les définitions de méthodes suivantes à la classe **Application()** (après la ligne 33) :

```
    def showMusi17(self):
        self.can.create_text(10, 10, anchor =NW, text ='H. Purcell',
                             font=('Times', 20, 'bold'), fill ='yellow')

    def showMusi18(self):
        self.can.create_text(245, 40, anchor =NE, text ="W. A. Mozart",
                             font =('Times', 20, 'italic'), fill ='dark green')
```

Lorsque vous y aurez ajouté toutes ces lignes, sauvegardez le script et exécutez-le.

Votre barre de menus comporte à présent une rubrique supplémentaire : la rubrique « Musiciens ».

Le menu correspondant propose deux items qui sont affichés avec des couleurs et des polices personnalisées. Vous pourrez vous inspirer de ces techniques décoratives pour vos projets personnels. A utiliser avec modération !

Les commandes que nous avons associées à ces items sont évidemment simplifiées afin de ne pas alourdir l'exercice : elles provoquent l'affichage de petits textes sur le canevas.

Analyse du script

Les seules nouveautés introduites dans ces lignes concernent l'utilisation de polices de caractères bien déterminées (option **font**), ainsi que de couleurs pour l'avant-plan (option **foreground**) et le fond (option **background**) des textes affichés.

Veuillez noter encore une fois l'utilisation de l'option **underline** pour désigner les caractères correspondant à des raccourcis claviers (en n'oubliant pas que la numérotation des caractères d'une chaîne commence à partir de zéro), et surtout que l'option **command** de ces widgets accède aux méthodes de l'autre classe, par l'intermédiaire de la référence mémorisée dans l'attribut **boss**.

La méthode **create_text()** du canevas doit être utilisée avec deux arguments numériques, qui sont les coordonnées X et Y d'un point dans le canevas. Le texte transmis sera positionné par rapport à ce point, en fonction de la valeur choisie pour l'option **anchor** : Celle-ci détermine comment le fragment de texte doit être « ancré » au point choisi dans le canevas, par son centre, par son coin supérieur gauche, etc., en fonction d'une syntaxe qui utilise l'analogie des points cardinaux géographiques (**NW** = angle supérieur gauche, **SE** = angle inférieur droit, **CENTER** = centre, etc.)

Ajout de la rubrique « Peintres » :

Cette nouvelle rubrique est construite d'une manière assez semblable à la précédente, mais nous lui avons ajouté une fonctionnalité supplémentaire : des menus « en cascade ». Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe **MenuBar()** :

```
        ##### Menu <Peintres> #####
        self.pein = Menubutton(self, text ='Peintres')
        self.pein.pack(side =LEFT, padx='3')
        # Partie "déroulante" :
        me1 = Menu(self.pein)
        me1.add_command(label ='classiques', state=DISABLED)
        me1.add_command(label ='romantiques', underline =0,
                        command = boss.showRomanti)
        # Sous-menu pour les peintres impressionistes :
        me2 = Menu(me1)
        me2.add_command(label ='Claude Monet', underline =7,
```

```
                    command = boss.tabMonet)
        me2.add_command(label ='Auguste Renoir', underline =8,
                    command = boss.tabRenoir)
        me2.add_command(label ='Edgar Degas', underline =6,
                    command = boss.tabDegas)
        # Intégration du sous-menu :
        me1.add_cascade(label ='impressionistes', underline=0, menu =me2)
        # Intégration du menu :
        self.pein.configure(menu =me1)
```

... et les définitions suivantes dans la classe **Application()** :

```
    def showRomanti(self):
        self.can.create_text(245, 70, anchor =NE, text = "E. Delacroix",
                    font =('Times', 20, 'bold italic'), fill ='blue')

    def tabMonet(self):
        self.can.create_text(10, 100, anchor =NW, text = 'Nymphéas à Giverny',
                    font =('Technical', 20), fill ='red')

    def tabRenoir(self):
        self.can.create_text(10, 130, anchor =NW,
                    text = 'Le moulin de la galette',
                    font =('Dom Casual BT', 20), fill ='maroon')

    def tabDegas(self):
        self.can.create_text(10, 160, anchor =NW, text = 'Danseuses au repos',
                    font =('President', 20), fill ='purple')
```

Analyse du script :

Vous pouvez réaliser aisément des menus en cascade, en enchaînant des sous-menus les uns aux autres jusqu'à un niveau quelconque (il vous est cependant déconseillé d'aller au-delà de 5 niveaux successifs : vos utilisateurs s'y perdraient).

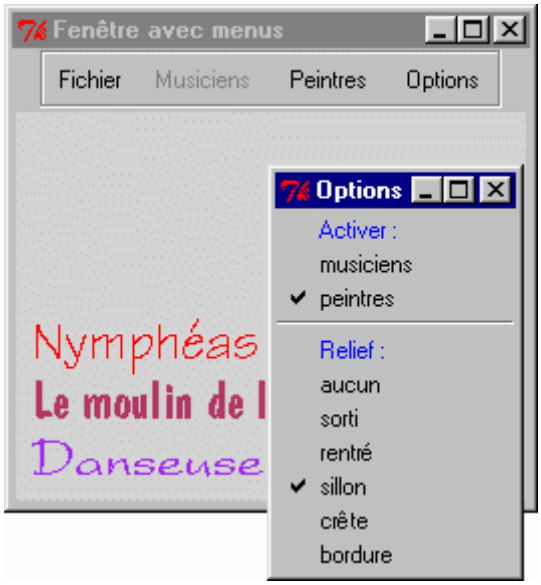
Un sous-menu est défini comme un menu « esclave » du menu de niveau précédent (dans notre exemple, me2 est défini comme un menu « esclave » de **me1**). L'intégration est assurée ensuite à l'aide de la méthode **add_cascade()**.

L'un des items est désactivé (option **state = DISABLED**). L'exemple suivant vous montrera comment vous pouvez activer ou désactiver à volonté des items, par programme.

Ajout de la rubrique « Options » :

La définition de cette rubrique est un peu plus compliquée, parce que nous allons y intégrer l'utilisation de variables internes à Tkinter.

Les fonctionnalités de ce menu sont cependant beaucoup plus élaborées : les options ajoutées permettent en effet d'activer ou de désactiver à volonté les rubriques « Musiciens » et « Peintres », et vous pouvez également modifier à volonté l'aspect de la barre de menus elle-même.



Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe **MenuBar()** :

```
        ##### Menu <Options> #####
        optMenu = Menubutton(self, text ='Options')
        optMenu.pack(side =LEFT, padx ='3')
        # Variables Tkinter :
        self.relief = IntVar()
        self.actPein = IntVar()
        self.actMusi = IntVar()
        # Partie "déroulante" du menu :
        self.mo = Menu(optMenu)
        self.mo.add_command(label = 'Activer :', foreground ='blue')
        self.mo.add_checkbutton(label ='musiciens',
                    command = self.choixActifs, variable =self.actMusi)
        self.mo.add_checkbutton(label ='peintres',
                    command = self.choixActifs, variable =self.actPein)
        self.mo.add_separator()
        self.mo.add_command(label = 'Relief :', foreground ='blue')
        for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                         (3,'sillon'), (4,'crête'), (5,'bordure')]:
            self.mo.add_radiobutton(label =lab, variable =self.relief,
                                    value =v, command =self.reliefBarre)
        # Intégration du menu :
        optMenu.configure(menu = self.mo)
```

... ainsi que les définitions de méthodes suivantes (toujours dans la classe **MenuBar()**) :

```
    def reliefBarre(self):
        choix = self.relief.get()
        self.configure(relief =[FLAT,RAISED,SUNKEN,GROOVE,RIDGE,SOLID][choix])

    def choixActifs(self):
        p = self.actPein.get()
        m = self.actMusi.get()
        self.pein.configure(state =[DISABLED, NORMAL][p])
        self.musi.configure(state =[DISABLED, NORMAL][m])
```

Menu avec « cases à cocher »

Notre nouveau menu déroulant comporte deux parties. Afin de bien les mettre en évidence, nous avons inséré une ligne de séparation ainsi que deux « faux items » (« Activer : » et « Relief : ») qui servent simplement de titres. Nous faisons apparaître ceux-ci en couleur pour que l'utilisateur ne les confonde pas avec de véritables commandes.

Les items de la première partie sont dotées de « cases à cocher ». Lorsque l'utilisateur effectue un clic de souris sur l'un ou l'autre de ces items, les options correspondantes sont activées ou désactivées, et ces états « actif / inactif » sont affichés sous la forme d'une encoche. Les instructions qui servent à mettre en place ce type de rubrique sont assez explicites. Elles présentent en effet ces items comme des widgets de type **chekbutton** :

```
    self.mo.add_checkbutton(label = 'musiciens', command = choixActifs,
                            variable = mbu.me1.music)
```

Il est important de comprendre ici que ce type de widget comporte nécessairement une variable interne, destinée à mémoriser l'état « actif / inactif » du widget. Comme nous l'avons déjà expliqué plus haut, cette variable ne peut pas être une variable Python ordinaire, parce que les classes de la bibliothèque Tkinter sont écrites dans un autre langage. Et par conséquent, on ne pourra ac-

céder à une telle variable interne qu'à travers un objet-interface, que nous appellerons variable Tkinter pour simplifier[64].

C'est ainsi que dans notre exemple, nous utilisons la classe Tkinter **IntVar()** pour créer des objets équivalents à des variables de type entier.

- Nous instancions donc un de ces objets-variables, que nous mémorisons comme attribut d'instance : `self.actMusi =IntVar()`
  Après cette affectation, l'objet référencé dans **self.actMusi** contient désormais l'équivalent d'une variable de type entier, dans un format spécifique à Tkinter.
- Il faut ensuite associer l'option **variable** de l'objet **checkbutton** à la variable Tkinter ainsi définie :
  `self.mo.add_checkbutton(label ='musiciens', variable =self.actMusi)`
- Il est nécessaire de procéder ainsi en deux étapes, parce que Tkinter ne peut pas directement assigner des valeurs aux variables Python. Pour une raison similaire, il n'est pas possible à Python de lire directement le contenu d'une variable Tkinter. Il faut utiliser pour cela les méthodes spécifiques de cette classe d'objets : la méthode **get()** pour lire, et la méthode **set()** pour écrire :
  `m = self.actMusi.get()`
  Dans cette instruction, nous affectons à **m** (variable ordinaire de Python) le contenu de la variable Tkinter **self.actMusi** (laquelle est elle-même associée à un widget bien déterminé).

Tout ce qui précède peut vous paraître un peu compliqué. Considérez simplement qu'il s'agit de votre première rencontre avec les problèmes d'interfaçage entre deux langages de programmation différents, utilisés ensemble dans un projet composite.

Menu avec choix exclusifs

La deuxième partie du menu « Options » permet à l'utilisateur de choisir l'aspect que prendra la barre de menus, parmi six possibilités. Il va de soi que l'on ne peut activer qu'une seule de ces possibilités à la fois. Pour mettre en place ce genre de fonctionnalité, on fait classiquement appel appel à des widgets de type « boutons radio ». La caractéristique essentielle de ces widgets est que plusieurs d'entre eux doivent être associés à une seule et même variable Tkinter. A chaque bouton radio correspond alors une valeur particulière, et c'est cette valeur qui est affectée à la variable lorsque l'utilisateur sélectionne le bouton.

Ainsi, l'instruction :

```
    self.mo.add_radiobutton(label ='sillon', variable =self.relief,
                            value =3, command =self.reliefBarre)
```

configure un item du menu «Options» de telle manière qu'il se comporte comme un bouton radio.

Lorsque l'utilisateur sélectionne cet item, la valeur 3 est affectée à la variable Tkinter **self.relief** (celle-ci étant désignée à l'aide de l'option **variable** du widget), et un appel est lancé en direction de la méthode **reliefBarre()**. Celle-ci récupère alors la valeur mémorisée dans la variable Tkinter pour effectuer son travail.

Dans le contexte particulier de ce menu, nous souhaitons proposer 6 possibilités différentes à l'utilisateur. Il nous faut donc six « boutons radio », pour lesquels nous pourrions encoder six instructions similaires à celle que nous avons reproduite ci-dessus, chacune d'elles ne différant des cinq autres que par ses options **value** et **label**. Dans une situation de ce genre, la bonne pratique de programmation consiste à placer les valeurs de ces options dans une liste, et à parcourir en-

64 Voir également page 197.

suite cette liste à l'aide d'une boucle **for**, afin d'instancier les widgets avec une instruction commune :

```
      for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                       (3,'sillon'), (4,'crête'), (5,'bordure')]:
          self.mo.add_radiobutton(label =lab, variable =self.relief,
                                  value =v, command =self.reliefBarre)
```

La liste utilisée est une liste de 6 tuples (valeur, libellé). A chacune des 6 itérations de la boucle, un nouvel item radiobutton est instancié, dont les options label et value sont extraites de la liste par l'intermédiaire des variables **lab** et **v**.

Dans vos projets personnels, il vous arrivera fréquemment de constater que vous pouvez ainsi remplacer des suites d'instructions similaires, par une structure de programmation plus compacte (en général, la combinaison d'une liste et d'une boucle, comme dans l'exemple ci-dessus).

Vous découvrirez petit à petit encore d'autres techniques pour alléger votre code : nous en fournissons encore un exemple dans le paragraphe suivant. Tâchez cependant de garder à l'esprit cette règle essentielle, qu'un bon programme doit avant tout rester *très lisible* et *bien commenté*.

Contrôle du flux d'exécution à l'aide d'une liste

Veuillez à présent considérer la définition de la méthode **reliefBarre()** :

A la première ligne, la méthode **get()** nous permet de récupérer l'état d'une variable Tkinter qui contient le numéro du choix opéré par l'utilisateur dans le sous-menu « Relief : ».

A la seconde ligne, nous utilisons le contenu de la variable **choix** pour extraire d'une liste de six éléments celui qui nous intéresse. Par exemple, si **choix** contient la valeur 2, c'est l'option **SUNKEN** qui sera utilisée pour reconfigurer le widget.

La variable **choix** est donc utilisée ici comme un *index*, servant à désigner un élément de la liste. En lieu et place de cette construction compacte, nous aurions pu programmer une série de tests conditionnels, comme par exemple :

```
if choix ==0:
    self.configure(relief =FLAT)
elif choix ==1:
    self.configure(relief =RAISED)
elif choix ==2:
    self.configure(relief =SUNKEN)
...
etc.
```

D'un point de vue strictement fonctionnel, le résultat serait exactement le même. Vous admettrez cependant que la construction que nous avons choisie est d'autant plus efficiente, que le nombre de possibilités de choix est élevé. Imaginez par exemple que l'un de vos programmes personnels doive effectuer une sélection dans un très grand nombre d'éléments : avec une construction du type ci-dessus, vous seriez peut-être amené à encoder plusieurs pages de **elif** !

Nous utilisons encore la même technique dans la méthode **choixActifs()**. Ainsi l'instruction :

```
      self.pein.configure(state =[DISABLED, NORMAL][p])
```

utilise le contenu de la variable **p** comme index pour désigner lequel des deux états **DISABLED**, **NORMAL** doit être sélectionné pour reconfigurer le menu « Peintres ».

Lorsqu'elle est appelée, la méthode **choixActifs()** reconfigure donc les deux rubriques « Peintres » et « Musiciens » de la barre de menus, pour les faire apparaître « normales » ou « désactivées » en fonction de l'état des variables **m** et **p**, lesquelles sont elles-mêmes le reflet de variables Tkinter.

Ces variables intermédiaires **m** et **p** ne servent en fait qu'à clarifier le script. Il serait en effet parfaitement possible de les éliminer, et de rendre le script encore plus compact, en utilisant la composition d'instructions. On pourrait par exemple remplacer les deux instructions :

```
    m = self.actMusi.get()
    self.musi.configure(state =[DISABLED, NORMAL][m])
```

par une seule, telle que :

```
    self.musi.configure(state =[DISABLED, NORMAL][self.actMusi.get()])
```

Notez cependant que ce que l'on gagne en compacité peut se payer d'une certaine perte de lisibilité.

Pré-sélection d'une rubrique

Pour terminer cet exercice, voyons encore comment vous pouvez déterminer à l'avance certaines sélections, ou bien les modifier par programme.

Veuillez donc ajouter l'instruction suivante dans le constructeur de la classe **Application()** (juste avant l'instruction **self.pack()** ,par exemple) :

```
    mBar.mo.invoke(2)
```

Lorsque vous exécutez le script ainsi modifié, vous constatez qu'au départ la rubrique « Musiciens » de la barre de menus est active, alors que la rubrique « Peintres » ne l'est pas. Programmées comme elles le sont, ces deux rubriques devraient être actives toutes deux par défaut. Et c'est effectivement ce qui se passe si nous supprimons l'instruction `mBar.mo.invoke(2)`

Nous vous avons suggéré d'ajouter cette instruction au script, pour vous montrer comment vous pouvez effectuer par programme la même opération que celle que l'on obtient normalement avec un clic de souris.

L'instruction ci-dessus invoque le widget **mBar.mo** en actionnant la commande associée au deuxième item de ce widget. En consultant le listing, vous pouvez vérifier que ce deuxième item est bien l'objet de type **checkbutton** qui active/désactive le menu « Peintres » (Rappelons encore une fois que l'on numérote toujours à partir de zéro).

Au démarrage du programme, tout se passe donc comme si l'utilisateur effectuait tout de suite un premier clic sur la rubrique « Peintres » du menu « Options », ce qui a pour effet de désactiver le menu correspondant.

15

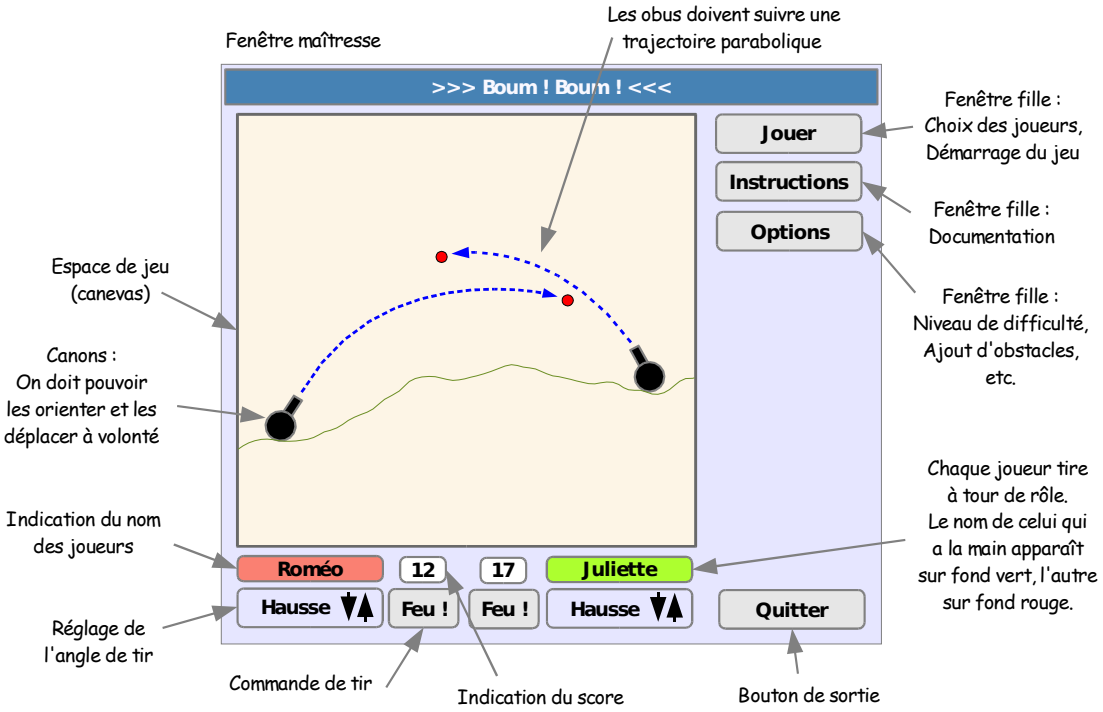
Analyse de programmes concrets

Dans ce chapitre, nous allons nous efforcer d'illustrer la démarche de conception d'un programme graphique, depuis ses premières ébauches jusqu'à un stade de développement relativement avancé. Nous souhaitons montrer ainsi combien la programmation orientée objet peut faciliter et surtout *sécuriser* la stratégie de *développement incrémental* que nous préconisons[65].

L'utilisation de classes s'impose, lorsque l'on constate qu'un projet en cours de réalisation se révèle nettement plus complexe que ce que l'on avait imaginé au départ. Vous vivrez certainement vous-même des cheminements similaires à celui que nous décrivons ci-dessous.

Jeu des bombardes

Ce projet de jeu[66] s'inspire d'un travail similaire réalisé par des élèves de terminale.



Il est vivement recommandé de commencer l'ébauche d'un tel projet par une série de petits dessins et de schémas, dans lesquels seront décrits les différents éléments graphiques à construire, ainsi qu'un maximum de *cas d'utilisations*. Si vous rechignez à utiliser pour cela la bonne vieille technologie papier/crayon (laquelle a pourtant bien fait ses preuves), vous pouvez tirer profit

65 Voir page 18 : *Recherche des erreurs et expérimentation*, et aussi page 225 : *Fenêtres avec menus*

66 Nous n'hésitons pas à discuter ici le développement d'un logiciel de jeu, parce qu'il s'agit d'un domaine directement accessible à tous, et dans lequel les objectifs concrets sont aisément identifiables. Il va de soi que les mêmes techniques de développement peuvent s'appliquer à d'autres applications plus "sérieuses".

d'un logiciel de dessin technique, tel l'utilitaire *Draw* de la suite bureautique *OpenOffice.org*[67]. C'est l'outil qui a été utilisé pour réaliser le schéma ci-dessous :

L'idée de départ est simple : deux joueurs s'affrontent au canon. Chacun doit ajuster son angle de tir pour tâcher d'atteindre son adversaire, les obus décrivant des trajectoires balistiques.

L'emplacement des canons est défini au début du jeu de manière aléatoire (tout au moins en hauteur). Après chaque tir, les canons se déplacent (afin d'accroître l'intérêt du jeu, l'ajustement des tirs étant ainsi rendu plus difficile). Les coups au but sont comptabilisés.

Le dessin préliminaire que nous avons reproduit ci-dessus est l'une des formes que peut prendre votre *travail d'analyse*. Avant de commencer le développement d'un projet de programmation, il vous faut en effet toujours vous efforcer d'établir un *cahier des charges* détaillé. Cette étude préalable est très importante. La plupart des débutants commencent bien trop vite à écrire de nombreuses lignes de code au départ d'une vague idée, en négligeant de rechercher la structure d'ensemble. Leur programmation risque alors de devenir chaotique, parce qu'ils devront de toute façon mettre en place cette structure tôt ou tard. Il s'apercevront alors bien souvent qu'il leur faut supprimer et ré-écrire des pans entiers d'un projet qu'ils ont conçu d'une manière *trop monolithique* et/ou *mal paramétrée*.

- *Trop monolithique* : cela signifie que l'on a négligé de décomposer un problème complexe en plusieurs sous-problèmes plus simples. Par exemple, on a imbriqué plusieurs niveaux successifs d'instructions composées, au lieu de faire appel à des fonctions ou à des classes.
- *Mal paramétrée* : cela signifie que l'on a traité seulement un cas particulier, au lieu d'envisager le cas général. Par exemple, on a donné à un objet graphique des dimensions fixes, au lieu de prévoir des variables pour permettre son redimensionnement.

Vous devez donc toujours commencer le développement d'un projet par une phase d'analyse aussi fouillée que possible, et concrétiser le résultat de cette analyse dans un ensemble de documents (schémas, plans, descriptions...) qui constitueront le cahier des charges. Pour les projets de grande envergure, il existe d'ailleurs des méthodes d'analyse très élaborées (*UML*, *Merise*...) que nous ne pouvons nous permettre de décrire ici car elles font l'objet de livres entiers.

Cela étant dit, il faut malheureusement admettre qu'il est très difficile (et même probablement impossible) de réaliser dès le départ l'analyse tout à fait complète d'un projet de programmation. C'est seulement lorsqu'il commence à fonctionner véritablement qu'un programme révèle ses faiblesses. On constate alors qu'il reste des cas d'utilisation ou des contraintes qui n'avaient pas été prévues au départ. D'autre part, un projet logiciel est pratiquement toujours destiné à évoluer : il vous arrivera fréquemment de devoir modifier le cahier des charges au cours du développement lui-même, pas nécessairement parce que l'analyse initiale a été mal faite, mais tout simplement parce que l'on souhaite encore ajouter des fonctionnalités supplémentaires.

En conclusion, tâchez de toujours aborder un nouveau projet de programmation en respectant les deux consignes suivantes :

- Décrivez votre projet en profondeur avant de commencer la rédaction des premières lignes de code, en vous efforçant de mettre en évidence les composants principaux et les relations qui les lient (pensez notamment à décrire les différents cas d'utilisation de votre programme)
- Lorsque vous commencerez sa réalisation effective, évitez de vous laisser entraîner à rédiger de trop grands blocs d'instructions. Veillez au contraire à découper votre application en un certain nombre de composants *paramétrables* bien *encapsulés*, de telle manière que vous puis-

[67] Il s'agit d'une suite bureautique complète, libre et gratuite, largement compatible avec MS-Office, disponible pour Linux, Windows, MacOS, Solaris ... Le présent manuel a été entièrement rédigé avec son traitement de textes.
Vous pouvez vous la procurer par téléchargement depuis le site Web : *http://www.openoffice.org*

siez aisément modifier l'un ou l'autre d'entre eux sans compromettre le fonctionnement des autres, et peut-être même les réutiliser dans différents contextes si le besoin s'en fait sentir.

C'est pour satisfaire cette exigence que la programmation orientée objets est a été inventée.

Considérons par exemple l'ébauche dessinée à la page précédente.

L'apprenti programmeur sera peut-être tenté de commencer la réalisation de ce jeu en n'utilisant que la seule programmation procédurale (c'est-à-dire en omettant de définir de nouvelles classes). C'est d'ailleurs ainsi que nous avons procédé nous-même lors de notre première approche des interfaces graphiques, tout au long du chapitre 8. Cette façon de procéder ne se justifie cependant que pour de tout petits programmes (des exercices ou des tests préliminaires). Lorsque l'on s'attaque à un projet d'une certaine importance, la complexité des problèmes qui se présentent se révèle rapidement trop importante, et il devient alors indispensable de fragmenter et de compartimenter.

L'outil logiciel qui va permettre cette fragmentation est la *classe*.

Nous pouvons peut-être mieux comprendre son utilité en nous aidant d'une analogie :

Tous les appareils électroniques sont constitués d'un petit nombre de composants de base, à savoir des transistors, des diodes, des résistances, des condensateurs, etc. Les premiers ordinateurs ont été construits directement à partir de ces composants. Ils étaient volumineux, très chers, et pourtant ils n'avaient que très peu de fonctionnalités et tombaient fréquemment en panne.

On a alors développé différentes techniques pour encapsuler dans un même boîtier un certain nombre de composants électroniques de base. Pour utiliser ces nouveaux *circuits intégrés*, il n'était plus nécessaire de connaître leur contenu exact : seule importait leur fonction globale. Les premières fonctions intégrées étaient encore relativement simples : c'étaient par exemple des portes logiques, des bascules, etc. En combinant ces circuits entre eux, on obtenait des caractéristiques plus élaborées, telles que des registres ou des décodeurs, qui purent à leur tour être intégrés, et ainsi de suite, jusqu'aux microprocesseurs actuels. Ceux-ci contiennent dorénavant plusieurs millions de composants, et pourtant leur fiabilité reste extrêmement élevée.

En conséquence, pour l'électronicien moderne qui veut construire par exemple un compteur binaire (circuit qui nécessite un certain nombre de bascules), il est évidemment bien plus simple, plus rapide et plus sûr de se servir de bascules intégrées, plutôt que de s'échiner à combiner sans erreur plusieurs centaines de transistors et de résistances.

D'une manière analogue, le programmeur moderne que vous êtes peut bénéficier du travail accumulé par ses prédécesseurs en utilisant la fonctionnalité intégrée dans les nombreuses bibliothèques de classes déjà disponibles pour Python. Mieux encore, il peut aisément créer lui-même de nouvelles classes pour encapsuler les principaux composants de son application, particulièrement ceux qui y apparaissent en plusieurs exemplaires. Procéder ainsi est plus simple, plus rapide et plus sûr que de multiplier les blocs d'instructions similaires dans un corps de programme monolithique, de plus en plus volumineux et de moins en moins compréhensible.

Examinons à présent notre ébauche dessinée. Les composants les plus importants de ce jeu sont bien évidemment les petits canons, qu'il faudra pouvoir dessiner à différents emplacements et dans différentes orientations, et dont il nous faudra au moins deux exemplaires.

Plutôt que de les dessiner morceau par morceau dans le canevas au fur et à mesure du déroulement du jeu, nous avons intérêt à les considérer comme des objets logiciels à part entière, dotés de plusieurs propriétés ainsi que d'un certain comportement (ce que nous voulons exprimer par là est le fait qu'il devront être dotés de divers mécanismes, que nous pourrons activer par pro-

gramme à l'aide de *méthodes* particulières). Il est donc certainement judicieux de leur consacrer une classe spécifique.

Prototypage d'une classe « Canon »

En définissant une telle classe, nous gagnons sur plusieurs tableaux. Non seulement nous rassemblons ainsi tout le code correspondant au dessin et au fonctionnement du canon dans une même « capsule », bien à l'écart du reste du programme, mais de surcroît nous nous donnons la possibilité d'instancier aisément un nombre quelconque de ces canons dans le jeu, ce qui nous ouvre des perspectives de développements ultérieurs.

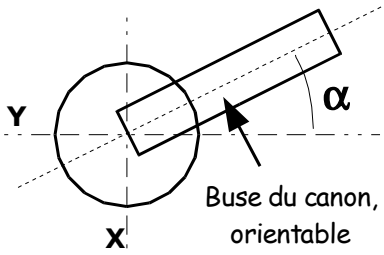
Lorsqu'une première implémentation de la classe **Canon()** aura été construite et testée, il sera également possible de la perfectionner en la dotant de caractéristiques supplémentaires, sans modifier (ou très peu) son interface, c'est-à-dire en quelque sorte son « mode d'emploi » : à savoir les instructions nécessaires pour l'instancier et l'utiliser dans des applications diverses.

Entrons à présent dans le vif du sujet.

Le dessin de notre canon peut être simplifié à l'extrême. Nous avons estimé qu'il pouvait se résumer à un cercle combiné avec un rectangle, celui-ci pouvant d'ailleurs être lui-même considéré comme un simple segment de ligne droite particulièrement épais.

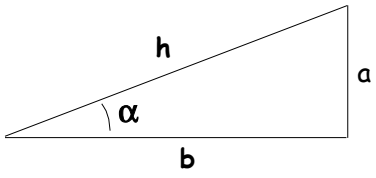
Si l'ensemble est rempli d'une couleur uniforme (en noir, par exemple), nous obtiendrons ainsi une sorte de petite bombarde suffisamment crédible.

Dans la suite du raisonnement, nous admettrons que la position du canon est en fait la position du centre du cercle (coordonnées x et y dans le dessin ci-contre). Ce point clé indique également l'axe de rotation de la buse du canon, ainsi que l'une des extrémités de la ligne épaisse qui représentera cette buse.



Pour terminer notre dessin, il nous restera alors à déterminer les coordonnées de l'autre extrémité de cette ligne. Ces coordonnées peuvent être calculées sans grande difficulté, à la condition de nous remémorer deux concepts fondamentaux de la trigonométrie (le sinus et le cosinus) que vous devez certainement bien connaître :

Dans un triangle rectangle, le rapport entre le coté opposé à un angle et l'hypoténuse du triangle est une propriété spécifique de cet angle qu'on appelle sinus de l'angle. Le cosinus du même angle est le rapport entre le côté adjacent à l'angle et l'hypoténuse.



Ainsi, dans le schéma ci-contre : $\sin\alpha = \frac{a}{h}$ et $\cos\alpha = \frac{b}{h}$.

Pour représenter la buse de notre canon, en supposant que nous connaissions sa longueur l et l'angle de tir α , il nous faut donc tracer un segment de ligne droite épaisse, à partir des coordonnées du centre du cercle (x et y), jusqu'à un autre point situé plus à droite et plus haut, l'écart horizontal Δx étant égal à l.cos α , et l'écart vertical Δy étant égal à l.sin α .

En résumant tout ce qui précède, dessiner un canon au point x, y consistera simplement à :

- tracer un cercle noir centré sur x, y
- tracer une ligne noire épaisse depuis le point x, y jusqu'au point x + l.cos α, y + l.sin α.

Nous pouvons à présent commencer à envisager une ébauche de programmation correspondant à une classe « Canon ». Il n'est pas encore question ici de programmer le jeu proprement dit. Nous voulons seulement vérifier si l'analyse que nous avons faite jusqu'à présent « tient la route », en réalisant un premier *prototype* fonctionnel.

Un prototype est un petit programme destiné à expérimenter une idée, que l'on se propose d'intégrer ensuite dans une application plus vaste. Du fait de sa simplicité et de sa concision, Python se prête fort bien à l'élaboration de prototypes, et de nombreux programmeurs l'utilisent pour mettre au point divers composants logiciels qu'ils reprogrammeront éventuellement ensuite dans d'autres langages plus « lourds », tels que le C par exemple.

Dans notre premier prototype, la classe **Canon()** ne comporte que deux méthodes : un constructeur qui crée les éléments de base du dessin, et une méthode permettant de modifier celui-ci à volonté pour ajuster l'angle de tir (l'inclinaison de la buse). Comme nous l'avons souvent fait dans d'autres exemples, nous inclurons quelques lignes de code à la fin du script afin de pouvoir tester la classe tout de suite :

```
 1#  from Tkinter import *
 2#  from math import pi, sin, cos
 3#
 4#  class Canon(object):
 5#      """Petit canon graphique"""
 6#      def __init__(self, boss, x, y):
 7#          self.boss = boss            # référence du canevas
 8#          self.x1, self.y1 = x, y     # axe de rotation du canon
 9#          # dessiner la buse du canon, à l'horizontale pour commencer :
10#          self.lbu = 50               # longueur de la buse
11#          self.x2, self.y2 = x + self.lbu, y
12#          self.buse = boss.create_line(self.x1, self.y1, self.x2, self.y2,
13#                                       width =10)
14#          # dessiner ensuite le corps du canon par-dessus :
15#          r = 15                      # rayon du cercle
16#          boss.create_oval(x-r, y-r, x+r, y+r, fill='blue', width =3)
17#
18#      def orienter(self, angle):
19#          "choisir l'angle de tir du canon"
20#          # rem : le paramètre <angle> est reçu en tant que chaîne de car.
21#          # il faut le traduire en nombre réel, puis convertir en radians :
22#          self.angle = float(angle)*2*pi/360
23#          self.x2 = self.x1 + self.lbu*cos(self.angle)
24#          self.y2 = self.y1 - self.lbu*sin(self.angle)
25#          self.boss.coords(self.buse, self.x1, self.y1, self.x2, self.y2)
26#
27#  if __name__ == '__main__':
28#      # Code pour tester sommairement la classe Canon :
29#      f = Tk()
30#      can = Canvas(f,width =250, height =250, bg ='ivory')
31#      can.pack(padx =10, pady =10)
32#      c1 = Canon(can, 50, 200)
33#
34#      s1 =Scale(f, label='hausse', from_=90, to=0, command=c1.orienter)
35#      s1.pack(side=LEFT, pady =5, padx =20)
36#      s1.set(25)                          # angle de tir initial
37#
38#      f.mainloop()
```

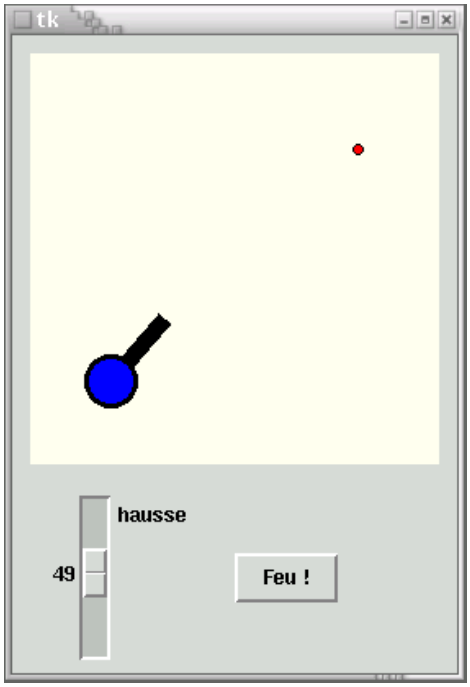
Commentaires :

- Ligne 6 : Dans la liste des paramètres qui devront être transmis au constructeur lors de l'instanciation, nous prévoyons les coordonnées **x** et **y**, qui indiqueront l'emplacement du canon dans le canevas, mais également une référence au canevas lui-même (la variable **boss**). Cette référence est indispensable : elle sera utilisée pour invoquer les méthodes du canevas.
  Nous pourrions inclure aussi un paramètre pour choisir un angle de tir initial, mais puisque

nous avons l'intention d'implémenter une méthode spécifique pour régler cette orientation, il sera plus judicieux de faire appel à celle-ci au moment voulu.

- Lignes 7 et 8 : Ces références seront utilisées un peu partout dans les différentes méthodes que nous allons développer dans la classe. Il faut donc en faire des attributs d'instance.
- Lignes 9 à 16 : Nous dessinons la buse d'abord, et le corps du canon ensuite. Ainsi une partie de la buse reste cachée. Cela nous permet de colorer éventuellement le corps du canon.
- Lignes 18 à 25 : Cette méthode sera invoquée avec un argument **angle**, lequel sera fourni en degrés (comptés à partir de l'horizontale). S'il est produit à l'aide d'un widget tel que **Entry** ou **Scale**, il sera transmis sous la forme d'une chaîne de caractères, et nous devrons donc le convertir d'abord en nombre réel avant de l'utiliser dans nos calculs (ceux-ci ont été décrits à la page précédente).
- Lignes 27 à 38 : Pour tester notre nouvelle classe, nous ferons usage d'un widget **Scale**. Pour définir la position initiale de son curseur, et donc fixer l'angle de hausse initial du canon, nous devons faire appel à sa méthode **set()** (ligne 36).

Ajout de méthodes au prototype



Notre prototype est fonctionnel, mais beaucoup trop rudimentaire. Nous devons à présent le perfectionner pour lui ajouter la capacité de tirer des obus.

Ceux-ci seront traités plutôt comme des « boulets » : ce seront de simples petits cercles que nous ferons partir de la bouche du canon avec une vitesse initiale d'orientation identique à celle de sa buse. Pour leur faire suivre une trajectoire réaliste, nous devons à présent nous rappeler quelques éléments de physique :

Comment un objet laissé à lui-même évolue-t-il dans l'espace, si l'on néglige les phénomènes secondaires tels que la résistance de l'air ?

Ce problème peut vous paraître complexe, mais en réalité sa résolution est très simple : il vous suffit d'admettre que le boulet se déplace à la fois horizontalement et verticalement, et que ces deux mouvements, quoique simultanés, sont tout à fait indépendants l'un de l'autre.

Vous allez donc établir une boucle d'animation, dans laquelle vous recalculez les nouvelles coordonnées **x** et **y** du boulet à intervalles de temps réguliers, en sachant que :

- Le mouvement horizontal est *uniforme*. À chaque itération, il vous suffit d'augmenter graduellement la coordonnée **x** du boulet, en lui ajoutant toujours un même déplacement Δx.
- Le mouvement vertical est *uniformément accéléré*. Cela signifie simplement qu'à chaque itération, vous devez ajouter à la coordonnée **y** un déplacement Δy qui augmente lui-même graduellement, toujours de la même quantité.

Voyons cela dans le script :

Pour commencer, il faut ajouter les lignes suivantes à la fin de la méthode constructeur. Elles vont servir à créer l'objet « obus », et à préparer une variable d'instance qui servira d'interrupteur de

l'animation. L'obus est créé au départ avec des dimensions minimales (un cercle d'un seul pixel) afin de rester presqu'invisible :

```
        # dessiner un obus (réduit à un simple point, avant animation) :
        self.obus =boss.create_oval(x, y, x, y, fill='red')
        self.anim =False                    # interrupteur d'animation
        # retrouver la largeur et la hauteur du canevas :
        self.xMax =int(boss.cget('width'))
        self.yMax =int(boss.cget('height'))
```

Les deux dernières lignes utilisent la méthode **cget()** du widget « maître » (le canevas, ici), afin de retrouver certaines de ses caractéristiques. Nous voulons en effet que notre classe **Canon** soit généraliste, c'est-à-dire réutilisable dans n'importe quel contexte, et nous ne pouvons donc pas tabler à l'avance sur des dimensions particulières pour le canevas dans lequel ce canon sera utilisé.

> *Tkinter renvoie ces valeurs sous la forme de chaînes de caractères. Il faut donc les convertir dans un type numérique si nous voulons pouvoir les utiliser dans un calcul.*

Ensuite, nous devons ajouter deux nouvelles méthodes : l'une pour déclencher le tir, et l'autre pour gérer l'animation du boulet une fois que celui-ci aura été lancé :

```
 1#      def feu(self):
 2#          "déclencher le tir d'un obus"
 3#          if not self.anim:
 4#              self.anim =True
 5#              # position de départ de l'obus (c'est la bouche du canon) :
 6#              self.boss.coords(self.obus, self.x2 -3, self.y2 -3,
 7#                                          self.x2 +3, self.y2 +3)
 8#              v =15              # vitesse initiale
 9#              # composantes verticale et horizontale de cette vitesse :
10#              self.vy = -v *sin(self.angle)
11#              self.vx = v *cos(self.angle)
12#              self.animer_obus()
13#
14#      def animer_obus(self):
15#          "animation de l'obus (trajectoire balistique)"
16#          if self.anim:
17#              self.boss.move(self.obus, int(self.vx), int(self.vy))
18#              c = self.boss.coords(self.obus)     # coord. résultantes
19#              xo, yo = c[0] +3, c[1] +3   # coord. du centre de l'obus
20#              if yo > self.yMax or xo > self.xMax:
21#                  self.anim =False        # arrêter l'animation
22#              self.vy += .5
23#              self.boss.after(30, self.animer_obus)
```

Commentaires :

- Lignes 1 à 4 : Cette méthode sera invoquée par appui sur un bouton. Elle déclenche le mouvement de l'obus, et attribue une valeur « vraie » à notre « interrupteur d'animation » (la variable **self.anim** : voir ci-après). Il faut cependant nous assurer que pendant toute la durée de cette animation, un nouvel appui sur le bouton ne puisse pas activer d'autres boucles d'animation parasites. C'est le rôle du test effectué à la ligne 3 : le bloc d'instruction qui suit ne peut s'exécuter que si la variable **self.anim** possède la valeur « faux », ce qui signifie que l'animation n'a pas encore commencé.
- Lignes 5 à 7 : Le canevas Tkinter dispose de deux méthodes pour déplacer les objets graphiques :
  - La méthode **coords()** (utilisée à la ligne 6) effectue un positionnement absolu ; il faut cependant lui fournir toutes les coordonnées de l'objet (comme si on le redessinait).

  - La méthode **move()** (utilisée plus loin, à la ligne 17), provoque un déplacement relatif ; elle s'utilise avec deux arguments seulement, à savoir les composantes horizontale et verticale du déplacement souhaité.

- Lignes 8 à 12 : La vitesse initiale de l'obus est choisie à la ligne 8. Comme nous l'avons expliqué à la page précédente, le mouvement du boulet est la résultante d'un mouvement horizontal et d'un mouvement vertical. Nous connaissons la valeur de la vitesse initiale ainsi que son inclinaison (c'est-à-dire l'angle de tir). Pour déterminer les composantes horizontale et verticale de cette vitesse, il nous suffit d'utiliser des relations trigonométriques tout à fait similaires à que celles que nous avons déjà exploitées pour dessiner la buse du canon. Le signe - utilisé à la ligne 9 provient du fait que les coordonnées verticales se comptent de haut en bas.
  La ligne 12 active l'animation proprement dite.
- Lignes 14 à 23 : Cette procédure se ré-appelle elle-même toutes les 30 millisecondes par l'intermédiaire de la méthode **after()** invoquée à la ligne 23. Cela continue aussi longtemps que la variable **self.anim** (notre « interrupteur d'animation ») reste « vraie », condition qui changera lorsque les coordonnées de l'obus sortiront des limites imposées (test de la ligne 20).
- Lignes 18, 19 : Pour retrouver ces coordonnées après chaque déplacement, on fait appel encore une fois à la méthode **coords()** du canevas : utilisée cette fois avec la référence d'un objet graphique comme unique argument, elle renvoie ses quatre coordonnées dans un tuple.
- Lignes 17 et 22 : La coordonnée horizontale de l'obus augmente toujours de la même quantité (mouvement uniforme), tandis que la coordonnée verticale augmente d'une quantité qui est elle-même augmentée à chaque fois à la ligne 24 (mouvement uniformément accéléré). Le résultat est une trajectoire parabolique.

> *L'opérateur* `+=` *permet d'incrémenter une variable :*
> *ainsi* `a += 3` *équivaut à* `a = a + 3` *. Veuillez noter au passage que l'utilisation de cet opérateur spécifique est plus efficace que la ré-affectation utilisée jusqu'ici.*
>
> *À partir de la version 2.3, Python initialise automatiquement deux variables nommées* `True` *et* `False` *pour représenter la véracité et la fausseté d'une expression (notez bien que ces noms commencent tous deux par une majuscule). Comme nous l'avons fait dans le script ci-dessus, vous pouvez utiliser ces variables dans les expressions conditionnelles afin d'augmenter la lisibilité de votre code. Si vous préférez, vous pouvez cependant continuer à utiliser aussi des valeurs numériques, comme nous l'avons fait précédemment.* *(Cf. « Véracité/Fausseté d'une expression », page 63).*

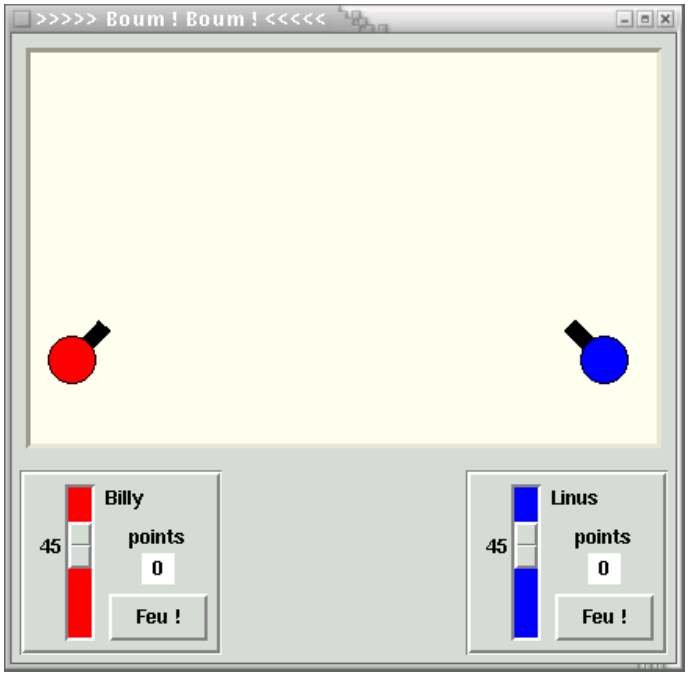
Il reste enfin à ajouter un bouton déclencheur dans la fenêtre principale. Une ligne telle que la suivante (à insérer dans le code de test) fera parfaitement l'affaire :

```
    Button(f, text='Feu !', command =c1.feu).pack(side=LEFT)
```

Développement de l'application

Disposant désormais d'une classe d'objets « canon » assez bien dégrossie, nous pouvons à présent envisager l'élaboration de l'application proprement dite. Et puisque nous sommes décidés à exploiter la méthodologie de la programmation orientée objet, nous devons concevoir cette application comme *un ensemble d'objets qui interagissent par l'intermédiaire de leurs méthodes.*

Plusieurs de ces objets proviendront de classes préexistantes, bien entendu : ainsi le canevas, les boutons, etc. Mais nous avons vu dans les pages précédentes que nous avons intérêt à regrouper des ensembles bien délimités de ces objets basiques dans de nouvelles classes, chaque fois que



nous pouvons identifier pour ces ensembles une fonctionnalité particulière. C'était le cas par exemple pour cet ensemble de cercles et de lignes mobiles que nous avons décidé d'appeler « canon ».

Pouvons-nous encore distinguer dans notre projet initial d'autres composants qui mériteraient d'être encapsulés dans des nouvelles classes ? Certainement. Il y a par exemple le pupitre de contrôle que nous voulons associer à chaque canon : nous pouvons y rassembler le dispositif de réglage de la hausse (l'angle de tir), le bouton de mise à feu, le score réalisé, et peut-être d'autres indications encore, comme le nom du joueur, par exemple. Il est d'autant plus intéressant de lui consacrer une classe particulière, que nous savons d'emblée qu'il nous en faudra deux instances.

Il y a aussi l'application elle-même, bien sûr. En l'encapsulant dans une classe, nous en ferons notre objet principal, celui qui dirigera tous les autres.

Veuillez à présent analyser le script ci-dessous. Vous y retrouverez la classe **Canon()** encore davantage développée : nous y avons ajouté quelques attributs et trois méthodes supplémentaires, afin de pouvoir gérer les déplacements du canon lui-même, ainsi que les coups au but.

La classe **Application()** remplace désormais le code de test des prototypes précédents. Nous y instancions deux objets **Canon()**, et deux objets de la nouvelle classe **Pupitre()**, que nous plaçons dans des dictionnaires en prévision de développements ultérieurs (nous pouvons en effet imaginer d'augmenter le nombre de canons et donc de pupitres). Le jeu est à présent fonctionnel : les canons se déplacent après chaque tir, et les coups au but sont comptabilisés.

```
  1# from Tkinter import *
  2# from math import sin, cos, pi
  3# from random import randrange
  4#
  5# class Canon(object):
  6#     """Petit canon graphique"""
  7#     def __init__(self, boss, id, x, y, sens, coul):
  8#         self.boss = boss            # réf. du canevas
  9#         self.appli = boss.master    # réf. de la fenêtre d'application
 10#         self.id = id                # identifiant du canon (chaîne)
 11#         self.coul = coul            # couleur associée au canon
 12#         self.x1, self.y1 = x, y     # axe de rotation du canon
 13#         self.sens = sens            # sens de tir (-1:gauche, +1:droite)
```

```
14#          self.lbu = 30               # longueur de la buse
15#          self.angle = 0              # hausse par défaut (angle de tir)
16#          # retrouver la largeur et la hauteur du canevas :
17#          self.xMax = int(boss.cget('width'))
18#          self.yMax = int(boss.cget('height'))
19#          # dessiner la buse du canon (horizontale) :
20#          self.x2, self.y2 = x + self.lbu * sens, y
21#          self.buse = boss.create_line(self.x1, self.y1,
22#                                       self.x2, self.y2, width =10)
23#          # dessiner le corps du canon (cercle de couleur) :
24#          self.rc = 15                # rayon du cercle
25#          self.corps = boss.create_oval(x -self.rc, y -self.rc, x +self.rc,
26#                                        y +self.rc, fill =coul)
27#          # pré-dessiner un obus caché (point en dehors du canevas) :
28#          self.obus = boss.create_oval(-10, -10, -10, -10, fill='red')
29#          self.anim = False           # indicateurs d'animation
30#          self.explo = False          #    et d'explosion
31#
32#      def orienter(self, angle):
33#          "régler la hausse du canon"
34#          # rem: le paramètre <angle> est reçu en tant que chaîne.
35#          # Il faut donc le traduire en réel, puis le convertir en radians :
36#          self.angle = float(angle)*pi/180
37#          # rem: utiliser la méthode coords de préférence avec des entiers :
38#          self.x2 = int(self.x1 + self.lbu * cos(self.angle) * self.sens)
39#          self.y2 = int(self.y1 - self.lbu * sin(self.angle))
40#          self.boss.coords(self.buse, self.x1, self.y1, self.x2, self.y2)
41#
42#      def deplacer(self, x, y):
43#          "amener le canon dans une nouvelle position x, y"
44#          dx, dy = x -self.x1, y -self.y1     # valeur du déplacement
45#          self.boss.move(self.buse, dx, dy)
46#          self.boss.move(self.corps, dx, dy)
47#          self.x1 += dx
48#          self.y1 += dy
49#          self.x2 += dx
50#          self.y2 += dy
51#
52#      def feu(self):
53#          "tir d'un obus - seulement si le précédent a fini son vol"
54#          if not (self.anim or self.explo):
55#              self.anim =True
56#              # récupérer la description de tous les canons présents :
57#              self.guns = self.appli.dictionnaireCanons()
58#              # position de départ de l'obus (c'est la bouche du canon) :
59#              self.boss.coords(self.obus, self.x2 -3, self.y2 -3,
60#                                          self.x2 +3, self.y2 +3)
61#              v = 17              # vitesse initiale
62#              # composantes verticale et horizontale de cette vitesse :
63#              self.vy = -v *sin(self.angle)
64#              self.vx = v *cos(self.angle) *self.sens
65#              self.animer_obus()
66#              return True     # => signaler que le coup est parti
67#          else:
68#              return False    # => le coup n'a pas pu être tiré
69#
70#      def animer_obus(self):
71#          "animer l'obus (trajectoire balistique)"
72#          if self.anim:
73#              self.boss.move(self.obus, int(self.vx), int(self.vy))
74#              c = self.boss.coords(self.obus)     # coord. résultantes
75#              xo, yo = c[0] +3, c[1] +3      # coord. du centre de l'obus
76#              self.test_obstacle(xo, yo)     # a-t-on atteint un obstacle ?
77#              self.vy += .4                  # accélération verticale
78#              self.boss.after(20, self.animer_obus)
79#          else:
80#              # animation terminée - cacher l'obus et déplacer les canons :
81#              self.fin_animation()
82#
83#      def test_obstacle(self, xo, yo):
```

```
 84#         "évaluer si l'obus a atteint une cible ou les limites du jeu"
 85#         if yo >self.yMax or xo <0 or xo >self.xMax:
 86#             self.anim =False
 87#             return
 88#         # analyser le dictionnaire des canons pour voir si les coord.
 89#         # de l'un d'entre eux sont proches de celles de l'obus :
 90#         for id in self.guns:              # id = clef dans dictionn.
 91#             gun = self.guns[id]           # valeur correspondante
 92#             if xo < gun.x1 +self.rc and xo > gun.x1 -self.rc \
 93#             and yo < gun.y1 +self.rc and yo > gun.y1 -self.rc :
 94#                 self.anim =False
 95#                 # dessiner l'explosion de l'obus (cercle jaune) :
 96#                 self.explo = self.boss.create_oval(xo -12, yo -12,
 97#                                 xo +12, yo +12, fill ='yellow', width =0)
 98#                 self.hit =id       # référence de la cible touchée
 99#                 self.boss.after(150, self.fin_explosion)
100#                 break
101#
102#     def fin_explosion(self):
103#         "effacer l'explosion ; ré-initaliser l'obus ; gérer le score"
104#         self.boss.delete(self.explo)    # effacer l'explosion
105#         self.explo =False               # autoriser un nouveau tir
106#         # signaler le succès à la fenêtre maîtresse :
107#         self.appli.goal(self.id, self.hit)
108#
109#     def fin_animation(self):
110#         "actions à accomplir lorsque l'obus a terminé sa trajectoire"
111#         self.appli.disperser()          # déplacer les canons
112#         # cacher l'obus (en l'expédiant hors du canevas) :
113#         self.boss.coords(self.obus, -10, -10, -10, -10)
114#
115# class Pupitre(Frame):
116#     """Pupitre de pointage associé à un canon"""
117#     def __init__(self, boss, canon):
118#         Frame.__init__(self, bd =3, relief =GROOVE)
119#         self.score =0
120#         self.appli =boss                # réf. de l'application
121#         self.canon =canon               # réf. du canon associé
122#         # Système de réglage de l'angle de tir :
123#         self.regl =Scale(self, from_ =85, to =-15, troughcolor=canon.coul,
124#                          command =self.orienter)
125#         self.regl.set(45)               # angle initial de tir
126#         self.regl.pack(side =LEFT)
127#         # Étiquette d'identification du canon :
128#         Label(self, text =canon.id).pack(side =TOP, anchor =W, pady =5)
129#         # Bouton de tir :
130#         self.bTir =Button(self, text ='Feu !', command =self.tirer)
131#         self.bTir.pack(side =BOTTOM, padx =5, pady =5)
132#         Label(self, text ="points").pack()
133#         self.points =Label(self, text=' 0 ', bg ='white')
134#         self.points.pack()
135#         # positionner à gauche ou à droite suivant le sens du canon :
136#         if canon.sens == -1:
137#             self.pack(padx =5, pady =5, side =RIGHT)
138#         else:
139#             self.pack(padx =5, pady =5, side =LEFT)
140#
141#     def tirer(self):
142#         "déclencher le tir du canon associé"
143#         self.canon.feu()
144#
145#     def orienter(self, angle):
146#         "ajuster la hausse du canon associé"
147#         self.canon.orienter(angle)
148#
149#     def attribuerPoint(self, p):
150#         "incrémenter ou décrémenter le score, de <p> points"
151#         self.score += p
152#         self.points.config(text = ' %s ' % self.score)
153#
```

```
154#  class Application(Frame):
155#      '''Fenêtre principale de l'application'''
156#      def __init__(self):
157#          Frame.__init__(self)
158#          self.master.title('>>>>> Boum ! Boum ! <<<<<')
159#          self.pack()
160#          self.jeu = Canvas(self, width =400, height =250, bg ='ivory',
161#                            bd =3, relief =SUNKEN)
162#          self.jeu.pack(padx =8, pady =8, side =TOP)
163#
164#          self.guns ={}           # dictionnaire des canons présents
165#          self.pupi ={}           # dictionnaire des pupitres présents
166#          # Instanciation de 2 'objets canons (+1, -1 = sens opposés) :
167#          self.guns["Billy"] = Canon(self.jeu, "Billy", 30, 200, 1, "red")
168#          self.guns["Linus"] = Canon(self.jeu, "Linus", 370,200,-1, "blue")
169#          # Instanciation de 2 pupitres de pointage associés à ces canons :
170#          self.pupi["Billy"] = Pupitre(self, self.guns["Billy"])
171#          self.pupi["Linus"] = Pupitre(self, self.guns["Linus"])
172#
173#      def disperser(self):
174#          "déplacer aléatoirement les canons"
175#          for id in self.guns:
176#              gun =self.guns[id]
177#              # positionner à gauche ou à droite, suivant sens du canon :
178#              if gun.sens == -1 :
179#                  x = randrange(320,380)
180#              else:
181#                  x = randrange(20,80)
182#              # déplacement proprement dit :
183#              gun.deplacer(x, randrange(150,240))
184#
185#      def goal(self, i, j):
186#          "le canon <i> signale qu'il a atteint l'adversaire <j>"
187#          if i != j:
188#              self.pupi[i].attribuerPoint(1)
189#          else:
190#              self.pupi[i].attribuerPoint(-1)
191#
192#      def dictionnaireCanons(self):
193#          "renvoyer le dictionnaire décrivant les canons présents"
194#          return self.guns
195#
196#  if __name__ =='__main__':
197#      Application().mainloop()
```

Commentaires :

- Ligne 7 : Par rapport au prototype, trois paramètres ont été ajoutés à la méthode constructeur. Le paramètre **id** nous permet d'identifier chaque instance de la classe **Canon()** à l'aide d'un nom quelconque. Le paramètre **sens** indique s'il s'agit d'un canon qui tire vers la droite (sens = 1) ou vers la gauche (sens = -1). Le paramètre **coul** spécifie la couleur associée au canon.
- Ligne 9 : Il faut savoir que tous les widgets Tkinter possèdent un attribut **master** qui contient la référence leur widget maître éventuel (leur « contenant »). Cette référence est donc pour nous celle de l'application principale. (Nous avons implémenté nous-mêmes une technique similaire pour référencer le canevas, à l'aide de l'attribut **boss**).
- Lignes 42 à 50 : Cette méthode permet d'amener le canon dans un nouvel emplacement. Elle servira à repositionner les canons au hasard après chaque tir, ce qui augmente l'intérêt du jeu.
- Lignes 56, 57 : Nous essayons de construire notre classe canon de telle manière qu'elle puisse être réutilisée dans des projets plus vastes, impliquant un nombre quelconque d'objets canons qui pourront apparaître et disparaître au fil des combats. Dans cette perspective, il faut

que nous puissions disposer d'une description de tous les canons présents, avant chaque tir, de manière à pouvoir déterminer si une cible a été touchée ou non. Cette description est gérée par l'application principale, dans un dictionnaire, dont on peut lui demander une copie par l'intermédiaire de sa méthode **dictionnaireCanons()**.

- Lignes 66 à 68 : Dans cette même perspective généraliste, il peut être utile d'informer éventuellement le programme appelant que le coup a effectivement été tiré ou non.
- Ligne 76 : L'animation de l'obus est désormais traitée par deux méthodes complémentaires. Afin de clarifier le code, nous avons placé dans une méthode distincte les instructions servant à déterminer si une cible a été atteinte (méthode **test_obstacle()**).
- Lignes 79 à 81 : Nous avons vu précédemment que l'on interrompt l'animation de l'obus en attribuant une valeur « fausse » à la variable **self.anim**. La méthode **animer_obus()** cesse alors de boucler et exécute le code de la ligne 81.
- Lignes 83 à 100 : Cette méthode évalue si les coordonnées actuelles de l'obus sortent des limites de la fenêtre, ou encore si elles s'approchent de celles d'un autre canon. Dans les deux cas, l'interrupteur d'animation est actionné, mais dans le second, on dessine une « explosion » jaune, et la référence du canon touché est mémorisée. La méthode annexe **fin_explosion**() est invoquée après un court laps de temps pour terminer le travail, c'est-à-dire effacer le cercle d'explosion et envoyer un message à la fenêtre maîtresse pour signaler le coup au but.
- Lignes 115 à 152 : La classe **Pupitre()** définit un nouveau widget par dérivation de la classe **Frame()**, selon une technique qui doit désormais vous être devenue familière. Ce nouveau widget regroupe les commandes de hausse et de tir, ainsi que l'afficheur de points associés à un canon bien déterminé. La correspondance visuelle entre les deux est assurée par l'adoption d'une couleur commune. Les méthodes **tirer()** et **orienter()** communiquent avec l'objet **Canon()** associé, par l'intermédiaire des méthodes de celui-ci.
- Lignes 154 à 171 : La fenêtre d'application est elle aussi un widget dérivé de **Frame()**. Son constructeur instancie les deux canons et leurs pupitres de pointage, en plaçant ces objets dans les deux dictionnaires **self.guns** et **self.pupi**. Cela permet d'effectuer ensuite divers traitements systématiques sur chacun d'eux (comme par exemple à la méthode suivante). En procédant ainsi, on se réserve en outre la possibilité d'augmenter sans effort le nombre de ces canons si nécessaire, dans les développements ultérieurs du programme.
- Lignes 173 à 183 : Cette méthode est invoquée après chaque tir pour déplacer aléatoirement les deux canons, ce qui augmente la difficulté du jeu.

Développements complémentaires

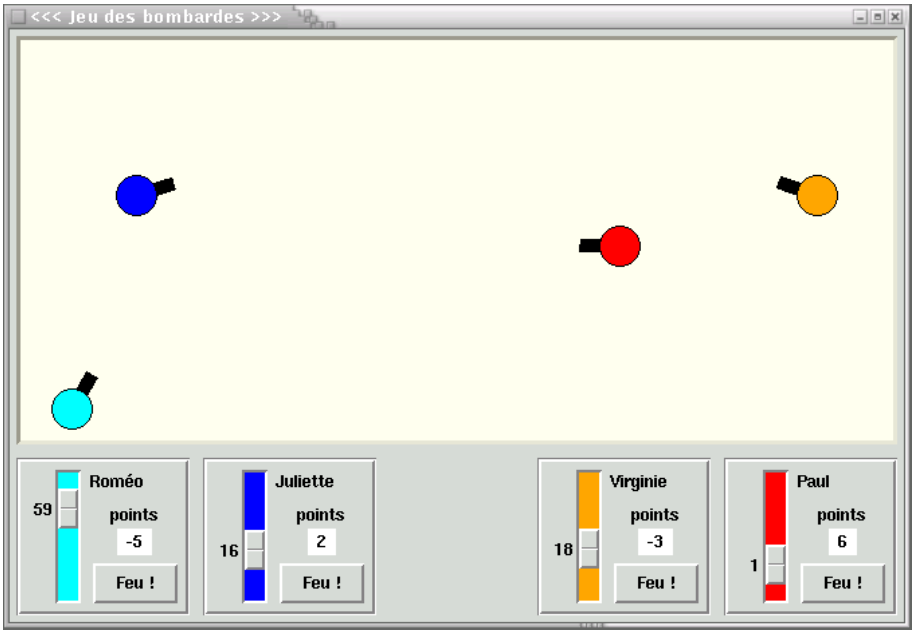
Tel qu'il vient d'être décrit, notre programme correspond déjà plus ou moins au cahier des charges initial, mais il est évident que nous pouvons continuer à le perfectionner.

A) Nous devrions par exemple *mieux le paramétrer*. Qu'est-ce à dire ? Dans sa forme actuelle, notre jeu comporte un canevas de taille prédéterminée (400 x 250 pixels, voir ligne 161). Si nous voulons modifier ces valeurs, nous devons veiller à modifier aussi les autres lignes du script où ces dimensions interviennent (comme par exemple aux lignes 168-169, ou 179-184). De telles lignes interdépendantes risquent de devenir nombreuses si nous ajoutons encore d'autres fonctionnalités. Il serait donc plus judicieux de dimensionner le canevas *à l'aide de variables*, dont la valeur serait définie en un seul endroit. Ces variables seraient ensuite exploitées dans toutes les lignes d'instructions où les dimensions du canevas interviennent.

Nous avons déjà effectué une partie de ce travail : dans la classe **Canon()**, en effet, les dimensions du canevas sont récupérées à l'aide d'une méthode prédéfinie (voir lignes 17-18), et placées dans des attributs d'instance qui peuvent être utilisés partout dans la classe.

B) Après chaque tir, nous provoquons un déplacement aléatoire des canons, en redéfinissant leurs coordonnées au hasard. Il serait probablement plus réaliste de provoquer de véritables *déplacements relatifs*, plutôt que de redéfinir au hasard des positions absolues. Pour ce faire, il suffit de retravailler la méthode **deplacer()** de la classe **Canon()**. En fait, il serait encore plus intéressant de faire en sorte que cette méthode puisse produire à volonté, aussi bien un déplacement relatif qu'un positionnement absolu, en fonction d'une valeur transmise en argument.

C) Le système de commande des tirs devrait être amélioré : puisque nous ne disposons que d'une seule souris, il faut demander aux joueurs de tirer à tour de rôle, et nous n'avons mis en place aucun mécanisme pour les forcer à le faire. Une meilleure approche consisterait à prévoir des commandes de hausse et de tir utilisant certaines touches du clavier, qui soient distinctes pour les deux joueurs.



D) Mais le développement le plus intéressant pour notre programme serait certainement d'en faire *une application réseau*. Le jeu serait alors installé sur plusieurs machines communicantes, chaque joueur ayant le contrôle d'un seul canon. Il serait d'ailleurs encore plus attrayant de permettre la mise en oeuvre de plus de deux canons, de manière à autoriser des combats impliquant davantage de joueurs.

Ce type de développement suppose cependant que nous ayons appris à maîtriser au préalable deux domaines de programmation qui débordent un peu le cadre de ce cours :

- la technique des *sockets*, qui permet d'établir une communication entre deux ordinateurs ;
- la technique des *threads*, qui permet à un même programme d'effectuer plusieurs tâches simultanément (cela nous sera nécessaire, si nous voulons construire une application capable de communiquer en même temps avec plusieurs partenaires).

Ces matières ne font pas strictement partie des objectifs que nous nous sommes fixés pour ce cours, et leur leur traitement nécessite à lui seul un chapitre entier. Nous n'aborderons donc pas cette question ici. Que ceux que le sujet intéresse se rassurent cependant : ce chapitre existe, mais

sous la forme d'un complément à la fin du livre (chapitre 18) : vous y trouverez la version réseau de notre jeu de bombardes.

En attendant, voyons tout de même comment nous pouvons encore progresser, en apportant à notre projet quelques améliorations qui en feront un jeu pour 4 joueurs. Nous nous efforcerons aussi de mettre en place une programmation bien compartimentée, de manière à ce que les méthodes de nos classes soient réutilisables dans une large mesure. Nous allons voir au passage comment cette évolution peut se faire sans modifier le code existant, en utilisant *l'héritage* pour produire de nouvelles classes à partir de celles qui sont déjà écrites.

Commençons par sauvegarder notre ouvrage précédent dans un fichier, dont nous admettrons pour la suite de ce texte que le nom est : **canon03.py**.

Nous disposons ainsi d'un véritable *module* Python, que nous pouvons importer dans un nouveau script à l'aide d'une seule ligne d'instruction. En exploitant cette technique, nous continuons à perfectionner notre application, en ne conservant sous les yeux que les nouveautés :

```
 1# from Tkinter import *
 2# from math import sin, cos, pi
 3# from random import randrange
 4# import canon03
 5#
 6# class Canon(canon03.Canon):
 7#     """Canon amélioré"""
 8#     def __init__(self, boss, id, x, y, sens, coul):
 9#         canon03.Canon.__init__(self, boss, id, x, y, sens, coul)
10#
11#     def deplacer(self, x, y, rel =False):
12#         "déplacement, relatif si <rel> est vrai, absolu si <rel> est faux"
13#         if rel:
14#             dx, dy = x, y
15#         else:
16#             dx, dy = x -self.x1, y -self.y1
17#         # limites horizontales :
18#         if self.sens ==1:
19#             xa, xb = 20, int(self.xMax *.33)
20#         else:
21#             xa, xb = int(self.xMax *.66), self.xMax -20
22#         # ne déplacer que dans ces limites :
23#         if self.x1 +dx < xa:
24#             dx = xa -self.x1
25#         elif self.x1 +dx > xb:
26#             dx = xb -self.x1
27#         # limites verticales :
28#         ya, yb = int(self.yMax *.4), self.yMax -20
29#         # ne déplacer que dans ces limites :
30#         if self.y1 +dy < ya:
31#             dy = ya -self.y1
32#         elif self.y1 +dy > yb:
33#             dy = yb -self.y1
34#         # déplacement de la buse et du corps du canon :
35#         self.boss.move(self.buse, dx, dy)
36#         self.boss.move(self.corps, dx, dy)
37#         # renvoyer les nouvelles coord. au programme appelant :
38#         self.x1 += dx
39#         self.y1 += dy
40#         self.x2 += dx
41#         self.y2 += dy
42#         return self.x1, self.y1
43#
44#     def fin_animation(self):
45#         "actions à accomplir lorsque l'obus a terminé sa trajectoire"
46#         # déplacer le canon qui vient de tirer :
47#         self.appli.depl_aleat_canon(self.id)
48#         # cacher l'obus (en l'expédiant hors du canevas) :
49#         self.boss.coords(self.obus, -10, -10, -10, -10)
```

```
 50#
 51#     def effacer(self):
 52#         "faire disparaître le canon du canevas"
 53#         self.boss.delete(self.buse)
 54#         self.boss.delete(self.corps)
 55#         self.boss.delete(self.obus)
 56#
 57#  class AppBombardes(Frame):
 58#     '''Fenêtre principale de l'application'''
 59#     def __init__(self, larg_c, haut_c):
 60#         Frame.__init__(self)
 61#         self.pack()
 62#         self.xm, self.ym = larg_c, haut_c
 63#         self.jeu = Canvas(self, width =self.xm, height =self.ym,
 64#                           bg ='ivory', bd =3, relief =SUNKEN)
 65#         self.jeu.pack(padx =4, pady =4, side =TOP)
 66#
 67#         self.guns ={}           # dictionnaire des canons présents
 68#         self.pupi ={}           # dictionnaire des pupitres présents
 69#         self.specificites()     # objets différents dans classes dérivées
 70#
 71#     def specificites(self):
 72#         "instanciation des canons et des pupitres de pointage"
 73#         self.master.title('<<< Jeu des bombardes >>>')
 74#         id_list =[("Paul","red"),("Roméo","cyan"),
 75#                   ("Virginie","orange"),("Juliette","blue")]
 76#         s = False
 77#         for id, coul in id_list:
 78#             if s:
 79#                 sens =1
 80#             else:
 81#                 sens =-1
 82#             x, y = self.coord_aleat(sens)
 83#             self.guns[id] = Canon(self.jeu, id, x, y, sens, coul)
 84#             self.pupi[id] = canon03.Pupitre(self, self.guns[id])
 85#             s = not s           # changer de côté à chaque itération
 86#
 87#     def depl_aleat_canon(self, id):
 88#         "déplacer aléatoirement le canon <id>"
 89#         gun =self.guns[id]
 90#         dx, dy = randrange(-60, 61), randrange(-60, 61)
 91#         # déplacement (avec récupération des nouvelles coordonnées) :
 92#         x, y = gun.deplacer(dx, dy, True)
 93#         return x, y
 94#
 95#     def coord_aleat(self, s):
 96#         "coordonnées aléatoires, à gauche (s =1) ou à droite (s =-1)"
 97#         y =randrange(int(self.ym /2), self.ym -20)
 98#         if s == -1:
 99#             x =randrange(int(self.xm *.7), self.xm -20)
100#         else:
101#             x =randrange(20, int(self.xm *.3))
102#         return x, y
103#
104#     def goal(self, i, j):
105#         "le canon n°i signale qu'il a atteint l'adversaire n°j"
106#         # de quel camp font-ils partie chacun ?
107#         ti, tj = self.guns[i].sens, self.guns[j].sens
108#         if ti != tj :               # ils sont de sens opposés :
109#             p = 1                   # on gagne 1 point
110#         else:                       # ils sont dans le même sens :
111#             p = -2                  # on a touché un allié !!
112#         self.pupi[i].attribuerPoint(p)
113#         # celui qui est touché perd de toute façon un point :
114#         self.pupi[j].attribuerPoint(-1)
115#
116#     def dictionnaireCanons(self):
117#         "renvoyer le dictionnaire décrivant les canons présents"
118#         return self.guns
119#
```

```
120#  if __name__ =='__main__':
121#      AppBombardes(650,300).mainloop()
```

Commentaires :

- Ligne 6 : La forme d'importation utilisée à la ligne 4 nous permet de redéfinir une nouvelle classe **Canon()** dérivée de la précédente, tout en lui conservant le même nom. De cette manière, les portions de code qui utilisent cette classe ne devront pas être modifiées (Cela n'aurait pas été possible si nous avions utilisé par exemple : « `from canon03 import *` »).
- Lignes 11 à 16 : La méthode définie ici porte le même nom qu'une méthode de la classe parente. *Elle va donc remplacer celle-ci dans la nouvelle classe* (On pourra dire également que la méthode **deplacer()** a été *surchargée*). Lorsque l'on réalise ce genre de modification, on s'efforce en général de faire en sorte que la nouvelle méthode effectue le même travail que l'ancienne quand elle est invoquée de la même façon que l'était cette dernière. On s'assure ainsi que les applications qui utilisaient la classe parente pourront aussi utiliser la classe fille, sans devoir être elles-mêmes modifiées.
  Nous obtenons ce résultat en ajoutant un ou plusieurs paramètres, dont les valeurs par défaut forceront l'ancien comportement. Ainsi, lorsque l'on ne fournit aucun argument pour le paramètre **rel**, les paramètres **x** et **y** sont utilisés comme des coordonnées absolues (ancien comportement de la méthode). Par contre, si l'on fournit pour **rel** un argument « vrai », alors les paramètres **x** et **y** sont traités comme des déplacements relatifs (nouveau comportement).
- Lignes 17 à 33 : Les déplacements demandés seront produits aléatoirement. Il nous faut donc prévoir un système de barrières logicielles, afin d'éviter que l'objet ainsi déplacé ne sorte du canevas.
- Ligne 42 : Nous renvoyons les coordonnées résultantes au programme appelant. Il se peut en effet que celui-ci commande un déplacement du canon sans connaître sa position initiale.
- Lignes 44 à 49 : Il s'agit encore une fois de *surcharger* une méthode qui existait dans la classe parente, de manière à obtenir un comportement différent : après chaque tir, désormais on ne disperse plus tous les canons présents, mais seulement celui qui vient de tirer.
- Lignes 51 à 55 : Méthode ajoutée en prévision d'applications qui souhaiteraient installer ou retirer des canons au fil du déroulement du jeu.
- Lignes 57 et suivantes : Cette nouvelle classe est conçue dès le départ de manière telle qu'elle puisse aisément être dérivée. C'est la raison pour laquelle nous avons fragmenté son constructeur en deux parties : La méthode **__init__()** contient le code commun à tous les objets, aussi bien ceux qui seront instanciés à partir de cette classe que ceux qui seront instanciés à partir d'une classe dérivée éventuelle. La méthode **specificites()** contient des portions de code plus spécifiques : cette méthode est clairement destinée à être *surchargée* dans les classes dérivées éventuelles.

Jeu de Ping

Dans les pages qui suivent, vous trouverez le script correspondant à un petit programme complet. Ce programme vous est fourni à titre d'exemple de ce que vous pouvez envisager de développer vous-même comme projet personnel de synthèse. Il vous montre encore une fois comment vous pouvez utiliser plusieurs classes afin de construire un script bien structuré. Il vous montre également comment vous pouvez paramétrer une application graphique de manière à ce que tout y soit *redimensionnable*.

Principe

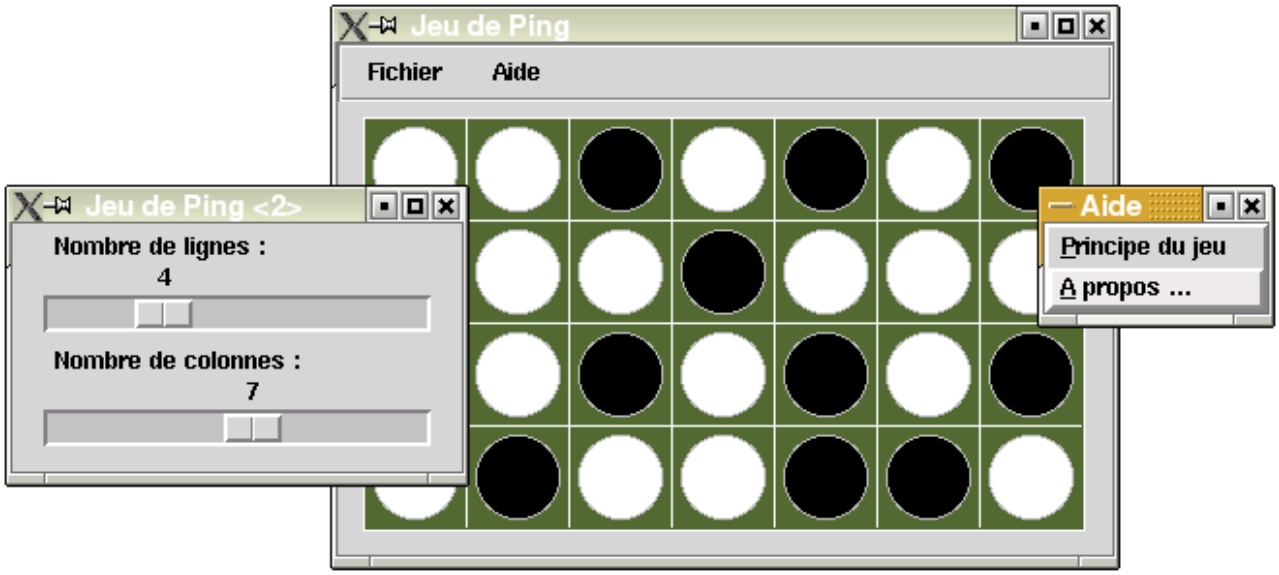
Le «jeu» mis en œuvre ici est plutôt une sorte d'exercice mathématique. Il se joue sur un panneau ou est représenté un quadrillage de dimensions variables, dont toutes les cases sont occupées par des pions. Ces pions possèdent chacun une face blanche et une face noire (comme les pions du jeu *Othello/Reversi*), et au début de l'exercice ils présentent tous leur face blanche par-dessus.

Lorsque l'on clique sur un pion à l'aide de la souris, les 8 pions adjacents se retournent.

Le jeu consiste alors à essayer de retourner tous les pions, en cliquant sur certains d'entre eux.

L'exercice est très facile avec une grille de 2 x 2 cases (il suffit de cliquer sur chacun des 4 pions). Il devient plus difficile avec des grilles plus grandes, et est même tout à fait impossible avec certaines d'entre elles. A vous de déterminer lesquelles !

(Ne négligez pas d'étudier le cas des grilles 1 x n).



Note : Vous trouverez la discussion complète du jeu de Ping, sa théorie et ses extensions, dans la revue « Pour la science » n° 298 - Août 2002, pages 98 à 102.

Programmation

Lorsque vous développez un projet logiciel, veillez toujours à faire l'effort de décrire votre démarche le plus clairement possible. Commencez par établir un cahier des charges détaillé, et ne négligez pas de commenter ensuite très soigneusement votre code, au fur et à mesure de son élaboration (et non après coup !).

En procédant ainsi, vous vous forcez vous-même à exprimer ce que vous souhaitez que la machine fasse, ce qui vous aide à analyser les problèmes et à structurer convenablement votre code.

Cahier des charges du logiciel à développer

- L'application sera construite sur la base d'une fenêtre principale comportant le panneau de jeu et une barre de menus.
- L'ensemble devra être extensible à volonté par l'utilisateur, les cases du panneau devant cependant rester carrées.
- Les options du menu permettront de :
  - choisir les dimensions de la grille (en nombre de cases)
  - réinitialiser le jeu (c'est-à-dire disposer tous les pions avec leur face blanche au-dessus)
  - afficher le principe du jeu dans une fenêtre d'aide
  - terminer.(fermer l'application)
- La programmation fera appel à trois classes :
  - une classe principale
  - une classe pour la barre de menus
  - une classe pour le panneau de jeu
- Le panneau de jeu sera dessiné dans un canevas, lui-même installé dans un cadre (*frame*). En fonction des redimensionnements opérés par l'utilisateur, le cadre occupera à chaque fois toute la place disponible : il se présentera donc au programmeur comme un rectangle quelconque, dont les dimensions doivent servir de base au calcul des dimensions de la grille à dessiner.
- Puisque les cases de cette grille doivent rester carrées, il est facile de commencer par calculer leur taille maximale, puis d'établir les dimensions du canevas en fonction de celle-ci.
- Gestion du clic de souris : on liera au canevas une méthode-gestionnaire pour l'événement <clic du bouton gauche>. Les coordonnées de l'événement serviront à déterminer dans quelle case de la grille (n° de ligne et n° de colonne) le clic a été effectué, quelles que soient les dimensions de cette grille. Dans les 8 cases adjacentes, les pions présents seront alors « retournés » (échange des couleurs noire et blanche).

```
##########################################
#  Jeu de ping                           #
#  Références : Voir article de la revue #
#  <Pour la science>, Aout 2002          #
#                                        #
# (C) Gérard Swinnen (Verviers, Belgique) #
# http://www.ulg.ac.be/cifen/inforef/swi #
#                                        #
#  Version du 29/09/2002 - Licence : GPL #
##########################################
```

```
from Tkinter import *

class MenuBar(Frame):
    """Barre de menus déroulants"""
    def __init__(self, boss =None):
        Frame.__init__(self, borderwidth =2, relief =GROOVE)
        ##### Menu <Fichier> #####
        fileMenu = Menubutton(self, text ='Fichier')
        fileMenu.pack(side =LEFT, padx =5)
        me1 = Menu(fileMenu)
        me1.add_command(label ='Options', underline =0,
                        command = boss.options)
        me1.add_command(label ='Restart', underline =0,
                        command = boss.reset)
        me1.add_command(label ='Terminer', underline =0,
                        command = boss.quit)
        fileMenu.configure(menu = me1)

        ##### Menu <Aide> #####
        helpMenu = Menubutton(self, text ='Aide')
        helpMenu.pack(side =LEFT, padx =5)
        me1 = Menu(helpMenu)
        me1.add_command(label ='Principe du jeu', underline =0,
                        command = boss.principe)
        me1.add_command(label ='A propos ...', underline =0,
                        command = boss.aPropos)
        helpMenu.configure(menu = me1)

class Panneau(Frame):
    """Panneau de jeu (grille de n x m cases)"""
    def __init__(self, boss =None):
        # Ce panneau de jeu est constitué d'un cadre redimensionnable
        # contenant lui-même un canevas. A chaque redimensionnement du
        # cadre, on calcule la plus grande taille possible pour les
        # cases (carrées) de la grille, et on adapte les dimensions du
        # canevas en conséquence.
        Frame.__init__(self)
        self.nlig, self.ncol = 4, 4         # Grille initiale = 4 x 4
        # Liaison de l'événement <resize> à un gestionnaire approprié :
        self.bind("<Configure>", self.redim)
        # Canevas :
        self.can =Canvas(self, bg ="dark olive green", borderwidth =0,
                         highlightthickness =1, highlightbackground ="white")
        # Liaison de l'événement <clic de souris> à son gestionnaire :
        self.can.bind("<Button-1>", self.clic)
        self.can.pack()
        self.initJeu()

    def initJeu(self):
        "Initialisation de la liste mémorisant l'état du jeu"
        self.etat =[]                 # construction d'une liste de listes
        for i in range(12):           # (équivalente à un tableau
            self.etat.append([0]*12)  #  de 12 lignes x 12 colonnes)

    def redim(self, event):
        "Opérations effectuées à chaque redimensionnement"
        # Les propriétés associées à l'événement de reconfiguration
        # contiennent les nouvelles dimensions du cadre :
        self.width, self.height = event.width -4, event.height -4
        # La différence de 4 pixels sert à compenser l'épaisseur
        # de la 'highlightbordure" entourant le canevas)
        self.traceGrille()

    def traceGrille(self):
        "Dessin de la grille, en fonction des options & dimensions"
        # largeur et hauteur maximales possibles pour les cases :
        lmax = self.width/self.ncol
        hmax = self.height/self.nlig
        # Le coté d'une case sera égal à la plus petite de ces dimensions :
        self.cote = min(lmax, hmax)
```

```
        # -> établissement de nouvelles dimensions pour le canevas :
        larg, haut = self.cote*self.ncol, self.cote*self.nlig
        self.can.configure(width =larg, height =haut)
        # Tracé de la grille :
        self.can.delete(ALL)                # Effacement dessins antérieurs
        s =self.cote
        for l in range(self.nlig -1):       # lignes horizontales
            self.can.create_line(0, s, larg, s, fill="white")
            s +=self.cote
        s =self.cote
        for c in range(self.ncol -1):       # lignes verticales
            self.can.create_line(s, 0, s, haut, fill ="white")
            s +=self.cote
        # Tracé de tous les pions, blancs ou noirs suivant l'état du jeu :
        for l in range(self.nlig):
            for c in range(self.ncol):
                x1 = c *self.cote +5            # taille des pions =
                x2 = (c +1)*self.cote -5        # taille de la case -10
                y1 = l *self.cote +5            #
                y2 = (l +1)*self.cote -5
                coul =["white","black"][self.etat[l][c]]
                self.can.create_oval(x1, y1, x2, y2, outline ="grey",
                                     width =1, fill =coul)

    def clic(self, event):
        "Gestion du clic de souris : retournement des pions"
        # On commence par déterminer la ligne et la colonne :
        lig, col = event.y/self.cote, event.x/self.cote
        # On traite ensuite les 8 cases adjacentes :
        for l in range(lig -1, lig+2):
            if l <0 or l >= self.nlig:
                continue
            for c in range(col -1, col +2):
                if c <0 or c >= self.ncol:
                    continue
                if l ==lig and c ==col:
                    continue
                # Retournement du pion par inversion logique :
                self.etat[l][c] = not (self.etat[l][c])
        self.traceGrille()


class Ping(Frame):
    """corps principal du programme"""
    def __init__(self):
        Frame.__init__(self)
        self.master.geometry("400x300")
        self.master.title(" Jeu de Ping")

        self.mbar = MenuBar(self)
        self.mbar.pack(side =TOP, expand =NO, fill =X)

        self.jeu =Panneau(self)
        self.jeu.pack(expand =YES, fill=BOTH, padx =8, pady =8)

        self.pack()

    def options(self):
        "Choix du nombre de lignes et de colonnes pour la grille"
        opt =Toplevel(self)
        curL =Scale(opt, length =200, label ="Nombre de lignes :",
              orient =HORIZONTAL,
              from_ =1, to =12, command =self.majLignes)
        curL.set(self.jeu.nlig)     # position initiale du curseur
        curL.pack()
        curH =Scale(opt, length =200, label ="Nombre de colonnes :",
              orient =HORIZONTAL,
              from_ =1, to =12, command =self.majColonnes)
        curH.set(self.jeu.ncol)
        curH.pack()
```

```
    def majColonnes(self, n):
        self.jeu.ncol = int(n)
        self.jeu.traceGrille()

    def majLignes(self, n):
        self.jeu.nlig = int(n)
        self.jeu.traceGrille()

    def reset(self):
        self.jeu.initJeu()
        self.jeu.traceGrille()

    def principe(self):
        "Fenêtre-message contenant la description sommaire du principe du jeu"
        msg =Toplevel(self)
        Message(msg, bg ="navy", fg ="ivory", width =400,
            font ="Helvetica 10 bold",
            text ="Les pions de ce jeu possèdent chacun une face blanche et "\
            "une face noire. Lorsque l'on clique sur un pion, les 8 "\
            "pions adjacents se retournent.\nLe jeu consiste a essayer "\
            "de les retouner tous.\n\nSi l'exercice se révèle très facile "\
            "avec une grille de 2 x 2 cases. Il devient plus difficile avec "\
            "des grilles plus grandes. Il est même tout à fait impossible "\
            "avec certaines grilles.\nA vous de déterminer lesquelles !\n\n"\
            "Réf : revue 'Pour la Science' - Aout 2002")\
            .pack(padx =10, pady =10)

    def aPropos(self):
        "Fenêtre-message indiquant l'auteur et le type de licence"
        msg =Toplevel(self)
        Message(msg, width =200, aspect =100, justify =CENTER,
            text ="Jeu de Ping\n\n(C) Gérard Swinnen, Aout 2002.\n"\
            "Licence = GPL").pack(padx =10, pady =10)

if __name__ == '__main__':
    Ping().mainloop()
```

> *Rappel : Si vous souhaitez expérimenter ces programmes sans avoir à les réécrire, vous pouvez trouver leur code source à l'adresse : http://www.ulg.ac.be/cifen/inforef/swi/python.htm*

16

Gestion d'une base de données

Les bases de données sont des outils de plus en plus fréquemment utilisés. Elles permettent de stocker des données nombreuses dans un seul ensemble bien structuré. Lorsqu'il s'agit de bases de données relationnelles, il devient en outre tout à fait possible d'éviter l'« enfer des doublons ». Vous avez sûrement été déjà confrontés à ce problème :

Des données identiques ont été enregistrées dans plusieurs fichiers différents. Lorsque vous souhaitez modifier ou supprimer l'une de ces données, vous devez ouvrir et modifier tous les fichiers qui la contiennent ! Le risque d'erreur est très réel, qui conduit inévitablement à des incohérences, sans compter la perte de temps que cela représente.

Les bases de données constituent *la* solution à ce type de problème. Python vous permet d'en utiliser de nombreux systèmes, mais nous n'en examinerons que deux dans nos exemples : *Gadfly* et *MySQL*.

Les bases de données

Il existe de nombreux types de bases de données. On peut par exemple déjà considérer comme une base de données élémentaire, un fichier qui contient une liste de noms et d'adresses.

Si la liste n'est pas trop longue, et si l'on ne souhaite pas pouvoir y effectuer des recherches en fonction de critères complexes, il va de soi que l'on peut accéder à ce type de données en utilisant des instructions simples, telles celles que nous avons abordées page Erreur : source de la référence non trouvée111.

La situation se complique cependant très vite si l'on souhaite pouvoir effectuer des sélections et des tris parmi les données, surtout si celles-ci deviennent très nombreuses. La difficulté augmente encore si les données sont répertoriées dans différents ensembles reliés par un certain nombre de relations hiérarchiques, et si plusieurs utilisateurs doivent pouvoir y accéder en parallèle.

Imaginez par exemple que la direction de votre école vous confie la charge de mettre au point un système de bulletins informatisé. En y réfléchissant quelque peu, vous vous rendrez compte rapidement que cela suppose la mise en œuvre de toute une série de tables différentes : une table des noms d'élèves (laquelle pourra bien entendu contenir aussi d'autres informations spécifiques à ces élèves : adresse, date de naissance, etc.) ; une table contenant la liste des cours (avec le nom du professeur titulaire, le nombre d'heures enseignées par semaine, etc.) ; une table mémorisant les travaux pris en compte pour l'évaluation (avec leur importance, leur date, leur contenu, etc.) ; une table décrivant la manière dont les élèves sont groupés par classes ou par options, les cours suivis par chacun, etc., etc.

Vous comprenez bien que ces différentes tables ne sont pas indépendantes. Les travaux effectués par un même élève sont liés à des cours différents. Pour établir le bulletin de cet élève, il faut donc extraire des données de la table des travaux, bien sûr, mais en relation avec des informations trouvées dans d'autres tables (celles des cours, des classes, des options, etc.)

Nous verrons plus loin comment représenter des tables de données et les relations qui les lient.

SGBDR - Le modèle client/serveur

Les programmes informatiques capables de gérer efficacement de tels ensembles de données complexes sont forcément complexes, eux aussi. On appelle ces programmes des *SGBDR* (Systèmes de Gestion de Bases de Données Relationnelles). Il s'agit d'applications informatiques de première importance pour les entreprises. Certaines sont les fleurons de sociétés spécialisées (*IBM*, *Oracle*, *Microsoft*, *Informix*, *Sybase...*) et sont en général vendues à des prix fort élevés. D'autres ont été développées dans des centres de recherche et d'enseignement universitaires (*PostgreSQL*, *MySQL* ...); elles sont alors en général tout à fait gratuites.

Ces systèmes ont chacun leurs spécificités et leurs performances, mais la plupart fonctionnant sur *le modèle client/serveur* : cela signifie que la plus grosse partie de l'application (ainsi que la base de données prise en charge) est installée en un seul endroit, en principe sur une machine puissante (cet ensemble constituant donc le *serveur*), alors que l'autre partie, beaucoup plus simple, est installée sur un nombre indéterminé de postes de travail, et on appelle celles-ci des *clients*.

Les clients sont reliés au serveur, en permanence ou non, par divers procédés et protocoles (éventuellement par l'intermédiaire de *l'internet*). Chacun d'entre eux peut accéder à une partie plus ou moins importante des données, avec autorisation ou non de modifier certaines d'entre elles, d'en ajouter ou d'en supprimer, en fonction de règles d'accès bien déterminées. (Ces règles sont définies par un administrateur de la base de données).

Le serveur et ses clients sont en fait des applications distinctes qui s'échangent des informations. Imaginez par exemple que vous êtes l'un des utilisateurs du système. Pour accéder aux données, vous devez lancer l'exécution d'une application cliente sur un poste de travail quelconque. Dans son processus de démarrage, l'application cliente commence par établir la connexion avec le serveur et la base de données[68]. Lorsque la connexion est établie, l'application cliente peut interroger le serveur en lui envoyant une *requête* sous une forme convenue. Il s'agit par exemple de retrouver une information précise. Le serveur exécute alors la requête en recherchant les données correspondantes dans la base, puis il expédie en retour une certaine *réponse* au client.

Cette réponse peut être l'information demandée, ou encore un message d'erreur en cas d'insuccès.

La communication entre le client et le serveur est donc faite de *requêtes* et de *réponses*. Les requêtes sont de véritables instructions expédiées du client au serveur, non seulement pour extraire des données de la base, mais aussi pour en ajouter, en supprimer, en modifier, etc.

Le langage SQL - Gadfly

Étant donnée la diversité des SGBDR existants, on pourrait craindre que chacun d'eux nécessite l'utilisation d'un langage particulier pour les requêtes qu'on lui adresse. En fait, de grands efforts ont été accomplis un peu partout pour la mise au point d'un langage commun, et il existe à présent un standard bien établi : le SQL (*Structured Query Language*, ou langage de requêtes structuré)[69].

Vous aurez probablement l'occasion de rencontrer SQL dans d'autres domaines (bureautique, par exemple). Dans le cadre de cette introduction à l'apprentissage de la programmation avec Python,

---

[68] il vous faudra certainement entrer quelques informations pour obtenir l'accès : adresse du serveur sur le réseau, nom de la base de données, nom d'utilisateur, mot de passe, ...

[69] Quelques variantes subsistent entre différentes implémentations du SQL, pour des requêtes très spécifiques, mais la base reste cependant la même.

nous allons nous limiter à la présentation de deux exemples : la mise en oeuvre d'un petit SGBDR réalisé exclusivement à l'aide de Python, et l'ébauche d'un logiciel client plus ambitieux destiné à communiquer avec un serveur de bases de données *MySQL*.

Notre première réalisation utilisera un module nommé *Gadfly*. Entièrement écrit en Python, ce module ne fait pas partie de la distribution standard et doit donc être installé séparément[70]. Il intègre un large sous-ensemble de commandes SQL. Ses performances ne sont évidemment pas comparables à celles d'un gros SGBDR spécialisé[71], mais elles sont tout à fait excellentes pour la gestion de bases de données modestes. Absolument portable comme Python lui-même, *Gadfly* fonctionnera indifféremment sous *Windows* , *Linux* ou *MacOS*. De même, les répertoires contenant des bases de données produites sous Gadfly pourront être utilisées sans modification depuis l'un ou l'autre de ces systèmes.

Si vous souhaitez développer une application qui doit gérer des relations relativement complexes dans une petite base de données, le module *Gadfly* peut vous faciliter grandement la tâche.

Mise en œuvre d'une base de données simple avec Gadfly

Nous allons ci-après examiner comment mettre en place une application simple, qui fasse office à la fois de serveur et de client sur la même machine.

Création de la base de données

Comme vous vous y attendez certainement, il suffit d'importer le module **gadfly** pour accéder aux fonctionnalités correspondantes.
Vous devez ensuite créer une instance (un objet) de la classe **gadfly()** :

```
import gadfly
baseDonn = gadfly.gadfly()
```

L'objet **baseDonn** ainsi créé est votre moteur de base de données local, lequel effectuera la plupart de ses opérations en mémoire vive. Ceci permet une exécution très rapide des requêtes.

Pour créer la base de données proprement dite, il faut employer la méthode **startup()** de cet objet :

```
baseDonn.startup("mydata","E:/Python/essais/gadfly")
```

Le premier paramètre transmis, **mydata**, est le nom choisi pour la base de données (vous pouvez évidemment choisir un autre nom !). Le second paramètre est le répertoire où l'on souhaite installer cette base de données. (Ce répertoire doit avoir été créé au préalable, et toute base de données de même nom qui préexisterait dans ce répertoire est écrasée sans avertissement).

Les trois lignes de code que vous venez d'entrer sont suffisantes : vous disposez dès à présent d'une base de données fonctionnelle, dans laquelle vous pouvez créer différentes tables, puis ajouter, supprimer ou modifier des données dans ces tables.

Pour toutes ces opérations, vous allez utiliser le langage SQL.

---

[70] Le module Gadfly est disponible gratuitement sur l'internet. Voir http://sourceforge.net/projects/gadfly
L'installation de ce module est décrite dans l'annexe 17.6 , page 315.

[71] Gadfly se révèle relativement efficace pour la gestion de bases de données de taille moyenne, en mode mono-utilisateur. Pour gérer de grosses bases de données en mode multi-utilisateur, il faut faire appel à des SGDBR plus ambitieux tels que PostgreSQL, pour lesquels des modules clients Python existent aussi (Pygresql, par ex.).

Afin de pouvoir transmettre vos requêtes SQL à l'objet **baseDonn** , vous devez cependant mettre en œuvre un *curseur.* Il s'agit d'une sorte de tampon mémoire intermédiaire, destiné à mémoriser temporairement les données en cours de traitement, ainsi que les opérations que vous effectuez sur elles, avant leur transfert définitif dans de vrais fichiers. Cette technique permet donc d'annuler si nécessaire une ou plusieurs opérations qui se seraient révélées inadéquates (Vous pouvez en apprendre davantage sur ce concept en consultant l'un des nombreux manuels qui traitent du langage SQL).

Veuillez à présent examiner le petit script ci-dessous, et noter que les requêtes SQL sont des chaînes de caractères, prises en charge par la méthode **execute()** de l'objet curseur :

```
cur = baseDonn.cursor()
cur.execute("create table membres (age integer, nom varchar, taille float)")
cur.execute("insert into membres(age, nom, taille) values (21,'Dupont',1.83)")
cur.execute("INSERT INTO MEMBRES(AGE, NOM, TAILLE) VALUES (15,'Suleau',1.57)")
cur.execute("Insert Into Membres(Age, Nom, Taille) Values (18,'Forcas',1.69)")
baseDonn.commit()
```

La première des lignes ci-dessus crée l'objet curseur **cur**. Les chaînes de caractères comprises entre guillemets dans les 4 lignes suivantes contiennent des requêtes SQL très classiques. Notez bien que *le langage SQL ne tient aucun compte de la casse des caractères* : vous pouvez encoder vos requêtes SQL indifféremment en majuscules ou en minuscules (ce qui n'est pas le cas pour les instructions Python environnantes, bien entendu !)

La seconde ligne crée une table nommée **membres**, laquelle contiendra des enregistrements de 3 champs : le champ **age** de type « nombre entier », le champ **nom** de type « chaîne de caractères » (de longueur variable[72]) et le champ **taille**, de type « nombre réel » (à virgule flottante). Le langage SQL autorise en principe d'autres types, mais ils ne sont pas implémentés dans *Gadfly*.

Les trois lignes qui suivent sont similaires. Nous y avons mélangé majuscules et minuscules pour bien montrer que la casse n'est pas significative en SQL. Ces lignes servent à insérer trois enregistrements dans la table membres.

A ce stade des opérations, les enregistrement n'ont pas encore été transférés dans de véritables fichiers sur disque. Il est donc possible de revenir en arrière, comme nous le verrons un peu plus loin. Le transfert sur disque est activé par la méthode **commit()** de la dernière ligne d'instructions.

Connexion à une base de données existante

Supposons qu'à la suite des opérations ci-dessus, nous décidions de terminer le script, ou même d'éteindre l'ordinateur. Comment devrons-nous procéder par la suite pour accéder à nouveau à notre base de données ?

L'accès à une base de données existante ne nécessite que deux lignes de code :

```
import gadfly
baseDonn = gadfly.gadfly("mydata","E:/Python/essais/gadfly")
```

Ces deux lignes suffisent en effet pour transférer en mémoire vive les tables contenues dans les fichiers enregistrés sur disque. La base de données peut désormais être interrogée et modifiée :

```
cur = baseDonn.cursor()
cur.execute("select * from membres")
print cur.pp()
```

[72] Veuillez noter qu'en SQL, les chaînes de caractères doivent être délimitées par des apostrophes. Si vous souhaitez que la chaîne contienne elle-même une ou plusieurs apostrophes, il vous suffit de doubler celles-ci.

La première de ces trois lignes ouvre un curseur. La requête émise dans la seconde ligne demande la sélection d'un ensemble d'enregistrements, qui seront transférés de la base de données au curseur. Dans le cas présent, la sélection n'en n'est pas vraiment une : on y demande en effet d'extraire tous les enregistrements de la table membres (le symbole ***** est fréquemment utilisé en informatique avec la signification « tout » ou « tous »).

La méthode **pp()** utilisée sur le curseur, dans la troisième ligne, provoque un affichage de tout ce qui est contenu dans le curseur sous une forme pré-formatée (les données présentes sont automatiquement disposées en colonnes). « **pp** » doit en effet être compris comme « *pretty print* ».

Si vous préférez contrôler vous-même la mise en page des informations, il vous suffit d'utiliser à sa place la méthode **fetchall()** , laquelle renvoie une liste de tuples. Essayez par exemple :

```
for x in cur.fetchall():
    print x, x[0], x[1], x[2]
```

Vous pouvez bien entendu ajouter des enregistrements supplémentaires :

```
cur.execute("Insert Into Membres(Age, Nom, Taille) Values (19,'Ricard',1.75)")
```

Pour modifier un ou plusieurs enregistrements, exécutez une requête du type :

```
cur.execute("update membres set nom ='Gerart' where nom='Ricard'")
```

Pour supprimer un ou plusieurs enregistrements, utilisez une requête telle que :

```
cur.execute("delete from membres where nom='Gerart'")
```

Si vous effectuez toutes ces opérations à la ligne de commande de Python, vous pouvez en observer le résultat à tout moment en effectuant un « *pretty print* » comme expliqué plus haut. Étant donné que toutes les modifications apportées au curseur se passent en mémoire vive, rien n'est enregistré définitivement tant que vous n'exécutez pas l'instruction **baseDonn.commit()**.

Vous pouvez donc annuler toutes les modifications apportées depuis le **commit()** précédent, en refermant la connexion à l'aide de l'instruction :

```
baseDonn.close()
```

Recherches dans une base de données

16 *Exercice*

16.1 Avant d'aller plus loin, et à titre d'exercice de synthèse, nous allons vous demander de créer entièrement vous-même une base de données « Musique » qui contiendra les deux tables suivantes (Cela représente un certain travail, mais il faut que vous puissiez disposer d'un certain nombre de données pour pouvoir expérimenter les fonctions de recherche et de tri) :

Oeuvres
comp (chaîne)
titre (chaîne)
duree (entier)
interpr (chaîne)

Compositeurs
comp (chaîne)
a_naiss (entier)
a_mort (entier)

Commencez à remplir la table **Compositeurs** avec les données qui suivent (... et profitez de cette occasion pour faire la preuve des compétences que vous maîtrisez déjà, en écrivant un petit script pour vous faciliter l'entrée des informations : une boucle s'impose !)

```
comp          a_naiss      a_mort

Mozart        1756         1791
Beethoven     1770         1827
Handel        1685         1759
Schubert      1797         1828
Vivaldi       1678         1741
Monteverdi    1567         1643
Chopin        1810         1849
Bach          1685         1750
```

Dans la table **oeuvres**, entrez les données suivantes :

```
comp          titre                       duree    interpr

Vivaldi       Les quatre saisons           20      T. Pinnock
Mozart        Concerto piano N°12          25      M. Perahia
Brahms        Concerto violon N°2          40      A. Grumiaux
Beethoven     Sonate "au clair de lune"    14      W. Kempf
Beethoven     Sonate "pathétique"          17      W. Kempf
Schubert      Quintette "la truite"        39      SE of London
Haydn         La création                 109      H. Von Karajan
Chopin        Concerto piano N°1           42      M.J. Pires
Bach          Toccata & fugue               9      P. Burmester
Beethoven     Concerto piano N°4           33      M. Pollini
Mozart        Symphonie N°40               29      F. Bruggen
Mozart        Concerto piano N°22          35      S. Richter
Beethoven     Concerto piano N°3           37      S. Richter
```

Les champs **a_naiss** et **a_mort** contiennent respectivement l'année de naissance et l'année de la mort des compositeurs. La durée des œuvres est fournie en minutes. Vous pouvez évidemment ajouter autant d'enregistrements d'œuvres et de compositeurs que vous le voulez, mais ceux qui précèdent devraient suffire pour la suite de la démonstration.

Pour ce qui va suivre, nous supposerons donc que vous avez effectivement encodé les données des deux tables décrites ci-dessus. (Si vous éprouvez des difficultés à écrire le script nécessaire, veuillez consulter le corrigé de l'exercice 16.1, à la page 357).

Le petit script ci-dessous est fourni à titre purement indicatif. Il s'agit d'un client SQL rudimentaire, qui vous permet de vous connecter à la base de données « musique » qui devrait à présent exister dans l'un de vos répertoires, d'y ouvrir un curseur et d'utiliser celui-ci pour effectuer des requêtes. Notez encore une fois que rien n'est transcrit sur le disque tant que la méthode **commit()** n'a pas été invoquée.

```
# Utilisation d'une petite base de données acceptant les requêtes SQL

import gadfly

baseDonn = gadfly.gadfly("musique","E:/Python/essais/gadfly")
cur = baseDonn.cursor()
while 1:
    print "Veuillez entrer votre requête SQL (ou <Enter> pour terminer) :"
    requete = raw_input()
    if requete =="":
        break
    try:
        cur.execute(requete)        # tentative d'exécution de la requête SQL
    except:
        print '*** Requête incorrecte ***'
    else:
        print cur.pp()              # affichage du résultat de la requête
```

```
    print

choix = raw_input("Confirmez-vous l'enregistrement (o/n) ? ")
if choix[0] == "o" or choix[0] == "O":
    baseDonn.commit()
else:
    baseDonn.close()
```

Cette application très simple n'est évidemment qu'un exemple. Il faudrait y ajouter la possibilité de choisir la base de données ainsi que le répertoire de travail. Pour éviter que le script ne se « plante » lorsque l'utilisateur encode une requête incorrecte, nous avons utilisé ici le traitement des *exceptions* déjà décrit à la page 119.

La requête select

L'une des instructions les plus puissantes du langage SQL est la requête **select**, dont nous allons à présent explorer quelques fonctionnalités. Rappelons encore une fois que nous n'abordons ici qu'une très petite partie du sujet : la description détaillée de SQL peut occuper plusieurs livres.

Lancez donc le script ci-dessus, et analysez attentivement ce qui se passe lorsque vous proposez les requêtes suivantes :

```
select *  from oeuvres

select *  from oeuvres where comp = 'Mozart'

select comp, titre, duree  from oeuvres order by comp

select titre, comp from oeuvres where comp='Beethoven' or comp='Mozart'
    order by comp

select count(*) from oeuvres

select sum(duree) from oeuvres

select avg(duree) from oeuvres

select sum(duree) from oeuvres where comp='Beethoven'

select * from oeuvres where duree >35 order by duree desc
```

Pour chacune de ces requêtes, tâchez d'exprimer le mieux possible ce qui se passe. Fondamentalement, vous activez sur la base de données des *filtres de sélection* et des *tris.*

Les requêtes suivantes sont plus élaborées, car elles concernent les deux tables à la fois.

```
select o.titre, c.nom, c.a_naiss from oeuvres o, compositeurs c
    where o.comp = c.comp

select comp from oeuvres intersect select comp from compositeurs

select comp from oeuvres except select comp from compositeurs

select comp from compositeurs except select comp from oeuvres

select distinct comp from oeuvres union select comp from compositeurs
```

Il ne nous est pas possible de développer davantage le langage de requêtes dans le cadre restreint de ces notes. Nous allons cependant examiner encore un exemple de réalisation Python faisant appel à un système de bases de données, mais en supposant cette fois qu'il s'agisse de dialoguer avec un système serveur indépendant (lequel pourrait être par exemple un gros serveur de bases de données d'entreprise, un serveur de documentation dans une école, etc.).

Ébauche d'un logiciel client pour MySQL

Pour terminer ce chapitre, nous allons vous proposer dans les pages qui suivent un exemple de réalisation concrète. Il ne s'agira pas d'un véritable logiciel (le sujet exigerait qu'on lui consacre un ouvrage spécifique), mais plutôt d'une ébauche d'analyse, destinée à vous montrer comment vous pouvez « penser comme un programmeur » lorsque vous abordez un problème complexe.

Les techniques que nous allons mettre en oeuvre ici sont de simples suggestions, dans lesquelles nous essayerons d'utiliser au mieux les outils que vous avez découverts au cours de votre apprentissage dans les chapitres précédents, à savoir : les structures de données de haut niveau (listes et dictionnaires), et la programmation par objets. Il va de soi que les options retenues dans cet exercice restent largement critiquables : vous pouvez bien évidemment traiter les mêmes problèmes en utilisant des approches différentes.

Notre objectif concret est d'arriver à réaliser rapidement un client rudimentaire, capable de dialoguer avec un « vrai » serveur de bases de données tel que *MySQL*. Nous voudrions que notre client reste un petit utilitaire très généraliste : qu'il soit capable de mettre en place une petite base de données comportant plusieurs tables, qu'il puisse servir à produire des enregistrements pour chacune d'elles, qu'il permette de tester le résultat de requêtes SQL basiques.

Dans les lignes qui suivent, nous supposerons que vous avez déjà accès à un serveur *MySQL*, sur lequel une base de données « discotheque » aura été créée pour l'utilisateur « jules », lequel s'identifie à l'aide du mot de passe « abcde ». Ce serveur peut être situé sur une machine distante accessible via un réseau, ou localement sur votre ordinateur personnel.

L'installation et la configuration d'un serveur *MySQL* sortent du cadre de cet ouvrage, mais ce n'est pas une tâche bien compliquée. C'est même fort simple si vous travaillez sous Linux, installé depuis une distribution « classique » telle que *Debian*, *Ubuntu*, *RedHat*, *SuSE* ... Il vous suffit d'installer les paquetages **MySQL-server** et **Python-MySQL**, de démarrer le service *MySQL*, puis d'entrer les commandes :

```
mysqladmin -u root password xxxx
```

Cette première commande définit le mot de passe de l'administrateur principal de *MySQL*. Elle doit être exécutée par l'administrateur du système Linux (*root*), avec un mot de passe de votre choix. On se connecte ensuite au serveur sous le compte administrateur ainsi défini (le mot de passe sera demandé) :

```
mysql -u root mysql -p
grant all privileges on *.* to jules@localhost identified by 'abcde';
grant all privileges on *.* to jules@"%" identified by 'abcde';
\q
```

Ces commandes définissent un nouvel utilisateur « jules » pour le système *MySQL*, et cet utilisateur devra se connecter le mot de passe « abcde » (Les deux lignes autorisent respectivement l'accès local et l'accès via réseau). Le nom d'utilisateur est quelconque : il ne doit pas nécessairement correspondre à un utilisateur système.

L'utilisateur « jules » peut à présent se connecter et créer des bases de données :

```
mysql -u jules -p
create database discotheque;
\q
... etc.
```

À ce stade, le serveur *MySQL* est prêt à dialoguer avec le client Python décrit dans ces pages.

Décrire la base de données dans un dictionnaire d'application

Une application dialoguant avec une base de données est presque toujours une application complexe. Elle comporte donc de nombreuses lignes de code, qu'il s'agit de structurer le mieux possible en les regroupant dans des classes (ou au moins des fonctions) bien encapsulées.

En de nombreux endroits du code, souvent fort éloignés les uns des autres, des blocs d'instructions doivent prendre en compte la structure de la base de données, c'est-à-dire son découpage en un certain nombre de tables et de champs, ainsi que les relations qui établissent une hiérarchie dans les enregistrements.

Or, l'expérience montre que la structure d'une base de données est rarement définitive. Au cours d'un développement, on réalise souvent qu'il est nécessaire de lui ajouter ou de lui retirer des champs, parfois même de remplacer une table mal conçue par deux autres, etc. Il n'est donc pas prudent de programmer des portions de code trop spécifiques d'une structure particulière, « en dur ».

Au contraire, il est hautement recommandable *de décrire plutôt la structure complète de la base de données en un seul endroit du programme*, et d'utiliser ensuite cette description comme référence pour la génération semi-automatique des instructions particulières concernant telle table ou tel champ. On évite ainsi, dans une large mesure, le cauchemar de devoir traquer et modifier un grand nombre d'instructions un peu partout dans le code, chaque fois que la structure de la base de données change un tant soit peu. Au lieu de cela, il suffit de changer seulement la description de référence, et la plus grosse partie du code reste correcte sans nécessiter de modification.

> *Nous tenons là une idée maîtresse pour réaliser des applications robustes : un logiciel destiné au traitement de données devrait toujours être construit sur la base d'un dictionnaire d'application.*

Ce que nous entendons ici par « dictionnaire d'application » ne doit pas nécessairement revêtir la forme d'un dictionnaire Python. N'importe quelle structure de données peut convenir, l'essentiel étant de se construire *une référence centrale décrivant les données* que l'on se propose de manipuler, avec peut-être aussi un certain nombre d'informations concernant leur mise en forme.

Du fait de leur capacité à rassembler en une même entité des données de n'importe quel type, les listes, tuples et dictionnaires de Python conviennent parfaitement pour ce travail. Dans l'exemple des pages suivantes, nous avons utilisé nous-mêmes un dictionnaire, dont les valeurs sont des listes de tuples, mais vous pourriez tout aussi bien opter pour une organisation différente des mêmes informations.

Tout cela étant bien établi, il nous reste encore à régler une question d'importance : où allons-nous installer concrètement ce dictionnaire d'application ?

Ses informations devront pouvoir être consultées depuis n'importe quel endroit du programme. Il semble donc obligatoire de l'installer dans une variable globale, de même d'ailleurs que d'autres données nécessaires au fonctionnement de l'ensemble de notre logiciel. Or vous savez que l'utilisation de variables globales n'est pas recommandée : elle comporte des risques, qui augmentent avec la taille du programme. De toute façon, les variables dites globales ne sont en fait globales qu'à l'intérieur d'un même module. Si nous souhaitons organiser notre logiciel comme un ensemble de modules (ce qui constitue par ailleurs une excellente pratique), nous n'aurons accès à nos variables globales que dans un seul d'entre eux.

Pour résoudre ce petit problème, il existe cependant une solution simple et élégante : *regrouper dans une classe particulière toutes les variables qui nécessitent un statut global* pour l'ensemble de l'application. Ainsi encapsulées dans l'espace de noms d'une classe, ces variables peuvent être utili-

sées sans problème dans n'importe quel module : il suffit en effet que celui-ci importe la classe en question. De plus, l'utilisation de cette technique entraîne une conséquence intéressante : le caractère « global » des variables définies de cette manière apparaît très clairement dans leur nom qualifié, puisque ce nom commence par celui de la classe contenante.

Si vous choisissez, par exemple, un nom explicite tel que **Glob** pour la classe destinée à accueillir vos variables « globales », vous vous assurez de devoir faire référence à ces variables partout dans votre code avec des noms tout aussi explicites tels que **Glob.ceci** , **Glob.cela** , etc[73].

C'est cette technique que vous allez découvrir à présent dans les premières lignes de notre script. Nous y définissons effectivement une classe **Glob()**, qui n'est donc rien d'autre qu'un simple conteneur. Aucun objet ne sera instancié à partir de celle classe, laquelle ne comporte d'ailleurs aucune méthode. Nos variables « globales » y sont définies comme de simples variables de classe, et nous pourrons donc y faire référence dans tout le reste du programme en tant qu'attributs de **Glob()**. Le nom de la base de données, par exemple, pourra être retrouvé partout dans la variable **Glob.dbName** ; le nom ou l'adresse IP du serveur dans la variable **Glob.host**, etc. :

```
 1# class Glob(object):
 2#     """Espace de noms pour les variables et fonctions <pseudo-globales>"""
 3#
 4#     dbName = "discotheque"      # nom de la base de données
 5#     user = "jules"              # propriétaire ou utilisateur
 6#     passwd = "abcde"            # mot de passe d'accès
 7#     host = "192.168.0.235"      # nom ou adresse IP du serveur
 8#
 9#     # Structure de la base de données.  Dictionnaire des tables & champs :
10#     dicoT ={"compositeurs":[('id_comp', "k", "clé primaire"),
11#                             ('nom', 25, "nom"),
12#                             ('prenom', 25, "prénom"),
13#                             ('a_naiss', "i", "année de naissance"),
14#                             ('a_mort', "i", "année de mort")],
15#             "oeuvres":[('id_oeuv', "k", "clé primaire"),
16#                        ('id_comp', "i", "clé compositeur"),
17#                        ('titre', 50, "titre de l'oeuvre"),
18#                        ('duree', "i", "durée (en minutes)"),
19#                        ('interpr', 30, "interprète principal")]}
```

Le dictionnaire d'application décrivant la structure de la base de données est contenu dans la variable **Glob.dicoT**.

Il s'agit d'un dictionnaire Python, dont les clés sont les noms des tables. Quant aux valeurs, chacune d'elles est une liste contenant la description de tous les champs de la table, sous la forme d'autant de tuples.

Chaque tuple décrit donc un champ particulier de la table. Pour ne pas encombrer notre exercice, nous avons limité cette description à trois informations seulement : le nom du champ, son type et un bref commentaire. Dans une véritable application, il serait judicieux d'ajouter encore d'autres informations ici, concernant par exemple des valeurs limites éventuelles pour les données de ce champ, le formatage à leur appliquer lorsqu'il s'agit de les afficher à l'écran ou de les imprimer, le texte qu'il faut placer en haut de colonne lorsque l'on veut les présenter dans un tableau, etc.

Il peut vous paraître assez fastidieux de décrire ainsi très en détail la structure de vos données, alors que vous voudriez probablement commencer tout de suite une réflexion sur les divers algorithmes à mettre en oeuvre afin de les traiter. Sachez cependant que si elle est bien faite, une telle description structurée vous fera certainement gagner beaucoup de temps par la suite, parce qu'elle vous permettra d'automatiser pas mal de choses. Vous en verrez une démonstration un

[73] Vous pourriez également placer vos variables « globales » dans un module nommé Glob.py, puis importer celui-ci. Utiliser un module ou une classe comme espace de noms pour stocker des variables sont donc des techniques assez similaires. L'utilisation d'une classe est peut-être un peu plus souple et plus lisible, puisque la classe peut accompagner le reste du script, alors qu'un module est nécessairement un fichier distinct.

peu plus loin. En outre, vous devez vous convaincre que cette tâche un peu ingrate vous prépare à bien structurer aussi le reste de votre travail : organisation des formulaires, tests à effectuer, etc.

Définir une classe d'objets-interfaces

La classe **Glob()** décrite à la rubrique précédente sera donc installée en début de script, ou bien dans un module séparé importé en début de script. Pour la suite de l'exposé, nous supposerons que c'est cette dernière formule qui est retenue : nous avons sauvegardé la classe **Glob()** dans un module nommé **dict_app.py**, d'où nous pouvons à présent l'importer dans le script suivant.

Ce nouveau script définit une classe d'objets-interfaces. Nous voulons en effet essayer de mettre à profit ce que nous avons appris dans les chapitres précédents, et donc privilégier la programmation par objets, afin de créer des portions de code bien encapsulées et largement réutilisables.

Les objets-interfaces que nous voulons construire seront similaires aux objets-fichiers que nous avons abondamment utilisés pour la gestion des fichiers au chapitre 9. Vous vous rappelez par exemple que nous ouvrons un fichier en créant un objet-fichier, à l'aide de la fonction-fabrique **open()**. D'une manière similaire, nous ouvrirons la communication avec la base de données en commençant par créer un objet-interface à l'aide de la classe **GestionBD()**, ce qui établira la connexion. Pour lire ou écrire dans un fichier ouvert, nous utilisons diverses méthodes de l'objet-fichier. D'une manière analogue, nous effectuerons nos opérations sur la base de données par l'intermédiaire des diverses méthodes de l'objet-interface.

```
 1# import MySQLdb, sys
 2# from dict_app import *
 3#
 4# class GestionBD:
 5#     """Mise en place et interfaçage d'une base de données MySQL"""
 6#     def __init__(self, dbName, user, passwd, host, port =3306):
 7#         "Établissement de la connexion - Création du curseur"
 8#         try:
 9#             self.baseDonn = MySQLdb.connect(db =dbName,
10#                   user =user, passwd =passwd, host =host, port =port)
11#         except Exception, err:
12#             print 'La connexion avec la base de données a échoué :\n'\
13#                   'Erreur détectée :\n%s' % err
14#             self.echec =1
15#         else:
16#             self.cursor = self.baseDonn.cursor()   # création du curseur
17#             self.echec =0
18#
19#     def creerTables(self, dicTables):
20#         "Création des tables décrites dans le dictionnaire <dicTables>."
21#         for table in dicTables:            # parcours des clés du dict.
22#             req = "CREATE TABLE %s (" % table
23#             pk =''
24#             for descr in dicTables[table]:
25#                 nomChamp = descr[0]        # libellé du champ à créer
26#                 tch = descr[1]             # type de champ à créer
27#                 if tch =='i':
28#                     typeChamp ='INTEGER'
29#                 elif tch =='k':
30#                     # champ 'clé primaire' (incrémenté automatiquement)
31#                     typeChamp ='INTEGER AUTO_INCREMENT'
32#                     pk = nomChamp
33#                 else:
34#                     typeChamp ='VARCHAR(%s)' % tch
35#                 req = req + "%s %s, " % (nomChamp, typeChamp)
36#             if pk == '':
37#                 req = req[:-2] + ")"
38#             else:
39#                 req = req + "CONSTRAINT %s_pk PRIMARY KEY(%s))" % (pk, pk)
40#             self.executerReq(req)
```

```
41#
42#     def supprimerTables(self, dicTables):
43#         "Suppression de toutes les tables décrites dans <dicTables>"
44#         for table in dicTables.keys():
45#             req ="DROP TABLE %s" % table
46#             self.executerReq(req)
47#         self.commit()                       # transfert -> disque
48#
49#     def executerReq(self, req):
50#         "Exécution de la requête <req>, avec détection d'erreur éventuelle"
51#         try:
52#             self.cursor.execute(req)
53#         except Exception, err:
54#             # afficher la requête et le message d'erreur système :
55#             print "Requête SQL incorrecte :\n%s\nErreur détectée :\n%s"\
56#                    % (req, err)
57#             return 0
58#         else:
59#             return 1
60#
61#     def resultatReq(self):
62#         "renvoie le résultat de la requête précédente (un tuple de tuples)"
63#         return self.cursor.fetchall()
64#
65#     def commit(self):
66#         if self.baseDonn:
67#             self.baseDonn.commit()         # transfert curseur -> disque
68#
69#     def close(self):
70#         if self.baseDonn:
71#             self.baseDonn.close()
```

Commentaires :

- Lignes 1-2 : Outre notre propre module **dict_app** qui contient les variables « globales », nous importons le module **sys** qui contient quelques fonctions système, et le module **MySQLdb** qui contient tout ce qui est nécessaire pour communiquer avec MySQL. Rappelons que ce module ne fait pas partie de la distribution standard de Python, et qu'il doit donc être installé séparément.
- Ligne 5 : Lors de la création des objets-interfaces, nous devrons fournir les paramètres de la connexion : nom de la base de données, nom de son utilisateur, nom ou adresse IP de la machine où est situé le serveur. Le n° du port de communication est habituellement celui que nous avons prévu par défaut. Toutes ces informations sont supposées être en votre possession.
- Lignes 8 à 17 : Il est hautement recommandable de placer le code servant à établir la connexion à l'intérieur d'un gestionnaire d'exceptions **try**-**except**-**else** (voir page 119), car nous ne pouvons pas présumer que le serveur sera nécessairement accessible. Remarquons au passage que la méthode **__init__()** ne peut pas renvoyer de valeur (à l'aide de l'instruction **return**), du fait qu'elle est invoquée automatiquement par Python lors de l'instanciation d'un objet. En effet : ce qui est renvoyé dans ce cas au programme appelant est l'objet nouvellement construit. Nous ne pouvons donc pas signaler la réussite ou l'échec de la connexion au programme appelant à l'aide d'une valeur de retour. Une solution simple à ce petit problème consiste à mémoriser le résultat de la tentative de connexion dans un attribut d'instance (variable **self.echec**), que le programme appelant peut ensuite tester quand bon lui semble.
- Lignes 19 à 40 : Cette méthode automatise la création de toutes les tables de la base de données, en tirant profit de la description du dictionnaire d'application, lequel doit lui être transmis en argument. Une telle automatisation sera évidemment d'autant plus appréciable, que la structure de la base de données sera plus complexe (Imaginez par exemple une base de don-

nées contenant 35 tables !). Afin de ne pas alourdir la démonstration, nous avons restreint les capacités de cette méthode à la création de champs des types *integer* et *varchar*. Libre à vous d'ajouter les instructions nécessaires pour créer des champs d'autres types.

Si vous détaillez le code, vous constaterez qu'il consiste simplement à construire une requête SQL pour chaque table, morceau par morceau, dans la chaîne de caractères **req**. Celle-ci est ensuite transmise à la méthode **executerReq()** pour exécution. Si vous souhaitez visualiser la requête ainsi construite, vous pouvez évidemment ajouter une instruction `print req` juste après la ligne 40.

Vous pouvez également ajouter à cette méthode la capacité de mettre en place les *contraintes d'intégrité référentielle*, sur la base d'un complément au dictionnaire d'application qui décrirait ces contraintes. Nous ne développons pas cette question ici, mais cela ne devrait pas vous poser de problème si vous savez de quoi il retourne.

- Lignes 42 à 47 : Beaucoup plus simple que la précédente, cette méthode utilise le même principe pour supprimer toutes les tables décrites dans le dictionnaire d'application.
- Lignes 49 à 59 : Cette méthode transmet simplement la requête à l'objet curseur. Son utilité est de simplifier l'accès à celui-ci et de produire un message d'erreur si nécessaire.
- Lignes 61 à 71 : Ces méthodes ne sont que de simples relais vers les objets produits par le module **MySQLdb** : l'objet-connecteur produit par la fonction-fabrique **MySQLdb.connect()**, et l'objet curseur correspondant. Elles permettent de simplifier légèrement le code du programme appelant.

Construire un générateur de formulaires

Nous avons ajouté cette classe à notre exercice pour vous expliquer comment vous pouvez utiliser le même dictionnaire d'application afin d'élaborer du code généraliste. L'idée développée ici est de réaliser une classe d'objets-formulaires capables de prendre en charge l'encodage des enregistrements de n'importe quelle table, en construisant automatiquement les instructions d'entrée adéquates grâce aux informations tirées du dictionnaire d'application.

Dans une application véritable, ce formulaire trop simpliste devrait certainement être fortement remanié, et il prendrait vraisemblablement la forme d'une fenêtre spécialisée, dans laquelle les champs d'entrée et leurs libellés pourraient encore une fois être générés de manière automatique. Nous ne prétendons donc pas qu'il constitue un bon exemple, mais nous voulons simplement vous montrer comment vous pouvez automatiser sa construction dans une large mesure. Tâchez de réaliser vos propres formulaires en vous servant de principes semblables.

```
 1#  class Enregistreur:
 2#      """classe pour gérer l'entrée d'enregistrements divers"""
 3#      def __init__(self, bd, table):
 4#          self.bd =bd
 5#          self.table =table
 6#          self.descriptif =Glob.dicoT[table]   # descriptif des champs
 7#
 8#      def entrer(self):
 9#          "procédure d'entrée d'un enregistrement entier"
10#          champs ="("           # ébauche de chaîne pour les noms de champs
11#          valeurs ="("          # ébauche de chaîne pour les valeurs
12#          # Demander successivement une valeur pour chaque champ :
13#          for cha, type, nom in self.descriptif:
14#              if type =="k":    # on ne demandera pas le n° d'enregistrement
15#                  continue      # à l'utilisateur (numérotation auto.)
16#              champs = champs + cha + ","
```

```
17#             val = raw_input("Entrez le champ %s :" % nom)
18#             if type =="i":
19#                 valeurs = valeurs + val +","
20#             else:
21#                 valeurs = valeurs + "'%s'," % (val)
22#
23#         champs = champs[:-1] + ")"    # supprimer la dernière virgule,
24#         valeurs = valeurs[:-1] + ")"  # ajouter une parenthèse
25#         req ="INSERT INTO %s %s VALUES %s" % (self.table, champs, valeurs)
26#         self.bd.executerReq(req)
27#
28#         ch =raw_input("Continuer (O/N) ? ")
29#         if ch.upper() == "O":
30#             return 0
31#         else:
32#             return 1
```

Commentaires :

- Lignes 1 à 6 : Au moment de leur instanciation, les objets de cette classe reçoivent la référence de l'une des tables du dictionnaire. C'est ce qui leur donne accès au descriptif des champs.
- Ligne 8 : Cette méthode **entrer()** génère le formulaire proprement dit. Elle prend en charge l'entrée des enregistrements dans la table, en s'adaptant à leur structure propre grâce au descriptif trouvé dans le dictionnaire.
  Sa fonctionnalité concrète consiste encore une fois à construire morceau par morceau une chaîne de caractères qui deviendra une requête SQL, comme dans la méthode **creerTables()** de la classe **GestionBD()** décrite à la rubrique précédente.

  Vous pourriez bien entendu ajouter à la présente classe encore d'autres méthodes, pour gérer par exemple la suppression et/ou la modification d'enregistrements.
- Lignes 12 à 21 : L'attribut d'instance **self.descriptif** contient une liste de tuples, et chacun de ceux-ci est fait de trois éléments, à savoir le nom d'un champ, le type de données qu'il est censé recevoir, et sa description « en clair ». La boucle **for** de la ligne 13 parcourt cette liste et affiche pour chaque champ un message d'invite construit sur la base de la description qui accompagne ce champ. Lorsque l'utilisateur a entré la valeur demandée, celle-ci et formatée dans une chaîne en construction. Le formatage s'adapte aux conventions du langage SQL, conformément au type requis pour le champ.
- Lignes 23 à 26 : Lorsque tous les champs ont été parcourus, la requête proprement dite est assemblée et exécutée. Si vous souhaitez visualiser cette requête, vous pouvez bien évidemment ajouter une instruction `print req` juste après la ligne 25.

Le corps de l'application

Il ne nous paraît pas utile de développer davantage encore cet exercice dans le cadre d'un manuel d'initiation. Si le sujet vous intéresse, vous devriez maintenant en savoir assez pour commencer déjà quelques expériences personnelles. Veuillez alors consulter les bons ouvrages de référence, comme par exemple *Python : How to program* de Deitel & coll., ou encore les sites web consacrés aux extensions de Python.

Le script qui suit est celui d'une petite application destinée à tester les classes décrites dans les pages qui précèdent. Libre à vous de la perfectionner, ou alors d'en écrire une autre tout à fait différente !

```
1#  ###### Programme principal : #########
2#
```

```
 3#  # Création de l'objet-interface avec la base de données :
 4#  bd = GestionBD(Glob.dbName, Glob.user, Glob.passwd, Glob.host)
 5#  if bd.echec:
 6#      sys.exit()
 7#
 8#  while 1:
 9#      print "\nQue voulez-vous faire :\n"\
10#            "1) Créer les tables de la base de données\n"\
11#            "2) Supprimer les tables de la base de données ?\n"\
12#            "3) Entrer des compositeurs\n"\
13#            "4) Entrer des oeuvres\n"\
14#            "5) Lister les compositeurs\n"\
15#            "6) Lister les oeuvres\n"\
16#            "7) Exécuter une requête SQL quelconque\n"\
17#            "9) terminer ?                          Votre choix :",
18#      ch = int(raw_input())
19#      if ch ==1:
20#          # création de toutes les tables décrites dans le dictionnaire :
21#          bd.creerTables(Glob.dicoT)
22#      elif ch ==2:
23#          # suppression de toutes les tables décrites dans le dic. :
24#          bd.supprimerTables(Glob.dicoT)
25#      elif ch ==3 or ch ==4:
26#          # création d'un <enregistreur> de compositeurs ou d'oeuvres :
27#          table ={3:'compositeurs', 4:'oeuvres'}[ch]
28#          enreg =Enregistreur(bd, table)
29#          while 1:
30#              if enreg.entrer():
31#                  break
32#      elif ch ==5 or ch ==6:
33#          # listage de tous les compositeurs, ou toutes les oeuvres :
34#          table ={5:'compositeurs', 6:'oeuvres'}[ch]
35#          if bd.executerReq("SELECT * FROM %s" % table):
36#              # analyser le résultat de la requête ci-dessus :
37#              records = bd.resultatReq()      # ce sera un tuple de tuples
38#              for rec in records:             # => chaque enregistrement
39#                  for item in rec:            # => chaque champ dans l'enreg.
40#                      print item,
41#                  print
42#      elif ch ==7:
43#          req =raw_input("Entrez la requête SQL : ")
44#          if bd.executerReq(req):
45#              print bd.resultatReq()          # ce sera un tuple de tuples
46#      else:
47#          bd.commit()
48#          bd.close()
49#          break
```

Commentaires :

- On supposera bien évidemment que les classes décrites plus haut soient présentes dans le même script, ou qu'elles aient été importées.
- Lignes 3 à 6 : L'objet-interface est créé ici. Si la création échoue, l'attribut d'instance **bd.echec** contient la valeur 1. Le test des lignes 5 et 6 permet alors d'arrêter l'application immédiatement (la fonction **exit()** du module **sys** sert spécifiquement à cela).
- Ligne 8 : Le reste de l'application consiste à proposer sans cesse le même menu, jusqu'à ce que l'utilisateur choisisse l'option n° 9.
- Lignes 27 et 28 : La classe **Enregistreur()** accepte de gérer les enregistrements de n'importe quelle table. Afin de déterminer laquelle doit être utilisée lors de l'instanciation, on utilise un petit dictionnaire qui indique quel nom retenir, en fonction du choix opéré par l'utilisateur (option n° 3 ou n° 4).

- Lignes 29 à 31 : La méthode **entrer()** de l'objet-enregistreur renvoie une valeur 0 ou 1 suivant que l'utilisateur ait choisi de continuer à entrer des enregistrements, ou bien d'arrêter. Le test de cette valeur permet d'interrompre la boucle de répétition en conséquence.
- Lignes 35 et 44 : La méthode **executerReq**() renvoie une valeur 0 ou 1 suivant que la requête ait été acceptée ou non par le serveur. On peut donc tester cette valeur pour décider si le résultat doit être affiché ou non.

Exercices :

16.2 Modifiez le script décrit dans ces pages de manière à ajouter une table supplémentaire à la base de données. Ce pourrait être par exemple une table « orchestres », dont chaque enregistrement contiendrait le nom de l'orchestre, le nom de son chef, et le nombre total d'instruments.

16.3 Ajoutez d'autres types de champ à l'une des tables (par exemple un champ de type *float* (réel) ou de type *date*), et modifiez le script en conséquence.

17

Applications web

Vous avez certainement déjà appris par ailleurs un grand nombre de choses concernant la rédaction de *pages web*. Vous savez que ces pages sont des documents au format HTML, que l'on peut consulter via un réseau (intranet ou internet) à l'aide d'un logiciel appelé *browser web* ou *navigateur* (Netscape, Konqueror, Internet explorer, ...).

Les pages HTML sont installées dans les répertoires publics d'un autre ordinateur où fonctionne en permanence un logiciel appelé *serveur web* (Apache, IIS, Zope, ...). Lorsqu'une connexion a été établie entre cet ordinateur et le vôtre, votre logiciel navigateur peut dialoguer avec le logiciel serveur (par l'intermédiaire de toute une série de dispositifs matériels et logiciels dont nous ne traiterons pas ici : lignes téléphoniques, routeurs, caches, protocoles de communication ...).

Le protocole HTTP qui gère la transmission des pages web autorise l'échange de données dans les deux sens. Mais dans la grande majorité des cas, le transfert d'informations n'a pratiquement lieu que dans un seul, à savoir du serveur vers le navigateur : des textes, des images, des fichiers divers lui sont expédiés en grand nombre (ce sont les pages consultées) ; en revanche, le navigateur n'envoie guère au serveur que de toutes petites quantités d'information : essentiellement les adresses URL des pages que l'internaute désire consulter.

Pages web interactives

Vous savez cependant qu'il existe des sites web où vous êtes invité à fournir vous-même des quantités d'information plus importantes : vos références personnelles pour l'inscription à un club ou la réservation d'une chambre d'hôtel, votre numéro de carte de crédit pour la commande d'un article sur un site de commerce électronique, votre avis ou vos suggestions, etc.

Dans un cas comme ceux-là, vous vous doutez bien que l'information transmise doit être prise en charge, du côté du serveur, par un programme spécifique. Il faut donc que les pages web destinées à accueillir cette information soient dotées d'un mécanisme assurant son transfert vers le logiciel destiné à la traiter. Il faudra également que ce logiciel puisse lui-même transmettre en retour une information au serveur, afin que celui-ci puisse présenter le résultat de l'opération à l'internaute, sous la forme d'une nouvelle page web.

Le but du présent chapitre est de vous expliquer comment vous pouvez vous servir de vos compétences de programmeur Python pour ajouter une telle interactivité à un site web, en y intégrant de véritables applications.

Remarque importante : Ce que nous allons expliquer dans les paragraphes qui suivent sera directement fonctionnel sur l'intranet de votre établissement scolaire ou de votre entreprise (à la condition toutefois que l'administrateur de cet intranet ait configuré son serveur de manière appropriée). En ce qui concerne l'internet, par contre, les choses sont un peu plus compliquées. Il va de soi que l'installation de logiciels sur un ordinateur serveur relié à l'internet ne peut se faire qu'avec l'accord de son propriétaire. Si un fournisseur d'accès à l'internet a mis a votre disposi-

tion un certain espace où vous êtes autorisé à installer des pages web « statiques » (c'est-à-dire de simples documents à consulter), cela ne signifie pas pour autant que vous pourrez y faire fonctionner des scripts Python. Pour que cela puisse marcher, vous devrez demander une autorisation et un certain nombre de renseignements à votre fournisseur d'accès. Il faudra en particulier lui demander si vous pouvez activer des scripts CGI écrits en Python à partir de vos pages, et dans quel(s) répertoire(s) vous pouvez les installer.

L'interface CGI

L'interface CGI (*Common Gateway Interface*) est un composant de la plupart des logiciels serveurs de pages web. Il s'agit d'une passerelle qui leur permet de communiquer avec d'autres logiciels tournant sur le même ordinateur. Avec CGI, vous pouvez écrire des scripts dans différents langages (Perl, C, Tcl, Python ...).

Plutôt que de limiter le web à des documents écrits à l'avance, CGI permet de générer des pages web sur le champ, en fonction des données que fournit l'internaute par l'intermédiaire de son logiciel de navigation. Vous pouvez utiliser les scripts CGI pour créer une large palette d'applications : des services d'inscription en ligne, des outils de recherche dans des bases de données, des instruments de sondage d'opinions, des jeux, etc.

L'apprentissage de la programmation CGI peut faire l'objet de manuels entiers. Dans cet ouvrage d'initiation, nous vous expliquerons seulement quelques principes de base, afin de vous faire comprendre, par comparaison, l'énorme avantage que présentent les modules serveurs d'applications spécialisés tels que Karrigell, CherryPy, Django ou Zope, pour le programmeur désireux de développer un site web interacif.

Une interaction CGI rudimentaire

Pour la bonne compréhension de ce qui suit, nous supposerons que l'administrateur réseau de votre établissement scolaire ou de votre entreprise a configuré un serveur web d'intranet d'une manière telle que vous puissiez installer des pages HTML et des scripts Python dans un répertoire personnel.

Notre premier exemple sera constitué d'une page web extrêmement simple. Nous n'y placerons qu'un seul élément d'interactivité, à savoir un unique bouton. Ce bouton servira à lancer l'exécution d'un petit programme que nous décrirons par après.

Veuillez donc encoder le document HTML ci-dessous à l'aide d'un éditeur quelconque (n'encodez pas les numéros de lignes, ils ne sont là que pour faciliter les explications qui suivent) :

```
 1# <HTML>
 2# <HEAD><TITLE>Exercice avec Python</TITLE></HEAD>
 3# <BODY>
 4#
 5# <DIV ALIGN="center">
 6# <IMG SRC="penguin.gif">
 7# <H2>Page Web interactive</H2>
 8# <P>Cette page est associée à un script Python</P>
 9#
10# <FORM ACTION="http://Serveur/cgi-bin/input_query.py" METHOD="post">
11# <INPUT TYPE="submit" NAME="send" VALUE="Exécuter le script">
12# </FORM>
13#
14# </DIV></BODY></HTML>
```

- Vous savez certainement déjà que les balises initiales <**HTML**>, <**HEAD**>, <**TITLE**>, <**BODY**>, ainsi que les balises finales correspondantes, sont communes à tous les documents HTML. Nous ne détaillerons donc pas leur rôle ici.
- La balise <**DIV**> utilisée à la ligne 5 sert habituellement à diviser un document HTML en sections distinctes. Nous l'utilisons ici pour définir une section dans laquelle tous les éléments seront centrés (horizontalement) sur la page.
- A la ligne 6, nous insérons une petite image.
- La ligne 7 définit une ligne te texte comme étant un titre de 2e importance.
- La ligne 8 est un paragraphe ordinaire.
- Les lignes 10 à 12 contiennent le code important (pour ce qui nous occupe ici). Les balises <**FORM**> et </**FORM**> définissent en effet un *formulaire*, c'est-à-dire une portion de page Web susceptible de contenir divers widgets à l'aide desquels l'internaute pourra exercer une certaine activité : champs d'entrée, boutons, cases à cocher, boutons radio, etc.

La balise **FORM** doit contenir deux indications essentielles : *l'action à accomplir* lorsque le formulaire sera expédié (il s'agit en fait de fournir ici l'adresse URL du logiciel à invoquer pour traiter les données transmises), et *la méthode à utiliser* pour transmettre l'information (en ce qui nous concerne, ce sera toujours la méthode **POST**).

Dans notre exemple, le logiciel que nous voulons invoquer est un script Python nommé **input_query.py** qui est situé dans un répertoire particulier du serveur d'intranet. Sur de nombreux serveurs, ce répertoire s'appelle souvent **cgi-bin** , par pure convention. Nous supposerons ici que l'administrateur de votre intranet scolaire vous autorise à installer vos scripts Python dans le même répertoire que celui où vous placez vos pages web personnelles.

Vous devrez donc modifier la ligne 10 de notre exemple, en remplaçant l'adresse **`http://Serveur/cgi-bin/input_query.py`** par ce que votre professeur vous indiquera[74].

La ligne 11 contient la balise qui définit un widget de type « bouton d'envoi » (balise **`<INPUT TYPE="submit">`**). Le texte qui doit apparaître sur le bouton est précisé par l'attribut **`VALUE ="texte"`**. L'indication **`NAME`** est facultative dans le cas présent. Elle mentionne le nom du widget lui-même (au cas où le logiciel destinataire en aurait besoin).

Lorsque vous aurez terminé l'encodage de ce document, sauvegardez-le dans le répertoire que l'on vous a attribué spécifiquement pour y placer vos pages, sous un nom quelconque, mais de préférence avec l'extension **.html** ou **.htm** (par exemple : **essai.html**).

Le script Python input_query.py est détaillé ci-dessous. Comme déjà signalé plus haut, vous pouvez installer ce script dans le même répertoire que votre document HTML initial :

```
 1#  #! /usr/bin/python
 2#
 3#  # Affichage d'un formulaire HTML simplifié :
 4#  print "Content-Type: text/html\n"
 5#  print """
 6#  <H3><FONT COLOR="Royal blue">
 7#  Page web produite par un script Python
 8#  </FONT></H3>
 9#
10#  <FORM ACTION="print_result.py" METHOD="post">
11#  <P>Veuillez entrer votre nom dans le champ ci-dessous, s.v.p. :</P>
12#  <P><INPUT NAME="visiteur" SIZE=20 MAXLENGTH=20 TYPE="text"></P>
13#  <P>Veuillez également me fournir une phrase quelconque :</P>
14#  <TEXTAREA NAME="phrase" ROWS=2 COLS=50>Mississippi</TEXTAREA>
```

[74] Par exemple : http://192.168.0.100/cgi/Classe6A/Dupont/input_query.py .

```
15#  <P>J'utiliserai cette phrase pour établir un histogramme.</P>
16#  <INPUT TYPE="submit" NAME="send" VALUE="Action">
17#  </FORM>
18#  """
```

Ce script ne fait rien d'autre que d'afficher une nouvelle page web, laquelle contient encore une fois un formulaire, mais celui-ci nettement plus élaboré que le précédent.

- La première ligne est absolument nécessaire : elle indique à l'interface CGI qu'il faut lancer l'interpréteur Python pour pouvoir exécuter le script. La seconde ligne est un simple commentaire.
- La ligne 4 est indispensable. Elle permet à l'interpréteur Python d'initialiser un véritable document HTML qui sera transmis au serveur web. Celui-ci pourra à son tour le réexpédier au logiciel navigateur de l'internaute, et celui-ci le verra donc s'afficher dans la fenêtre de navigation.
- La suite est du pur code HTML, traité par Python comme une simple chaîne de caractères que l'on affiche à l'aide de l'instruction **print**. Pour pouvoir y insérer tout ce que nous voulons, y compris les sauts à la ligne, les apostrophes, les guillemets, etc., nous délimitons cette chaîne de caractères à l'aide de « triples guillemets » (Rappelons également ici que les sauts à la ligne sont complètement ignorés en HTML : nous pouvons donc en utiliser autant que nous voulons pour « aérer » notre code et le rendre plus lisible).

Un formulaire HTML pour l'acquisition des données

Analysons à présent le code HTML lui-même. Nous y trouvons essentiellement un nouveau formulaire, qui comporte plusieurs paragraphes, parmi lesquels on peut reconnaître quelques widgets.

- La ligne 10 indique le nom du script CGI auquel les données du formulaire seront transmises : il s'agira bien évidemment d'un autre script Python.
- A la ligne 12, on trouve la définition d'un widget de type « champ d'entrée » (Balise **INPUT**, avec `TYPE="text"`). L'utilisateur est invité à y encoder son nom. Le paramètre **MAXLENGTH** définit une longueur maximale pour la chaîne de caractères qui sera entrée ici (20 caractères, en l'occurrence). Le paramètre **SIZE** définit la taille du champ tel qu'il doit apparaître à l'écran, et le paramètre **NAME** est le nom que nous choisissons pour la variable destinée à mémoriser la chaîne de caractères attendue.
- Un second champ d'entrée un peu différent est défini à la ligne 14 (balise **TEXTAREA**). Il s'agit d'un réceptacle plus vaste, destiné à accueillir des textes de plusieurs lignes. (Ce champ est automatiquement pourvu d'ascenseurs si le texte à insérer se révèle trop volumineux). Ses paramètres **ROWS** et **COLS** sont assez explicites. Entre les balises initiale et finale, on peut insérer un texte par défaut (`Mississippi` dans notre exemple).
- Comme dans l'exemple précédent, la ligne 16 contient la définition du bouton qu'il faudra actionner pour transmettre les données au script CGI destinataire, lequel est décrit ci-après.

Un script CGI pour le traitement des données

Le mécanisme utilisé à l'intérieur d'un script CGI pour réceptionner les données transmises par un formulaire HTML est fort simple, comme vous pouvez l'analyser dans l'exemple ci-dessous :

```
1#  #! /usr/bin/python
2#  # Traitement des données transmises par un formulaire HTML
3#
```

```
 4#  import cgi                          # Module d'interface avec le serveur web
 5#  form = cgi.FieldStorage()          # Réception de la requête utilisateur :
 6#                                     # il s'agit d'une sorte de dictionnaire
 7#  if form.has_key("phrase"):         # La clé n'existera pas si le champ
 8#     text = form["phrase"].value     # correspondant est resté vide
 9#  else:
10#     text ="*** le champ phrase était vide ! ***"
11#
12#  if form.has_key("visiteur"):       # La clé n'existera pas si le champ
13#     nomv = form["visiteur"].value   # correspondant est resté vide
14#  else:
15#     nomv ="mais vous ne m'avez pas indiqué votre nom"
16#
17#  print "Content-Type: text/html\n"
18#  print """
19#  <H3>Merci, %s !</H3>
20#  <H4>La phrase que vous m'avez fournie était : </H4>
21#  <H3><FONT Color="red"> %s </FONT></H3>""" % (nomv, text)
22#
23#  histogr ={}
24#  for c in text:
25#     histogr[c] = histogr.get(c, 0) +1
26#
27#  liste = histogr.items()        # conversion en une liste de tuples
28#  liste.sort()                   # tri de la liste
29#  print "<H4>Fréquence de chaque caractère dans la phrase :</H4>"
30#  for c, f in liste:
31#     print 'le caractère <B>"%s"</B> apparaît %s fois <BR>' % (c, f)
```

Les lignes 4 et 5 sont les plus importantes :

- Le module cgi importé à la ligne 4 assure la connexion du script Python avec l'interface CGI , laquelle permet de dialoguer avec le serveur web.
- A la ligne 5, la fonction **FieldStorage()** de ce module renvoie un objet qui contient l'ensemble des données transmises par le formulaire HTML. Nous plaçons cet objet, lequel est assez semblable à un dictionnaire classique, dans la variable form.

Par rapport à un véritable dictionnaire, l'objet placé dans **form** présente la différence essentielle qu'il faudra lui appliquer la méthode **value()** pour en extraire les données. Les autres méthodes applicables aux dictionnaires, telles la méthode **has_key()** , par exemple, peuvent être utilisées de la manière habituelle.

Une caractéristique importante de l'objet dictionnaire retourné par **FieldStorage()** est qu'il ne possédera aucune clé pour les champs laissés vides dans le formulaire HTML correspondant.

Dans notre exemple, le formulaire comporte deux champs d'entrée, auxquels nous avons associé les noms **visiteur** et **phrase**. Si ces champs ont effectivement été complétés par l'utilisateur, nous trouverons leurs contenus dans l'objet dictionnaire, aux index **visiteur** et **phrase**. Par contre, si l'un ou l'autre de ces champs n'a pas été complété, l'index correspondant n'existera tout simplement pas. Avant toute forme de traitement de valeurs, il est donc indispensable de s'assurer de la présence de chacun des index attendus, et c'est ce que nous faisons aux lignes 7 à 15.

17 *Exercice :*

17.1 Pour vérifier ce qui précède, vous pouvez par exemple désactiver (en les transformant en commentaires) les lignes 7, 9, 10, 12, 14 & 15 du script. Si vous testez le fonctionnement de l'ensemble, vous constaterez que tout se passe bien si l'utilisateur complète effectivement les champs qui lui sont proposés. Si l'un des champs est laissé vide, par contre, une erreur se produit.

> ***Note importante :*** *le script étant lancé par l'intermédiaire d'une page web, les messages d'erreur de Python ne seront pas affichés dans cette page, mais*

> *plutôt enregistrés dans le journal des événements (log) du serveur web. Veuillez consulter l'administrateur de ce serveur pour savoir comment vous pouvez accéder à ce journal. De toute manière, attendez-vous à ce que la recherche des erreurs dans un script CGI soit plus ardue que dans une application ordinaire.*

Le reste du script est assez classique.

- Aux lignes 17 à 21, nous ne faisons qu'afficher les données transmises par le formulaire. Veuillez noter que les variables nomv et text doivent exister au préalable, ce qui rend indispensables les lignes 9, 10, 14 & 15.
- Aux lignes 23, 24 & 25, nous nous servons d'un dictionnaire pour construire un histogramme simple, comme nous l'avons expliqué à la page 155.
- A la ligne 27, nous convertissons le dictionnaire résultant en une liste de tuples, pour pouvoir trier celle-ci dans l'ordre alphabétique à la ligne 28.
- La boucle for des lignes 30 et 31 se passe de commentaires.

Un serveur web en pur Python !

Dans les pages précédentes, nous vous avons expliqué quelques rudiments de programmation CGI afin que vous puissiez mieux comprendre comment fonctionne une application web. Mais si vous voulez véritablement développer une telle application (par exemple un site web personnel doté d'une certaine interactivité), vous constaterez rapidement que l'interface CGI est un outil trop sommaire. Son utilisation telle quelle dans des scripts se révèle fort lourde, et il est donc préférable de faire appel à des outils plus élaborés.

L'intérêt pour le développement web est devenu très important, et il existe donc une forte demande pour des interfaces et des environnements de programmation bien adaptés à cette tâche. Or, même s'il ne peut pas prétendre à l'universalité de langages tels que C/C++, Python est déjà largement utilisé un peu partout dans le monde pour écrire des programmes très ambitieux, y compris dans le domaine des serveurs d'applications web. La robustesse et la facilité de mise en œuvre du langage ont séduit de nombreux développeurs de talent, qui ont réalisé des outils de développement web de très haut niveau. Plusieurs de ces applications peuvent vous intéresser si vous souhaitez réaliser vous-même des sites web interactifs de différents types.

Les produits existants sont pour la plupart des logiciels libres. Ils permettent de couvrir une large gamme de besoins, depuis le petit site personnel de quelques pages, jusqu'au gros site commercial collaboratif, capable de répondre à des milliers de requêtes journalières, et dont les différents secteurs sont gérés sans interférence par des personnes de compétences variées (infographistes, programmeurs, spécialistes de bases de données, etc.).

Le plus célèbre de ces produits est le logiciel *Zope*, déjà adopté par de grands organismes privés et publics pour le développement d'intranets et d'extranets collaboratifs. Il s'agit en fait d'un système serveur d'applications, très performant, sécurisé, presqu'entièrement écrit en Python, et que l'on peut administrer à distance à l'aide d'une simple interface web. Il ne nous est pas possible de décrire l'utilisation de *Zope* dans ces pages : le sujet est trop vaste, et un livre entier n'y suffirait pas. Sachez cependant que ce produit est parfaitement capable de gérer de très gros sites d'entreprise en offrant d'énormes avantages par rapport à des solutions classiques telles que PHP ou Java.

D'autres outils moins ambitieux mais tout aussi intéressants sont disponibles. Tout comme *Zope*, la plupart d'entre eux peuvent être téléchargés librement depuis l'internet. Le fait qu'ils soient

écrits en Python assure en outre leur portabilité : vous pourrez donc les employer aussi bien sous *Windows* que sous *Linux* ou *MacOs*. Chacun d'eux peut être utilisé en conjonction avec un serveur web « classique » tel que *Apache* ou *Xitami* (c'est préférable si le site à réaliser est destiné à supporter une charge de connexions très importante), mais la plupart d'entre eux intègrent leur propre serveur web, ce qui leur permet de fonctionner également de manière tout à fait autonome. Cette possibilité se révèle particulièrement intéressante au cours de la mise au point d'un site, car elle facilite la recherche des erreurs.

Cette totale autonomie alliée à la grande facilité de leur mise en oeuvre fait de ces produits de fort bonnes solutions pour la réalisation de sites web d'intranet spécialisés, notamment dans des petites et moyennes entreprises, des administrations, ou dans des écoles. Si vous souhaitez développer une application Python qui soit accessible par l'intermédiaire d'un simple navigateur web, via un intranet d'entreprise (ou même via l'internet, mais dans ce cas il est recommandé de les faire travailler en conjonction avec *Apache* ou *Xitami*), ces applications sont faites pour vous.

Il en existe une grande variété : *Django*, *Turbogears*, *Pylons*, *Spyce*, *Karrigell*, *Webware*, *Cherrypy*, *Quixote*, *Twisted*, etc. Choisissez en fonction de vos besoins : vous n'aurez que l'embarras du choix.

Dans les lignes qui suivent, nous allons décrire une petite application web fonctionnant à l'aide de *Karrigell*. Vous pouvez trouver ce système à l'adresse : *http://karrigell.sourceforge.net.* Il s'agit d'une solution de développement web simple, fort bien documentée en anglais *et aussi en français* (son auteur, Pierre Quentel, est en effet originaire de Bretagne, tout comme le mot *karrigell*, d'ailleurs, lequel signifie « charrette »).

Installation de Karrigell

L'installation de Karrigell est un jeu d'enfant : il vous suffit d'extraire dans un répertoire quelconque le fichier archive que vous aurez téléchargé depuis l'internet. L'opération de désarchivage crée automatiquement un sous-répertoire nommé *Karrigell-numéro de version*. C'est ce répertoire que nous considérerons comme répertoire racine dans les lignes qui suivent.

Si vous ne comptez pas utiliser le serveur de bases de données *Gadfly*[75] qui vous est fourni en complément de Karrigell lui-même, c'est tout ! Sinon, entrez dans le sous-répertoire **`gadfly-1.0.0`** et lancez la commande : **`python setup.py install`** (Sous Linux, il faut être *root*). Vous devez effectuer cette opération si vous souhaitez visualiser la totalité de la démonstration intégrée.

Démarrage du serveur :

Il s'agit donc bel et bien de mettre en route *un serveur web*, auquel vous pourrez accéder ensuite à l'aide d'un navigateur quelconque, localement ou par l'intermédiaire d'un réseau. Avant de le faire démarrer, il est cependant conseillé de jeter un petit coup d'oeil dans son fichier de configuration, lequel se nomme **Karrigell.ini** et se trouve dans le répertoire-racine.

Par défaut, Karrigell attend les requêtes *http* sur le port n° 80. Et c'est bien ce numéro de port que la plupart des logiciels navigateurs utilisent eux-mêmes par défaut. Cependant, si vous installez Karrigell sur une machine Linux dont vous n'êtes pas l'administrateur, vous n'avez pas le droit d'utiliser les numéros de port inférieurs à 1024 (pour des raisons de sécurité). Si vous êtes dans ce cas, vous devez donc modifier le fichier de configuration afin que Karrigell utilise un numéro de port plus élevé. Pour ce faire, vous devez repérer la section **[Server]** dans **Karrigell.ini** (aux alentours de la ligne 75), et y activer une ligne d'instruction telle que **`port=8080`** , ce qui activera l'utilisation du n° de port 8080. Plus tard, vous souhaiterez peut-être encore modifier le fichier de

[75] Voyez le chapitre précédent : Gadfly est un serveur de bases de données écrit en Python.

configuration afin de modifier l'emplacement du répertoire racine pour votre site web (par défaut, c'est le sous-répertoire **webapps**. Voyez pour ce faire la section **[Directories]**).

Une fois le fichier de configuration modifié, entrez dans le répertoire racine du serveur, si vous n'y êtes pas déjà, et lancez simplement la commande :

```
python Karrigell.py
```

C'est tout. Votre serveur Karrigell se met en route, et vous pouvez en vérifier le fonctionnement tout de suite à l'aide de votre navigateur web préféré. Si vous lancez celui-ci sur la même machine que le serveur, vous le dirigerez vers une adresse telle que : *http://localhost:8080/index.html*, « localhost » étant le terme consacré pour désigner la machine locale, « 8080 » le numéro de port choisi dans le fichier de configuration, et « index.html » le nom du fichier qui contient la page d'accueil du site. Par contre, si vous voulez accéder à cette même page d'accueil depuis une autre machine, vous devrez (dans le navigateur de celle-ci) indiquer le nom ou l'adresse IP du serveur, en lieu et place de *localhost*.

Avec l'adresse indiquée au paragraphe précédent[76], vous atteindrez la page d'accueil d'un site de démonstration de Karrigell, qui est déjà pré-installé dans le répertoire racine. Vous y retrouverez la documentation de base, ainsi que toute une série d'exemples.

Dans ce qui précède, il est sous-entendu que vous avez lancé le serveur depuis une console texte, ou depuis une fenêtre de terminal. Dans un cas comme dans l'autre, les messages de contrôle émis par le serveur apparaîtront dans cette console ou cette fenêtre. C'est là que vous pourrez rechercher certains messages d'erreur éventuels. C'est là aussi que vous devrez intervenir si vous voulez arrêter le serveur (avec la combinaison de touches CTRL-C).

Ébauche de site web

Essayons à présent de réaliser notre propre ébauche de site web. À la différence d'un serveur web classique, Karrigell peut gérer non seulement des pages HTML statiques (fichiers .htm, .html, .gif, .jpg, .css) mais également :

- des scripts Python (fichiers .py)
- des scripts hybrides Python Inside HTML (fichiers .pih)
- des scripts hybrides HTML Inside Python (fichiers .hip)

Laissons de côté les scripts hybrides, dont vous pourrez étudier vous-même la syntaxe (par ailleurs très simple) si vous vous lancez dans une réalisation d'une certaine importance (ils pourront vous faciliter la vie). Dans le contexte limité de ces pages, nous nous contenterons de quelques expériences de base avec des scripts Python ordinaires.

Comme tous les autres éléments du site (fichiers .html, .gif, .jpeg, etc.), ces scripts Python devront être placés dans le sous-répertoire **webapps**, ou mieux encore, dans un sous-répertoire de celui-ci, auquel vous donnerez un nom de votre choix, comme il est d'usage de le faire lorsque l'on cherche à structurer convenablement le site en construction. Il vous suffira dans ce cas d'inclure le nom de ces sous-répertoires dans les adresses correspondantes[77].

---

[76] Si vous avez laissé en place le n° de port par défaut (80), il est inutile de le rappeler dans les adresses, puisque c'est ce numéro de port qui est utilisé par défaut par la plupart des navigateurs. Une autre convention consiste à considérer que la page d'accueil d'un site Web se trouve presque toujours dans un fichier nommé *index.htm* ou *index.html,* Lorsque l'on souhaite visiter un site Web en commençant par sa page d'accueil, on peut donc en général omettre ce nom dans l'adresse. *Karrigell* respecte cette convention, et vous pouvez donc vous connecter en utilisant une adresse simplifiée telle que : *http://localhost:8080* ou même : *http://localhost* (si le n° de port est 80).

[77] Comme nous l'avons signalé à la page précédente, il est possible de changer les options par défaut de Karrigell en éditant le fichier Karrigell.ini. Vous pouvez notamment choisir un autre répertoire par défaut pour les applications que vous créez, en modifiant la section [Directories].

Par exemple, créez dans **webapps** un sous-répertoire **essais**, puis enregistrez-y un petit script d'une seule ligne comportant une instruction **print**, tel que :

```
print "<h2>Bienvenue dans ma première application web !</h2>"
```

Si vous donnez à ce petit script le nom de **hello.py**, il sera donc sauvegardé dans :

```
~/Karrigell-numéro_de_version/webapps/essais/hello.py
```

Si le serveur Karrigell fonctionne, il vous suffit alors d'entrer une adresse telle que :
http://localhost:8080/essais/hello.py , ou encore :
http://localhost:8080/essais/hello (l'extension .py peut être omise)
dans votre navigateur web. Vous devriez y voir apparaître le message.

Cet exemple vous montre donc que dans l'environnement Karrigell, la sortie de l'instruction **print** est redirigée vers la fenêtre du navigateur client, plutôt que la console (ou la fenêtre de terminal) du serveur.

Étant donné que l'affichage a lieu dans une fenêtre de navigateur web, vous pouvez utiliser toutes les ressources de la syntaxe HTML afin d'obtenir un formatage déterminé. Vous pouvez par exemple afficher un petit tableau de 2 lignes et 3 colonnes, avec les instructions suivantes :

```
print """
<TABLE BORDER="1" CELLPADDING="5">
<TR> <TD> Rouge </TD> <TD> Vert </TD> <TD> Bleu </TD> </TR>
<TR> <TD> 15 % </TD> <TD> 62 % </TD> <TD> 23 % </TD> </TR>
</TABLE>
"""
```

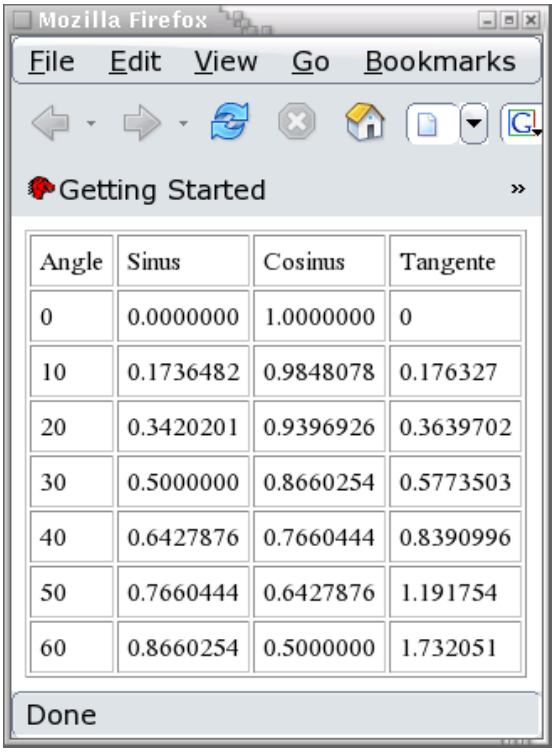
Rappelons que la balise **TABLE** définit un tableau. Son option **BORDER** spécifie la largeur des bordures de séparation, et **CELLPADDING** l'écart à réserver autour du contenu des cellules. Les Balises **TR** et **TD** (*Table Row* et *Table Data*) définissent les lignes et les cellules du tableau.

Vous pouvez bien entendu utiliser également toutes les ressources de Python, comme dans l'exemple ci-dessous où nous construisons une table des sinus, cosinus et tangentes des angles compris entre 0° et 90°, à l'aide d'une boucle classique :

```
 1#  from math import sin, cos, tan, pi
 2#
 3#  # Construction de l'en-tête du tableau avec les titres de colonnes :
 4#  print """<TABLE BORDER="1" CELLPADDING="5">
 5#  <TR><TD>Angle</TD><TD>Sinus</TD><TD>Cosinus</TD><TD>Tangente</TD></TR>"""
 6#
 7#  for angle in range(0,62,10):
 8#      # conversion des degrés en radians :
 9#      aRad = angle * pi / 180
10#      # construction d'une ligne de tableau, en exploitant le formatage des
11#      # chaînes de caractères pour fignoler l'affichage :
12#      print "<TR><TD>%s</TD><TD>%8.7f</TD><TD>%8.7f</TD><TD>%8.7g</TD></TR>"\
13#            % (angle, sin(aRad), cos(aRad), tan(aRad))
14#
15#  print "</TABLE>"
```

- Ligne 7 : Nous nous servons de la fonction **range()** pour définir la gamme d'angles à couvrir (de zéro à 60 degrés par pas de 10).

- Ligne 9 : Les fonctions trigonométriques de Python nécessitent que les angles soient exprimés en *radians*. Il faut donc effectuer une conversion.
- Ligne 12 : Chaque ligne du tableau comporte quatre valeurs, lesquelles sont mises en forme à l'aide du système de formatage des chaînes de caractères décrit déjà à la page 138 : le marqueur de conversion « `%8.7f` » force un affichage à 8 chiffres, dont 7 après la « virgule » décimale. Le marqueur « `%8.7g` » fait à peu près la même chose, mais passe à la notation scientifique lorsque c'est nécessaire.

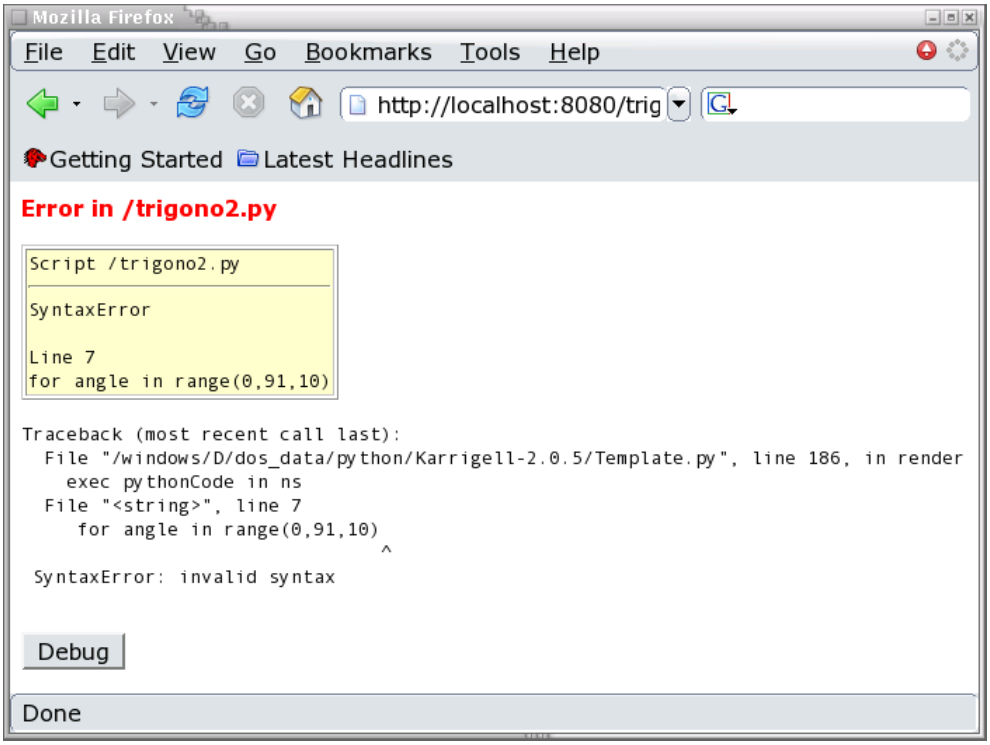


Angle	Sinus	Cosinus	Tangente
0	0.0000000	1.0000000	0
10	0.1736482	0.9848078	0.176327
20	0.3420201	0.9396926	0.3639702
30	0.5000000	0.8660254	0.5773503
40	0.6427876	0.7660444	0.8390996
50	0.7660444	0.6427876	1.191754
60	0.8660254	0.5000000	1.732051

À ce stade, vous vous demandez peut-être où se situe la différence entre ce que nous venons d'expérimenter ici et un script CGI classique (tels ceux des pages 275 et suivantes).

L'intérêt de travailler dans un environnement plus spécifique tel que Karrigell apparaît cependant très vite si vous faites des erreurs. En programmation CGI classique, les messages d'erreur émis par l'interpréteur Python ne s'affichent pas dans la fenêtre du navigateur. Ils sont enregistrés dans un fichier journal du serveur (*Apache*, par exemple), ce qui ne facilite pas leur consultation.

En revanche, avec un outil comme Karrigell, vous disposez d'une signalisation très efficace, ainsi que d'un outil de déboguage complet. Faites l'expérience d'introduire une petite erreur dans le script ci-dessus, et relancez votre navigateur sur la page modifiée. Par exemple, en supprimant le double point à la fin de la ligne 7, nous avons obtenu nous-mêmes l'affichage suivant :



En cliquant sur le bouton « Debug », on obtient encore une foule d'informations complémentaires (affichage du script complet, variables d'environnement, etc.).

Prise en charge des sessions

Lorsque l'on élabore un site web interactif, on souhaite fréquemment que la personne visitant le site puisse s'identifier et fournir un certain nombre de renseignements tout au long de sa visite dans différentes pages (l'exemple type étant le remplissage d'un « caddy » au cours de la consultation d'un site commercial), toutes ces informations étant conservées quelque part jusqu'à la fin de sa visite. Et il faut bien entendu réaliser cela, indépendamment pour chaque client connecté.

Il serait possible de transmettre les informations de page en page à l'aide de champs de formulaires cachés (balises **<INPUT TYPE="hidden">**), mais ce serait compliqué et très contraignant. Il est préférable que le système serveur soit doté d'un mécanisme spécifique, qui attribue à chaque client une *session* particulière. Karrigell réalise cet objectif par l'intermédiaire de *cookies*.

Lorsqu'un nouveau visiteur du site s'identifie, le serveur génère un cookie (c'est-à-dire un petit paquet d'informations) appelé *sessionId* et l'envoie au navigateur web, qui l'enregistre. Ce cookie contient un *identifiant de session* unique, auquel correspond un *objet-session* sur le serveur. Lorsque le visiteur parcourt les autres pages du site, son navigateur renvoie à chaque fois le contenu du cookie au serveur, ce qui permet à celui-ci de l'identifier et de retrouver l'objet-session qui lui correspond. L'objet-session reste donc disponible tout au long de la visite de l'internaute : il s'agit d'un objet Python ordinaire, dans lequel on mémorise un nombre quelconque d'informations sous forme d'attributs.

Au niveau de la programmation, voici comment cela se passe :

Pour chaque page dans laquelle vous voulez consulter ou modifier une information de session, vous commencez par créer un objet de la classe **Session()** :

```
oSess = Session()
```

Si vous êtes au début de la session, Karrigell génère un identifiant unique (une chaîne de caractères pseudo-aléatoire), le place dans un cookie et envoie celui-ci au navigateur web. Il mémorise le même identifiant dans l'objet-session. À partir de ce moment, vous pouvez ajouter un nombre quelconque d'attributs à cet objet-session :

```
oSess.nom = "Jean Dupont"
oSess.age = 17
... etc ...
```

Dans les autres pages, vous procédez de la même manière, mais en sachant que l'instruction :

```
oSess = Session()
```

ne produit plus cette fois un nouvel objet, mais restitue au contraire l'objet-session correspondant à l'utilisateur en ligne. En effet : lorsqu'il demande l'accès à ces autres pages, le navigateur de l'utilisateur envoie au serveur son identifiant de session (via le cookie). Cet identifiant, déjà connu du serveur, lui permet de retrouver l'objet-session correspondant, créé en début de session. Tout ceci se passe de manière quasi-transparente pour vous, le programmeur : il vous suffit de savoir que toutes les informations concernant l'utilisateur en ligne sont pour vous de simples attributs de l'objet-session. Vous pouvez donc aisément accéder aux valeurs de ces attributs, et aussi en ajouter de nouveaux :

```
oSess = Session()              # récupérer l'objet-session indiqué par le cookie
om = oSess.nom                 # retrouver la valeur d'un attribut existant
oSess.article = 49137          # ajouter un nouvel attribut
```

Les objets-sessions prennent aussi en charge une méthode **close()**, qui a pour effet d'effacer l'information de session. Vous n'êtes cependant pas obligé de clore explicitement les sessions : Karrigell s'assure de toute façon qu'il n'y ait jamais plus de 1000 sessions simultanées : il efface les plus anciennes quand on arrive à la 1000ème.

Exemple de mise en œuvre :

Sauvegardez les trois petits scripts ci-dessous dans le répertoire-racine. Le premier génère un formulaire HTML similaire à ceux qui ont été décrits plus haut. Nommez-le **sessionTest1.py** :

```
 1#  # Affichage d'un formulaire d'inscription :
 2#
 3#  print """
 4#  <H3>Veuillez vous identifier, SVP :</H3>
 5#
 6#  <FORM ACTION = "sessionTest2.py">
 7#  Votre nom : <INPUT NAME = "nomClient"> <BR>
 8#  Votre prénom : <INPUT NAME = "prenomClient"> <BR>
 9#  Votre sexe (m/f) : <INPUT NAME = "sexeClient" SIZE ="1"> <BR>
10#  <INPUT TYPE = "submit" VALUE = "OK">
11#  </FORM>"""
```

Le suivant sera nommé **sessionTest2.py**. C'est le script mentionné dans la balise d'ouverture du formulaire ci-dessus à la ligne 6, et qui sera invoqué lorsque l'utilisateur actionnera le bouton mis en place à la ligne 10. Ce script peut retrouver les valeurs entrées par l'utilisateur dans les différents champs du formulaire, par l'intermédiaire d'un *dictionnaire de requête* situé dans la variable d'environnement **QUERY** de Karrigell[78] :

```
 1#  obSess = Session()
 2#
 3#  obSess.nom = QUERY["nomClient"]
 4#  obSess.prenom = QUERY["prenomClient"]
 5#  obSess.sexe = QUERY["sexeClient"]
 6#
 7#  if obSess.sexe.upper() == "M":
 8#      vedette ="Monsieur"
 9#  else:
10#      vedette ="Madame"
11#  print "<H3> Bienvenue, %s %s </H3>" % (vedette, obSess.nom)
12#  print "<HR>"
13#  print """
14#  <a href = "sessionTest3.py"> Suite...</a>"""
```

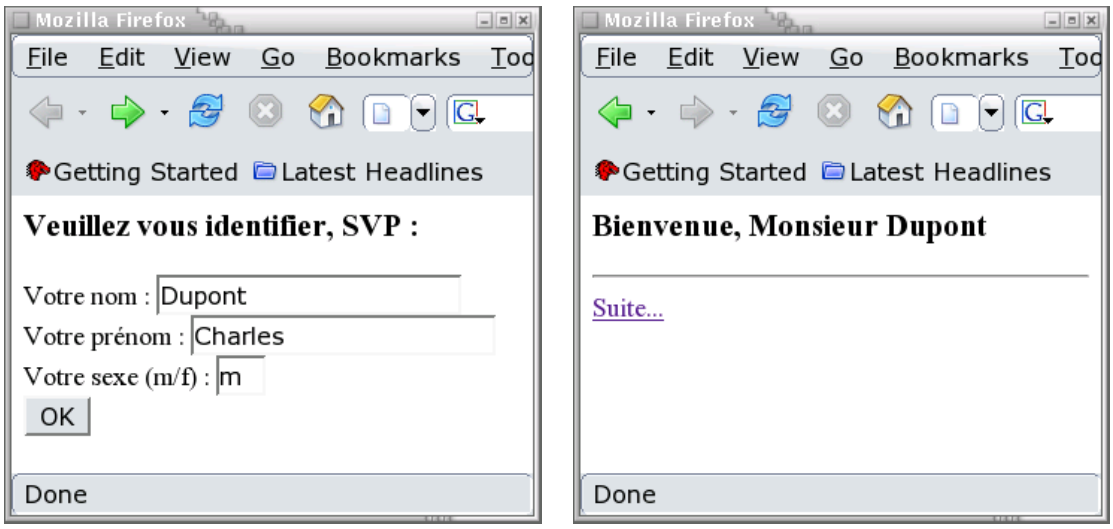
- La première ligne de ce script crée l'objet-session, génère pour lui un identifiant unique, et expédie celui-ci au navigateur, sous la forme d'un *cookie.*
- Dans les lignes 3, 4, 5, on récupère les valeurs entrées dans les champs du formulaire précédent, en utilisant leurs noms comme clés d'accès au *dictionnaire de requêtes.* Ce dictionnaire est créé et géré automatiquement par Karrigell.
- La ligne 14 définit un *lien http* pointant vers le troisième script, nommé **sessionTest3.py** :

```
 1#  suiviSess = Session()           # retrouver l'objet-session
 2#  suiviSess.article = 12345      # lui ajouter des attributs
 3#  suiviSess.prix = 43.67
 4#
 5#  print """
 6#  <H3> Page suivante </H3> <HR>
```

[78] *Karrigell* met en place un certain nombre de variables globales dont les noms sont en majuscules pour éviter un conflit éventuel avec les vôtres. Celle-ci joue le même rôle que la fonction *FieldStorage()* du module cgi. Veuillez consulter la documentation de *Karrigell* si vous souhaitez obtenir des explications plus détaillées.

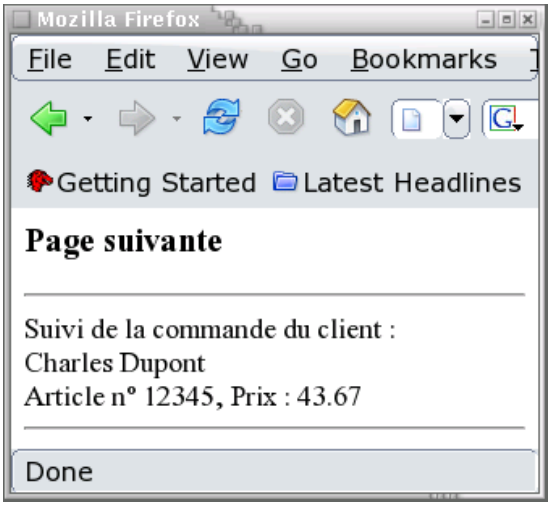
```
  7#  Suivi de la commande du client : <BR> %s %s <BR>
  8#  Article n° %s, Prix : %s <HR>
  9#  """ % (suiviSess.prenom, suiviSess.nom,
 10#         suiviSess.article, suiviSess.prix)
```

Dirigez votre navigateur web vers l'adresse : *http://localhost:8080/sessionTest1*. Entrez des valeurs de votre choix dans les champs du formulaire, et cliquez sur le bouton OK :



Comme attendu, les informations entrées dans le formulaire sont transmises à la deuxième page. À présent, si vous cliquez sur le lien : « Suite... » dans celle-ci, vous dirigez encore une fois votre navigateur vers une nouvelle page, mais celle-ci n'aura fait l'objet d'aucune transmission directe de données (puisqu'on n'y accède pas par l'intermédiaire d'un formulaire). Dans le script **sessionTest3.py** qui génère cette page, vous ne pouvez donc pas utiliser la variable **QUERY** pour retrouver les informations entrées par le visiteur.

C'est ici qu'intervient le mécanisme des objets-sessions. Lors du lancement de ce troisième script, le cookie mémorisé par le navigateur est relu par le serveur, ce qui lui permet de récupérer l'objet-session créé dans le script précédent.



Analysez les trois premières lignes du script **sessionTest3.py** : l'objet **suiviSess** instancié à partir de la classe **Session()** est l'objet-session régénéré. Il contient les informations sauvegardées à la page précédente, et on peut lui en ajouter d'autres dans des attributs supplémentaires.

Vous aurez compris que vous pouvez désormais récupérer toutes ces informations de la même manière dans n'importe quelle autre page, car elles persisteront jusqu'à ce que l'utilisateur termine sa visite du site, à moins que vous ne fermiez vous-même cette session par programme, à l'aide de la méthode **close()** évoquée plus haut.

Exercice :

17.2 Ajoutez au script précédent un lien vers une quatrième page, et écrivez le script qui générera celle-ci. Les informations devront cette fois être affichées dans un tableau :

```
Nom         Prénom      Sexe      Article     Prix

Dupont      Charles     m         12345       43.67
Durand      Louise      f         6789        12.75
etc.
```

Autres développements

Nous terminons ici cette brève étude de Karrigell, car il nous semble vous avoir expliqué l'essentiel de ce qu'il vous faut connaître pour démarrer. Si vous désirez en savoir davantage, il vous suffira de consulter la documentation (disponible en français) et les exemples fournis avec le produit. Comme nous l'avons déjà signalé plus haut, l'installation de Karrigell inclut l'installation du système de bases de données *Gadfly*. Vous pouvez donc très rapidement et très aisément réaliser un site interactif permettant la consultation à distance d'un ensemble de données quelconques, en admettant bien entendu que la charge de requêtes de votre site reste modérée, et que la taille de la base de données elle-même ne devienne pas gigantesque.

Si vous ambitionnez de réaliser un site web très ambitieux, prenez également la peine d'étudier d'autres offres logicielles, comme par exemple *Django*, *Pylons*, *TurboGears*, *Twisted*, *CherryPy*, *Zope* ... associés à *Apache* pour le système serveur, et *SQLite*, *MySQL* ou *PostgreSQL* pour le gestionnaire de bases de données. Vous aurez compris qu'il s'agit d'un domaine très vaste, où vous pourrez exercer votre créativité fort longtemps ...

18

Communications à travers un réseau

Le développement extraordinaire de l'internet a amplement démontré que les ordinateurs peuvent être des outils de communication très efficaces. Dans ce chapitre, nous allons expérimenter la plus simple des techniques d'interconnexion de deux programmes, qui leur permette de s'échanger des informations basiques (données numériques, chaînes de caractères, ...) par l'intermédiaire d'un réseau.

Pour ce qui va suivre, nous supposerons donc que vous collaborez avec un ou plusieurs de vos condisciples, et que vos postes de travail Python sont connectés à un réseau local dont les communications utilisent le protocole TCP/IP. Le système d'exploitation n'a pas d'importance : vous pouvez par exemple installer l'un des scripts Python décrits ci-après sur un poste de travail fonctionnant sous Linux, et le faire dialoguer avec un autre script mis en œuvre sur un poste de travail confié aux bons soins d'un système d'exploitation différent, tel que MacOS ou Windows.

Vous pouvez également expérimenter ce qui suit sur une seule et même machine, en mettant les différents scripts en œuvre dans des fenêtres indépendantes.

Les sockets

Le premier exercice qui va vous être proposé consistera à établir une communication entre deux machines seulement. L'une et l'autre pourront s'échanger des messages à tour de rôle, mais vous constaterez cependant que leurs configurations ne sont pas symétriques. Le script installé sur l'une de ces machines jouera en effet le rôle d'un logiciel *serveur*, alors que l'autre se comportera comme un logiciel *client*.

Le logiciel serveur fonctionne en continu, sur une machine dont l'identité est bien définie sur le réseau grâce à une *adresse IP* spécifique[79]. Il guette en permanence l'arrivée de requêtes expédiées par les clients potentiels en direction de cette adresse, par l'intermédiaire d'un *port de communication* bien déterminé. Pour ce faire, le script correspondant doit mettre en œuvre un objet logiciel associé à ce port, que l'on appelle un *socket*.

Au départ d'une autre machine, le logiciel client tente d'établir la connexion en émettant une *requête* appropriée. Cette requête est un message qui est confié au réseau, un peu comme on confie une lettre à la Poste. Le réseau pourrait en effet acheminer la requête vers n'importe quelle autre machine, mais une seule est visée : pour que la destination visée puisse être atteinte, la requête contient dans son en-tête l'indication de l'adresse IP et du port de communication destinataires.

[79] Une machine particulière peut également être désignée par un nom plus explicite, mais à la condition qu'un mécanisme ait été mis en place sur le réseau (DNS) pour traduire automatiquement ce nom en adresse IP. Veuillez consulter votre cours sur les systèmes d'exploitation et les réseaux pour en savoir davantage.

Lorsque la connexion est établie avec le serveur, le client lui assigne lui-même l'un de ses propres ports de communication. A partir de ce moment, on peut considérer qu'*un canal privilégié relie les deux machines*, comme si on les avait connectées l'une à l'autre par l'intermédiaire d'un fil (les deux ports de communication respectifs jouant le rôle des deux extrémités de ce fil). L'échange d'informations proprement dit peut commencer.

Pour pouvoir utiliser les ports de communication réseau, les programmes font appel à un ensemble de procédures et de fonctions du système d'exploitation, par l'intermédiaire d'objets interfaces que l'on appelle donc des sockets. Ceux-ci peuvent mettre en œuvre deux techniques de communication différentes et complémentaires : celle des *paquets* (que l'on appelle aussi des *datagrammes*), très largement utilisée sur l'internet, et celle de la *connexion continue*, ou *stream socket*, qui est un peu plus simple.

Construction d'un serveur élémentaire

Pour nos premières expériences, nous allons utiliser la technique des *stream sockets*.
Celle-ci est en effet parfaitement appropriée lorsqu'il s'agit de faire communiquer des ordinateurs interconnectés par l'intermédiaire d'un réseau local. C'est une technique particulièrement aisée à mettre en œuvre, et elle permet un débit élevé pour l'échange de données.

L'autre technologie (celle des *paquets*) serait préférable pour les communications expédiées *via* l'internet, en raison de sa plus grande fiabilité (les mêmes paquets peuvent atteindre leur destination par différents chemins, être émis ou ré-émis en plusieurs exemplaires si cela se révèle nécessaire pour corriger les erreurs de transmission), mais sa mise en œuvre est un peu plus complexe. Nous ne l'étudierons pas dans ce livre.

Le premier script ci-dessous met en place un serveur capable de communiquer avec un seul client. Nous verrons un peu plus loin ce qu'il faut lui ajouter, afin qu'il puisse prendre en charge en parallèle les connexions de plusieurs clients.

```
 1#  # Définition d'un serveur réseau rudimentaire
 2#  # Ce serveur attend la connexion d'un client, pour entamer un dialogue avec lui
 3#
 4#  import socket, sys
 5#
 6#  HOST = '192.168.14.152'
 7#  PORT = 50000
 8#
 9#  # 1) création du socket :
10#  mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11#
12#  # 2) liaison du socket à une adresse précise :
13#  try:
14#      mySocket.bind((HOST, PORT))
15#  except socket.error:
16#      print "La liaison du socket à l'adresse choisie a échoué."
17#      sys.exit()
18#
19#  while 1:
20#      # 3) Attente de la requête de connexion d'un client :
21#      print "Serveur prêt, en attente de requêtes ..."
22#      mySocket.listen(5)
23#
24#      # 4) Etablissement de la connexion :
25#      connexion, adresse = mySocket.accept()
26#      print "Client connecté, adresse IP %s, port %s" % (adresse[0], adresse[1])
27#
28#      # 5) Dialogue avec le client :
29#      connexion.send("Vous êtes connecté au serveur Marcel. Envoyez vos messages.")
30#      msgClient = connexion.recv(1024)
```

```
31#      while 1:
32#          print "C>", msgClient
33#          if msgClient.upper() == "FIN" or msgClient =="":
34#              break
35#          msgServeur = raw_input("S> ")
36#          connexion.send(msgServeur)
37#          msgClient = connexion.recv(1024)
38#
39#      # 6) Fermeture de la connexion :
40#      connexion.send("Au revoir !")
41#      print "Connexion interrompue."
42#      connexion.close()
43#
44#      ch = raw_input("<R>ecommencer <T>erminer ? ")
45#      if ch.upper() =='T':
46#          break
```

Commentaires :

- Ligne 4 : Le module **socket** contient toutes les fonctions et les classes nécessaires pour construire des programmes communiquants. Comme nous allons le voir dans les lignes suivantes, l'établissement de la communication comporte six étapes.
- Lignes 6 & 7 : Ces deux variables définissent l'identité du serveur, telle qu'on l'intégrera au socket. **HOST** doit contenir une chaîne de caractères indiquant l'adresse IP du serveur sous la forme décimale habituelle, ou encore le nom DNS de ce même serveur (mais à la condition qu'un mécanisme de résolution des noms ait été mis en place sur le réseau). **PORT** doit contenir un entier, à savoir le numéro d'un port qui ne soit pas déjà utilisé pour un autre usage, et de préférence une valeur supérieure à 1024 (Cfr. votre cours sur les services réseau).
- Lignes 9 & 10 : Première étape du mécanisme d'interconnexion. On instancie un objet de la classe **socket()**, en précisant deux options qui indiquent le type d'adresses choisi (nous utiliserons des adresses de type « internet ») ainsi que la technologie de transmission (datagrammes ou connexion continue (*stream*) : nous avons décidé d'utiliser cette dernière).
- Lignes 12 à 17 : Seconde étape. On tente d'établir la liaison entre le socket et le port de communication. Si cette liaison ne peut être établie (port de communication occupé, par exemple, ou nom de machine incorrect), le programme se termine sur un message d'erreur.
  Remarque : la méthode **bind()** du socket attend un argument du type tuple, raison pour laquelle nous devons enfermer nos deux variables dans une double paire de parenthèses.
- Ligne 19 : Notre programme serveur étant destiné à fonctionner en permanence dans l'attente des requêtes de clients potentiels, nous le lançons dans une boucle sans fin.
- Lignes 20 à 22 : Troisième étape. Le socket étant relié à un port de communication, il peut à présent se préparer à recevoir les requêtes envoyées par les clients. C'est le rôle de la méthode **listen()**. L'argument qu'on lui transmet indique le nombre maximum de connexions à accepter en parallèle. Nous verrons plus loin comment gérer celles-ci.
- Lignes 24 à 26 : Quatrième étape. Lorsqu'on fait appel à sa méthode **accept()**, le socket attend indéfiniment qu'une requête se présente. Le script est donc interrompu à cet endroit, un peu comme il le serait si nous faisions appel à une fonction **input()** pour attendre une entrée clavier. Si une requête est réceptionnée, la méthode **accept()** renvoie un tuple de deux éléments : le premier est la référence d'un nouvel objet de la classe **socket()**[80], qui sera la véri-

80 Nous verrons plus loin l'utilité de créer ainsi un nouvel objet socket pour prendre en charge la communication, plutôt que d'utiliser celui qui a déjà créé à la ligne 10. En bref, si nous voulons que notre serveur puisse prendre en charge simultanément les connexions de plusieurs clients, il nous faudra disposer d'un socket distinct pour chacun d'eux, indépendamment du premier que l'on laissera fonctionner en permanence pour réceptionner les requêtes qui continuent à arriver en provenance de nouveaux clients.

table interface de communication entre le client et le serveur, et le second un autre tuple contenant les coordonnées de ce client (son adresse IP et le n° de port qu'il utilise lui-même).

- Lignes 28 à 30 : Cinquième étape. La communication proprement dite est établie. Les méthodes **send()** et **recv()** du socket servent évidemment à l'émission et à la réception des messages, qui doivent être de simples chaînes de caractères.
  Remarques : la méthode **send()** renvoie le nombre d'octets expédiés. L'appel de la méthode **recv**() doit comporter un argument entier indiquant le nombre maximum d'octets à réceptionner en une fois (Les octets surnuméraires sont mis en attente dans un tampon. Ils sont transmis lorsque la même méthode **recv()** est appelée à nouveau).
- Lignes 31 à 37 : Cette nouvelle boucle sans fin maintient le dialogue jusqu'à ce que le client décide d'envoyer le mot « fin » ou une simple chaîne vide. Les écrans des deux machines afficheront chacune l'évolution de ce dialogue.
- Lignes 39 à 42 : Sixième étape. Fermeture de la connexion.

Construction d'un client rudimentaire

Le script ci-dessous définit un logiciel client complémentaire du serveur décrit dans les pages précédentes. On notera sa grande simplicité.

```
 1#  # Définition d'un client réseau rudimentaire
 2#  # Ce client dialogue avec un serveur ad hoc
 3#
 4#  import socket, sys
 5#
 6#  HOST = '192.168.14.152'
 7#  PORT = 50000
 8#
 9#  # 1) création du socket :
10#  mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11#
12#  # 2) envoi d'une requête de connexion au serveur :
13#  try:
14#      mySocket.connect((HOST, PORT))
15#  except socket.error:
16#      print "La connexion a échoué."
17#      sys.exit()
18#  print "Connexion établie avec le serveur."
19#
20#  # 3) Dialogue avec le serveur :
21#  msgServeur = mySocket.recv(1024)
22#
23#  while 1:
24#      if msgServeur.upper() == "FIN" or msgServeur =="":
25#          break
26#      print "S>", msgServeur
27#      msgClient = raw_input("C> ")
28#      mySocket.send(msgClient)
29#      msgServeur = mySocket.recv(1024)
30#
31#  # 4) Fermeture de la connexion :
32#  print "Connexion interrompue."
33#  mySocket.close()
```

Commentaires :

- Le début du script est similaire à celui du serveur. L'adresse IP et le port de communication doivent être ceux du serveur.

- Lignes 12 à 18 : On ne crée cette fois qu'un seul objet socket, dont on utilise la méthode **connect()** pour envoyer la requête de connexion.
- Lignes 20 à 33 : Une fois la connexion établie, on peut dialoguer avec le serveur en utilisant les méthodes **send()** et **recv()** déjà décrites plus haut pour celui-ci.

Gestion de plusieurs tâches en parallèle à l'aide des threads

Le système de communication que nous avons élaboré dans les pages précédentes est vraiment très rudimentaire : d'une part il ne met en relation que deux machines seulement, et d'autre part il limite la liberté d'expression des deux interlocuteurs. Ceux-ci ne peuvent en effet envoyer des messages que chacun à leur tour. Par exemple, lorsque l'un d'eux vient d'émettre un message, son système reste bloqué tant que son partenaire ne lui a pas envoyé une réponse. Lorsqu'il vient de recevoir une telle réponse, son système reste incapable d'en réceptionner une autre, tant qu'il n'a pas entré lui-même un nouveau message, ... et ainsi de suite.

Tous ces problèmes proviennent du fait que nos scripts habituels ne peuvent s'occuper que d'une seule chose à la fois. Lorsque le flux d'instructions rencontre une fonction **input()**, par exemple, il ne se passe plus rien tant que l'utilisateur n'a pas introduit la donnée attendue. Et même si cette attente dure très longtemps, il n'est habituellement pas possible que le programme effectue d'autres tâches pendant ce temps. Ceci n'est toutefois vrai qu'au sein d'un seul et même programme : vous savez certainement que vous pouvez exécuter d'autres applications entretemps sur votre ordinateur, car les systèmes d'exploitation modernes sont *multi-tâches*.

Les pages qui suivent sont destinées à vous expliquer comment vous pouvez introduire cette fonctionnalité *multi-tâches* dans vos programmes, afin que vous puissiez développer de véritables applications réseau, capables de communiquer simultanément avec plusieurs partenaires.

Veuillez à présent considérer le script de la page précédente. Sa fonctionnalité essentielle réside dans la boucle **while** des lignes 23 à 29. Or, cette boucle s'interrompt à deux endroits :

- à la ligne 27, pour attendre les entrées clavier de l'utilisateur (fonction **raw_input()**) ;
- à la ligne 29, pour attendre l'arrivée d'un message réseau.

Ces deux attentes sont donc *successives*, alors qu'il serait bien plus intéressant qu'elles soient *simultanées*. Si c'était le cas, l'utilisateur pourrait expédier des messages à tout moment, sans devoir attendre à chaque fois la réaction de son partenaire. Il pourrait également recevoir n'importe quel nombre de messages, sans l'obligation d'avoir à répondre à chacun d'eux pour recevoir les autres.

Nous pouvons arriver à ce résultat si nous apprenons à gérer plusieurs séquences d'instructions *en parallèle* au sein d'un même programme. Mais comment cela est-il possible ?

Au cours de l'histoire de l'informatique, plusieurs techniques ont été mises au point pour partager le temps de travail d'un processeur entre différentes tâches, de telle manière que celles-ci paraissent être effectuées en même temps (alors qu'en réalité le processeur s'occupe d'un petit bout de chacune d'elles à tour de rôle). Ces techniques sont implémentées dans le système d'exploitation, et il n'est pas nécessaire de les détailler ici, même s'il est possible d'accéder à chacune d'elles avec Python.

Dans les pages suivantes, nous allons apprendre à utiliser celle de ces techniques qui est à la fois la plus facile à mettre en œuvre, et la seule qui soit véritablement *portable* (elle est en effet suppor-

tée par tous les grands systèmes d'exploitation) : on l'appelle la technique des processus légers ou *threads*[81].

Dans un programme d'ordinateur, les threads sont des flux d'instructions qui sont menés en parallèle (quasi-simultanément), tout en partageant le même espace de noms global.

En fait, le flux d'instructions de n'importe quel programme Python suit toujours au moins un thread : le *thread principal*. À partir de celui-ci, d'autres *threads enfants* peuvent être amorcés, qui seront exécutés en parallèle. Chaque thread enfant se termine et disparaît sans autre forme de procès lorsque toutes les instructions qu'il contient ont été exécutées. Par contre, lorsque le thread principal se termine, il faut parfois s'assurer que tous ses threads enfants « meurent » avec lui.

Client gérant l'émission et la réception simultanées

Nous allons maintenant mettre en pratique la technique des threads pour construire un système de *chat*[82] simplifié. Ce système sera constitué d'un seul serveur et d'un nombre quelconque de clients. Contrairement à ce qui se passait dans notre premier exercice, personne n'utilisera le serveur lui-même pour communiquer, mais lorsque celui-ci aura été mis en route, plusieurs clients pourront s'y connecter et commencer à s'échanger des messages.

Chaque client enverra tous ses messages au serveur, mais celui-ci les ré-expédiera immédiatement à tous les autres clients connectés, de telle sorte que chacun puisse voir l'ensemble du trafic. Chacun pourra à tout moment envoyer ses messages, et recevoir ceux des autres, dans n'importe quel ordre, la réception et l'émission étant gérées simultanément, dans des threads séparés.

Le script ci-après définit le programme client. Le serveur sera décrit un peu plus loin. Vous constaterez que la partie principale du script (ligne 38 et suivantes) est similaire à celle de l'exemple précédent. Seule la partie « Dialogue avec le serveur » a été remplacée. Au lieu d'une boucle **while**, vous y trouvez à présent les instructions de création de deux objets threads (aux lignes 49 et 50), dont on démarre la fonctionnalité aux deux lignes suivantes. Ces objets threads sont crées par dérivation, à partir de la classe **Thread()** du module **threading**. Ils s'occuperont indépendamment de la réception et le l'émission des messages. Les deux *threads enfants* sont ainsi parfaitement encapsulés dans des objets distincts, ce qui facilite la compréhension du mécanisme.

```
 1# # Définition d'un client réseau gérant en parallèle l'émission
 2# # et la réception des messages (utilisation de 2 THREADS).
 3#
 4# host = '192.168.0.235'
 5# port = 40000
 6#
 7# import socket, sys, threading
 8#
 9# class ThreadReception(threading.Thread):
10#     """objet thread gérant la réception des messages"""
11#     def __init__(self, conn):
12#         threading.Thread.__init__(self)
13#         self.connexion = conn           # réf. du socket de connexion
14#
15#     def run(self):
16#         while 1:
```

[81] Dans un système d'exploitation de type Unix (comme Linux), les différents *threads* d'un même programme font partie d'un seul *processus*. Il est également possible de gérer différents processus à l'aide d'un même script Python (opération *fork*), mais l'explication de cette technique dépasse largement le cadre de ce livre.

[82] Le « chat » est l'occupation qui consiste à « papoter » par l'intermédiaire d'ordinateurs. Les canadiens francophones ont proposé le terme de *clavardage* pour désigner ce « bavardage par claviers interposés ».

```
17#             message_recu = self.connexion.recv(1024)
18#             print "*" + message_recu + "*"
19#             if message_recu =='' or message_recu.upper() == "FIN":
20#                 break
21#         # Le thread <réception> se termine ici.
22#         # On force la fermeture du thread <émission> :
23#         th_E._Thread__stop()
24#         print "Client arrêté. Connexion interrompue."
25#         self.connexion.close()
26#
27#  class ThreadEmission(threading.Thread):
28#      """objet thread gérant l'émission des messages"""
29#      def __init__(self, conn):
30#          threading.Thread.__init__(self)
31#          self.connexion = conn           # réf. du socket de connexion
32#
33#      def run(self):
34#          while 1:
35#              message_emis = raw_input()
36#              self.connexion.send(message_emis)
37#
38#  # Programme principal - Établissement de la connexion :
39#  connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40#  try:
41#      connexion.connect((host, port))
42#  except socket.error:
43#      print "La connexion a échoué."
44#      sys.exit()
45#  print "Connexion établie avec le serveur."
46#
47#  # Dialogue avec le serveur : on lance deux threads pour gérer
48#  # indépendamment l'émission et la réception des messages :
49#  th_E = ThreadEmission(connexion)
50#  th_R = ThreadReception(connexion)
51#  th_E.start()
52#  th_R.start()
```

Commentaires :

- Remarque générale : Dans cet exemple, nous avons décidé de créer deux objets threads indépendants du thread principal, afin de bien mettre en évidence les mécanismes. Notre programme utilise donc trois threads en tout, alors que le lecteur attentif aura remarqué que deux pourraient suffire. En effet : le thread principal ne sert en définitive qu'à lancer les deux autres ! Il n'y a cependant aucun intérêt à limiter le nombre de threads. Au contraire : à partir du moment où l'on décide d'utiliser cette technique, il faut en profiter pour compartimenter l'application en unités bien distinctes.
- Ligne 7 : Le module **threading** contient la définition de toute une série de classes intéressantes pour gérer les threads. Nous n'utiliserons ici que la seule classe **Thread()**, mais une autre sera exploitée plus loin (la classe **Lock()**), lorsque nous devrons nous préoccuper de problèmes de synchronisation entre différents threads concurrents.
- Lignes 9 à 25 : Les classes dérivées de la classe **Thread()** contiendront essentiellement une méthode **run()**. C'est dans celle-ci que l'on placera la portion de programme spécifiquement confiée au thread. Il s'agira souvent d'une boucle répétitive, comme ici. Vous pouvez parfaitement considérer le contenu de cette méthode comme *un script indépendant, qui s'exécute en parallèle avec les autres composants de votre application*. Lorsque ce code a été complètement exécuté, le thread se referme.
- Lignes 16 à 20 : Cette boucle gère la réception des messages. À chaque itération, le flux d'instructions s'interrompt à la ligne 17 dans l'attente d'un nouveau message, mais le reste du programme n'est pas figé pour autant : les autres threads continuent leur travail indépendamment.

- Ligne 19 : La sortie de boucle est provoquée par la réception d'un message `'fin'` (en majuscules ou en minuscules), ou encore d'un message vide (c'est notamment le cas si la connexion est coupée par le partenaire). Quelques instructions de « nettoyage » sont alors exécutées, et puis le thread se termine.
- Ligne 23 : Lorsque la réception des messages est terminée, nous souhaitons que le reste du programme se termine lui aussi. Il nous faut donc forcer la fermeture de l'autre objet thread, celui que nous avons mis en place pour gérer l'émission des messages. Cette fermeture forcée peut être obtenue à l'aide de la méthode **_Thread__stop()**[83].
- Lignes 27 à 36 : Cette classe définit donc un autre objet thread, qui contient cette fois une boucle de répétition perpétuelle. Il ne se pourra donc se terminer que contraint et forcé par méthode décrite au paragraphe précédent. À chaque itération de cette boucle, le flux d'instructions s'interrompt à la ligne 35 dans l'attente d'une entrée clavier, mais cela n'empêche en aucune manière les autres threads de faire leur travail.
- Lignes 38 à 45 : Ces lignes sont reprises à l'identique des scripts précédents.
- Lignes 47 à 52 : Instanciation et démarrage des deux objets *threads enfants*. Veuillez noter qu'il est recommandé de provoquer ce démarrage en invoquant la méthode intégrée **start()**, plutôt qu'en faisant appel directement à la méthode **run()** que vous aurez définie vous-même. Sachez également que vous ne pouvez invoquer **start()** qu'une seule fois (une fois arrêté, un objet thread ne peut pas être redémarré).

Serveur gérant les connexions de plusieurs clients en parallèle

Le script ci-après crée un serveur capable de prendre en charge les connexions d'un certain nombre de clients du même type que ce que nous avons décrit dans les pages précédentes.

Ce serveur n'est pas utilisé lui-même pour communiquer : ce sont les clients qui communiquent les uns avec les autres, par l'intermédiaire du serveur. Celui-ci joue donc le rôle d'un *relais* : il accepte les connexions des clients, puis attend l'arrivée de leurs messages. Lorsqu'un message arrive en provenance d'un client particulier, le serveur le ré-expédie à tous les autres, en lui ajoutant au passage une chaîne d'identification spécifique du client émetteur, afin que chacun puisse voir tous les messages, et savoir de qui ils proviennent.

```
 1# # Définition d'un serveur réseau gérant un système de CHAT simplifié.
 2# # Utilise les threads pour gérer les connexions clientes en parallèle.
 3#
 4# HOST = '192.168.0.235'
 5# PORT = 40000
 6#
 7# import socket, sys, threading
 8#
 9# class ThreadClient(threading.Thread):
10#     '''dérivation d'un objet thread pour gérer la connexion avec un client'''
11#     def __init__(self, conn):
12#         threading.Thread.__init__(self)
13#         self.connexion = conn
14#
15#     def run(self):
16#         # Dialogue avec le client :
17#         nom = self.getName()        # Chaque thread possède un nom
18#         while 1:
19#             msgClient = self.connexion.recv(1024)
```

[83] Que les puristes veuillent bien me pardonner : j'admets volontiers que cette astuce pour forcer l'arrêt d'un thread n'est pas vraiment recommandable. Je me suis autorisé ce raccourci afin de ne pas trop alourdir ce texte, qui se veut seulement une initiation. Le lecteur exigeant pourra approfondir cette question en consultant l'un ou l'autre des ouvrages de référence mentionnés dans la bibliographie (voir page 10)

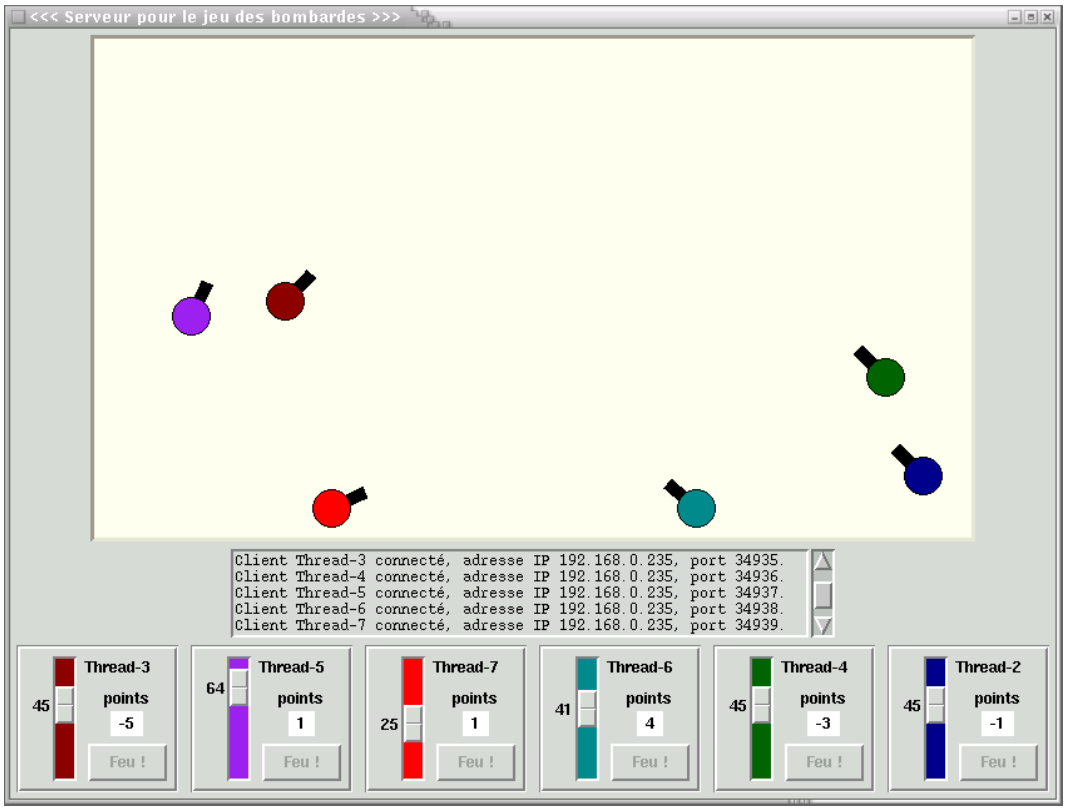
```
20#             if msgClient.upper() == "FIN" or msgClient =="":
21#                 break
22#             message = "%s> %s" % (nom, msgClient)
23#             print message
24#             # Faire suivre le message à tous les autres clients :
25#             for cle in conn_client:
26#                 if cle != nom:      # ne pas le renvoyer à l'émetteur
27#                     conn_client[cle].send(message)
28#
29#         # Fermeture de la connexion :
30#         self.connexion.close()      # couper la connexion côté serveur
31#         del conn_client[nom]        # supprimer son entrée dans le dictionnaire
32#         print "Client %s déconnecté." % nom
33#         # Le thread se termine ici
34#
35# # Initialisation du serveur - Mise en place du socket :
36# mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
37# try:
38#     mySocket.bind((HOST, PORT))
39# except socket.error:
40#     print "La liaison du socket à l'adresse choisie a échoué."
41#     sys.exit()
42# print "Serveur prêt, en attente de requêtes ..."
43# mySocket.listen(5)
44#
45# # Attente et prise en charge des connexions demandées par les clients :
46# conn_client = {}                # dictionnaire des connexions clients
47# while 1:
48#     connexion, adresse = mySocket.accept()
49#     # Créer un nouvel objet thread pour gérer la connexion :
50#     th = ThreadClient(connexion)
51#     th.start()
52#     # Mémoriser la connexion dans le dictionnaire :
53#     it = th.getName()        # identifiant du thread
54#     conn_client[it] = connexion
55#     print "Client %s connecté, adresse IP %s, port %s." %\
56#           (it, adresse[0], adresse[1])
57#     # Dialogue avec le client :
58#     connexion.send("Vous êtes connecté. Envoyez vos messages.")
```

Commentaires :

- Lignes 35 à 43 : L'initialisation de ce serveur est identique à celle du serveur rudimentaire décrit au début du présent chapitre.
- Ligne 46 : Les références des différentes connexions doivent être mémorisées. Nous pourrions les placer dans une liste, mais il est plus judicieux de les placer dans un dictionnaire, pour deux raisons : La première est que nous devrons pouvoir ajouter ou enlever ces références dans n'importe quel ordre, puisque les clients se connecteront et se déconnecteront à leur guise. La seconde est que nous pouvons disposer aisément d'un identifiant unique pour chaque connexion, lequel pourra servir de clé d'accès dans un dictionnaire. Cet identifiant nous sera en effet fourni automatiquement par La classe **Thread()**.
- Lignes 47 à 51 : Le programme commence ici une boucle de répétition perpétuelle, qui va constamment attendre l'arrivée de nouvelles connexions. Pour chacune de celles-ci, un nouvel objet **ThreadClient()** est créé, lequel pourra s'occuper d'elle indépendamment de toutes les autres.
- Lignes 52 à 54 : Obtention d'un identifiant unique à l'aide de la méthode **getName()**. Nous pouvons profiter ici du fait que Python attribue automatiquement un nom unique à chaque nouveau thread : ce nom convient bien comme identifiant (ou clé) pour retrouver la connexion correspondante dans notre dictionnaire. Vous pourrez constater qu'il s'agit d'une chaîne de caractères, de la forme : « `Thread-N` » (N étant le numéro d'ordre du thread).

- Lignes 15 à 17 : Gardez bien à l'esprit qu'il se créera autant d'objets **ThreadClient()** que de connexions, et que tous ces objets fonctionneront en parallèle. La méthode **getName()** peut alors être utilisée au sein de l'un quelconque de ces objets pour retrouver son identité particulière. Nous utiliserons cette information pour distinguer la connexion courante de toutes les autres (voir ligne 26).
- Lignes 18 à 23 : L'utilité du thread est de réceptionner tous les messages provenant d'un client particulier. Il faut donc pour cela une boucle de répétition perpétuelle, qui ne s'interrompra qu'à la réception du message spécifique : « fin », ou encore à la réception d'un message vide (cas où la connexion est coupée par le partenaire).
- Lignes 24 à 27 : Chaque message reçu d'un client doit être ré-expédié à tous les autres. Nous utilisons ici une boucle **for** pour parcourir l'ensemble des clés du dictionnaire des connexions, lesquelles nous permettent ensuite de retrouver les connexions elles-mêmes. Un simple test (à la ligne 26) nous évite de ré-expédier le message au client dont il provient.
- Ligne 31 : Lorsque nous fermons un socket de connexion, il est préférable de supprimer sa référence dans le dictionnaire, puisque cette référence ne peut plus servir. Et nous pouvons faire cela sans précaution particulière, car les éléments d'un dictionnaire ne sont pas ordonnés (nous pouvons en ajouter ou en enlever dans n'importe quel ordre).

Jeu des bombardes, version réseau



Au chapitre 15, nous avons commenté le développement d'un petit jeu de combat dans lequel des joueurs s'affrontaient à l'aide de bombardes. L'intérêt de ce jeu reste toutefois fort limité, tant qu'il se pratique sur un seul et même ordinateur. Nous allons donc le perfectionner, en y intégrant les techniques que nous venons d'apprendre. Comme le système de « chat » décrit dans les

pages précédentes, l'application complète se composera désormais de deux programmes distincts : un logiciel serveur qui ne sera mis en fonctionnement que sur une seule machine, et un logiciel client qui pourra être lancé sur toute une série d'autres. Du fait du caractère portable de Python, il vous sera même possible d'organiser des combats de bombardes entre ordinateurs gérés par des systèmes d'exploitation différents (*MacOS* <> *Linux* <> *Windows* !).

Programme serveur : vue d'ensemble

Les programmes serveur et client exploitent la même base logicielle, elle-même largement récupérée de ce qui avait déjà été mis au point tout au long du chapitre 15. Nous admettrons donc pour la suite de cet exposé que les deux versions précédentes du jeu ont été sauvegardées dans les fichiers-modules **canon03.py** et **canon04.py**, installés dans le répertoire courant. Nous pouvons en effet réutiliser une bonne partie du code qu'ils contiennent, en nous servant judicieusement de l'importation et de l'héritage de classes.

Du module **canon04**, nous allons réutiliser la classe **Canon()** telle quelle, aussi bien pour le logiciel serveur que pour le logiciel client. De ce même module, nous importerons également la classe **AppBombardes()**, dont nous ferons dériver la classe maîtresse de notre application serveur : **AppServeur()**. Vous constaterez plus loin que celle-ci produira elle-même la sous-classe **AppClient()**, toujours par héritage.

Du module **canon03**, nous récupérerons la classe **Pupitre()** dont nous tirerons une version plus adaptée au « contrôle à distance ».

Enfin, deux nouvelles classes viendront s'ajouter aux précédentes, chacune spécialisée dans la création d'un objet thread : la classe **ThreadClients()**, dont une instance surveillera en permanence le socket destiné à réceptionner les demandes de connexion de nouveaux clients, et la classe **ThreadConnexion()**, qui servira à créer autant d'objets sockets que nécessaire pour assurer le dialogue avec chacun des clients déjà connectés.

Ces nouvelles classes seront inspirées de celles que nous avions développées pour notre serveur de *chat* dans les pages précédentes. La principale différence par rapport à celui-ci est que nous devrons activer un thread spécifique pour le code qui gère l'attente et la prise en charge des connexions clientes, afin que l'application principale puisse faire autre chose pendant ce temps.

A partir de là, notre plus gros travail consistera à *développer un protocole de communication* pour le dialogue entre le serveur et ses clients. De quoi est-il question ? Tout simplement de définir la teneur des messages que vont s'échanger les machines connectées. Rassurez-vous : la mise au point de ce « langage » peut être progressive. On commence par établir un dialogue de base, puis on y ajoute petit à petit un « vocabulaire » plus étendu.

L'essentiel de ce travail peut être accompli en s'aidant du logiciel client développé précédemment pour le système de *chat*. On se sert de celui-ci pour envoyer des « ordres » au serveur en cours de développement, et on corrige celui-ci jusqu'à ce qu'il « obéisse » : en clair, les procédures que l'on met en place progressivement sur le serveur sont testées au fur et à mesure, en réponse aux messages correspondants émis « à la main » à partir du client.

Protocole de communication

Il va de soi que le protocole décrit ci-après est tout à fait *arbitraire*. Il serait parfaitement possible de choisir d'autres conventions complètement différentes. Vous pouvez bien évidemment critiquer les choix effectués, et vous souhaiterez peut-être même les remplacer par d'autres, plus efficients ou plus simples.

Vous savez déjà que les messages échangés sont de simples chaînes de caractères. Prévoyant que certains de ces messages devront transmettre plusieurs informations à la fois, nous avons décidé que chacun d'eux pourrait comporter plusieurs champs, que nous séparerons à l'aide de virgules. Lors de la réception de l'un quelconque de ces messages, nous pourrons alors aisément récupérer tous ses composants dans une liste, à l'aide de la méthode intégrée **split()**.

Voici un exemple de dialogue type, tel qu'il peut être suivi du côté d'un client. Les messages entre astérisques sont ceux qui sont reçus du serveur ; les autres sont ceux qui sont émis par le client lui-même :

```
 1#  *serveur OK*
 2#  client OK
 3#  *canons,Thread-3;104;228;1;dark red,Thread-2;454;166;-1;dark blue,*
 4#  OK
 5#  *nouveau_canon,Thread-4,481,245,-1,dark green,le_vôtre*
 6#  orienter,25,
 7#  feu
 8#  *mouvement_de,Thread-4,549,280,*
 9#  feu
10#  *mouvement_de,Thread-4,504,278,*
11#  *scores,Thread-4;1,Thread-3;-1,Thread-2;0,*
12#  *angle,Thread-2,23,*
13#  *angle,Thread-2,20,*
14#  *tir_de,Thread-2,*
15#  *mouvement_de,Thread-2,407,191,*
16#  *départ_de,Thread-2*
17#  *nouveau_canon,Thread-5,502,276,-1,dark green*
```

- Lorsqu'un nouveau client démarre, il envoie une requête de connexion au serveur, lequel lui expédie en retour le message : « serveur OK ». À la réception de ce dernier, le client répond alors en envoyant lui-même : « client OK ». Ce premier échange de politesses n'est pas absolument indispensable, mais il permet de vérifier que la communication passe bien dans les deux sens. Étant donc averti que le client est prêt à travailler, le serveur lui expédie alors une description des canons déjà présents dans le jeu (éventuellement aucun) : identifiant, emplacement sur le canevas, orientation et couleur (ligne 3).
- En réponse à l'accusé de réception du client (ligne 4), le serveur installe un nouveau canon dans l'espace de jeu, puis il signale les caractéristiques de cette installation, non seulement au client qui l'a provoquée, mais également à tous les autres clients connectés. Le message expédié au nouveau client comporte cependant une différence (car c'est lui le propriétaire de ce nouveau canon) : en plus des caractéristiques du canon, qui sont fournies à tout le monde, il comporte un champ supplémentaire contenant simplement « le_vôtre » (comparez par exemple la ligne 5 avec la ligne 17, laquelle signale la connexion d'un autre joueur). Cette indication supplémentaire permet au client propriétaire du canon de distinguer parmi plusieurs messages similaires éventuels, celui qui contient l'identifiant unique que lui a attribué le serveur.
- Les messages des lignes 6 et 7 sont des commandes envoyées par le client (réglage de la hausse et commande de tir). Dans la version précédente du jeu, nous avions déjà convenu que les canons se déplaceraient quelque peu (et au hasard) après chaque tir. Le serveur effectue donc cette opération, et s'empresse ensuite d'en faire connaître le résultat à tous les clients connectés. Le message reçu du serveur à la ligne 8 est donc l'indication d'un tel déplacement (les coordonnées fournies sont les coordonnées résultantes pour le canon concerné).
- La ligne 11 reproduit le type de message expédié par le serveur lorsqu'une cible a été touchée. Les nouveaux scores de tous les joueurs sont ainsi communiqués à tous les clients.

- Les messages serveur des lignes 12, 13 et 14 indiquent les actions entreprises par un autre joueur (réglage de hausse suivi d'un tir). Cette fois encore, le canon concerné est déplacé au hasard après qu'il ait tiré (ligne 15).
- Lignes 16 et 17 : lorsque l'un des clients coupe sa connexion, le serveur en avertit tous les autres, afin que le canon correspondant disparaisse de l'espace de jeu sur tous les postes. À l'inverse, de nouveaux clients peuvent se connecter à tout moment pour participer au jeu.

Remarques complémentaires :

Le premier champ de chaque message indique sa teneur. Les messages envoyés par le client sont très simples : ils correspondent aux différentes actions entreprises par le joueur (modifications de l'angle de tir et commandes de feu). Ceux qui sont envoyés par le serveur sont un peu plus complexes. La plupart d'entre eux sont expédiés à tous les clients connectés, afin de les tenir informés du déroulement du jeu. En conséquence, ces messages doivent mentionner l'identifiant du joueur qui a commandé une action ou qui est concerné par un changement quelconque. Nous avons vu plus haut que ces identifiants sont des noms générés automatiquement par le gestionnaire de threads du serveur, chaque fois qu'un nouveau client se connecte.

Certains messages concernant l'ensemble du jeu contiennent plusieurs informations par champ. Dans ce cas, les différents « sous-champs » sont séparés par des points-virgules (lignes 3 et 11).

Programme serveur : première partie

Vous trouverez dans les pages qui suivent le script complet du programme serveur. Nous vous le présentons en trois morceaux successifs afin de rapprocher les commentaires du code correspondant, mais la numérotation de ses lignes est continue. Bien qu'il soit déjà relativement long et complexe, vous estimerez probablement qu'il mérite d'être encore perfectionné, notamment au niveau de la présentation générale. Nous vous laisserons le soin d'y ajouter vous-même tous les compléments qui vous sembleront utiles (par exemple, une proposition de choisir les coordonnées de la machine hôte au démarrage, une barre de menus, etc.) :

```
 1#  ########################################################
 2#  # Jeu des bombardes - partie serveur                   #
 3#  # (C) Gérard Swinnen, Liège (Belgique)-  Juillet 2004 #
 4#  # Licence : GPL                                       #
 5#  # Avant d'exécuter ce script, vérifiez que l'adresse  #
 6#  # IP ci-dessous soit bien celle de la machine hôte.   #
 7#  # Vous pouvez choisir un numéro de port différent, ou #
 8#  # changer les dimensions de l'espace de jeu.          #
 9#  # Dans tous les cas, vérifiez que les mêmes choix ont #
10#  # été effectués pour chacun des scripts clients.      #
11#  ########################################################
12#
13#  host, port = '192.168.0.235', 35000
14#  largeur, hauteur = 700, 400             # dimensions de l'espace de jeu
15#
16#  from Tkinter import *
17#  import socket, sys, threading, time
18#  import canon03
19#  from canon04 import Canon, AppBombardes
20#
21#  class Pupitre(canon03.Pupitre):
22#      """Pupitre de pointage amélioré"""
23#      def __init__(self, boss, canon):
24#          canon03.Pupitre.__init__(self, boss, canon)
25#
26#      def tirer(self):
27#          "déclencher le tir du canon associé"
28#          self.appli.tir_canon(self.canon.id)
```

```
29#
30#     def orienter(self, angle):
31#         "ajuster la hausse du canon associé"
32#         self.appli.orienter_canon(self.canon.id, angle)
33#
34#     def valeur_score(self, sc =None):
35#         "imposer un nouveau score <sc>, ou lire le score existant"
36#         if sc == None:
37#             return self.score
38#         else:
39#             self.score =sc
40#             self.points.config(text = ' %s ' % self.score)
41#
42#     def inactiver(self):
43#         "désactiver le bouton de tir et le système de réglage d'angle"
44#         self.bTir.config(state =DISABLED)
45#         self.regl.config(state =DISABLED)
46#
47#     def activer(self):
48#         "activer le bouton de tir et le système de réglage d'angle"
49#         self.bTir.config(state =NORMAL)
50#         self.regl.config(state =NORMAL)
51#
52#     def reglage(self, angle):
53#         "changer la position du curseur de réglage"
54#         self.regl.config(state =NORMAL)
55#         self.regl.set(angle)
56#         self.regl.config(state =DISABLED)
57#
```

- La classe **Pupitre()** est construite par dérivation de la classe de même nom importée du modune **canon03**. Elle hérite donc toutes les caractéristiques de celle-ci, mais nous devons surcharger[84] ses méthodes **tirer()** et **orienter()** :
- Dans la version monoposte du logiciel, en effet, chacun des pupitres pouvait commander directement l'objet canon correspondant. Dans cette version réseau, par contre, ce sont les clients qui contrôlent à distance le fonctionnement des canons. Par conséquent, les pupitres qui apparaissent dans la fenêtre du serveur ne peuvent être que de simples répétiteurs des manoeuvres effectuées par les joueurs sur chaque client. Le bouton de tir et le curseur de réglage de la hausse sont donc désactivés, mais les indications fournies obéissent aux injonctions qui leur sont adressées par l'application principale.
- Cette nouvelle classe **Pupitre()** sera également utilisée telle quelle dans chaque exemplaire du programme client. Dans la fenêtre de celui-ci comme dans celle du serveur, tous les pupitres seront affichés comme des répétiteurs, mais l'un d'entre eux cependant sera complètement fonctionnel : celui qui correspond au canon du joueur.
- Toutes ces raisons expliquent également l'apparition des nouvelles méthodes : **activer()**, **desactiver()**, **reglage()** et **valeur_score()**, qui seront elles aussi invoquées par l'application principale, en réponse aux messages-instructions échangés entre le serveur et ses clients.
- La classe **ThreadConnexion()** ci-dessous sert à instancier la série d'objets threads qui s'occuperont en parallèle de toutes les connexions lancées par les clients. Sa méthode **run()** contient la fonctionnalité centrale du serveur, à savoir la boucle d'instructions qui gère la réception des messages provenant d'un client particulier, lesquels entraînent chacun toute une cascade de réactions. Vous y trouverez la mise en oeuvre concrète du protocole de communication décrit dans les pages précédentes (certains messages étant cependant générés par les méthodes **depl_aleat_canon()** et **goal()** de la classe **AppServeur()** décrite plus loin).

84 Rappel : dans une classe dérivée, vous pouvez définir une nouvelle méthode avec le même nom qu'une méthode de la classe parente, afin de modifier sa fonctionnalité dans la classe dérivée. Cela s'appelle **surcharger** cette méthode (voir aussi page 176).

```
 58#  class ThreadConnexion(threading.Thread):
 59#      """objet thread gestionnaire d'une connexion client"""
 60#      def __init__(self, boss, conn):
 61#          threading.Thread.__init__(self)
 62#          self.connexion = conn           # réf. du socket de connexion
 63#          self.app = boss                 # réf. de la fenêtre application
 64#
 65#      def run(self):
 66#          "actions entreprises en réponse aux messages reçus du client"
 67#          nom = self.getName()            # id. du client = nom du thread
 68#          while 1:
 69#              msgClient = self.connexion.recv(1024)
 70#              print "**%s** de %s" % (msgClient, nom)
 71#              deb = msgClient.split(',')[0]
 72#              if deb == "fin" or deb =="":
 73#                  self.app.enlever_canon(nom)
 74#                  # signaler le départ de ce canon aux autres clients :
 75#                  self.app.verrou.acquire()
 76#                  for cli in self.app.conn_client:
 77#                      if cli != nom:
 78#                          message = "départ_de,%s" % nom
 79#                          self.app.conn_client[cli].send(message)
 80#                  self.app.verrou.release()
 81#                  # fermer le présent thread :
 82#                  break
 83#              elif deb =="client OK":
 84#                  # signaler au nouveau client les canons déjà enregistrés :
 85#                  msg ="canons,"
 86#                  for g in self.app.guns:
 87#                      gun = self.app.guns[g]
 88#                      msg =msg +"%s;%s;%s;%s;%s," % \
 89#                                (gun.id, gun.x1, gun.y1, gun.sens, gun.coul)
 90#                  self.app.verrou.acquire()
 91#                  self.connexion.send(msg)
 92#                  # attendre un accusé de réception ('OK') :
 93#                  self.connexion.recv(100)
 94#                  self.app.verrou.release()
 95#                  # ajouter un canon dans l'espace de jeu serveur.
 96#                  # la méthode invoquée renvoie les caract. du canon créé :
 97#                  x, y, sens, coul = self.app.ajouter_canon(nom)
 98#                  # signaler les caract. de ce nouveau canon à tous les
 99#                  # clients déjà connectés :
100#                  self.app.verrou.acquire()
101#                  for cli in self.app.conn_client:
102#                      msg ="nouveau_canon,%s,%s,%s,%s,%s" % \
103#                                          (nom, x, y, sens, coul)
104#                      # pour le nouveau client, ajouter un champ indiquant
105#                      # que le message concerne son propre canon :
106#                      if cli == nom:
107#                          msg =msg +",le_vôtre"
108#                      self.app.conn_client[cli].send(msg)
109#                  self.app.verrou.release()
110#              elif deb =='feu':
111#                  self.app.tir_canon(nom)
112#                  # Signaler ce tir à tous les autres clients :
113#                  self.app.verrou.acquire()
114#                  for cli in self.app.conn_client:
115#                      if cli != nom:
116#                          message = "tir_de,%s," % nom
117#                          self.app.conn_client[cli].send(message)
118#                  self.app.verrou.release()
119#              elif deb =="orienter":
120#                  t =msgClient.split(',')
121#                  # Peut-être reçu plusieurs angles. Utiliser le dernier:
122#                  self.app.orienter_canon(nom, t[-2])
123#                  # Signaler ce changement à tous les autres clients :
124#                  self.app.verrou.acquire()
125#                  for cli in self.app.conn_client:
126#                      if cli != nom:
127#                          # virgule à la fin, car messages parfois groupés :
```

```
128#                          message = "angle,%s,%s," % (nom, t[-2])
129#                          self.app.conn_client[cli].send(message)
130#                  self.app.verrou.release()
131#
132#          # Fermeture de la connexion :
133#          self.connexion.close()          # couper la connexion
134#          del self.app.conn_client[nom]   # suppr. sa réf. dans le dictionn.
135#          self.app.afficher("Client %s déconnecté.\n" % nom)
136#          # Le thread se termine ici
137#
```

Synchronisation de threads concurrents à l'aide de « verrous » (thread locks)

Au cours de votre examen du code ci-dessus, vous aurez certainement remarqué la structure particulière des blocs d'instructions par lesquelles le serveur expédie un même message à tous ses clients. Considérez par exemple les lignes 74 à 80 :

La ligne 75 active la méthode **acquire()** d'un objet « verrou » qui a été créé par le constructeur de l'application principale (voir plus loin). Cet objet est une instance de la classe **Lock()**, laquelle fait partie du module **threading** que nous avons importé en début de script. Les lignes suivantes (76 à 79) provoquent l'envoi d'un message à tous les clients connectés (sauf un). Ensuite, l'objet-verrou est à nouveau sollicité, cette fois pour sa méthode **release**().

A quoi cet objet-verrou peut-il donc bien servir ? Puisqu'il est produit par une classe du module **threading**, vous pouvez deviner que son utilité concerne les threads. En fait, de tels objets-verrous servent à synchroniser les *threads concurrents*. De quoi s'agit-il ?

Vous savez que le serveur démarre un thread différent pour chacun des clients qui se connecte. Ensuite, tous ces threads fonctionnent en parallèle. Il existe donc un risque que de temps à autre, deux ou plusieurs de ces threads essaient d'utiliser une ressource commune en même temps.

Dans les lignes de code que nous venons de discuter, par exemple, nous avons affaire à un thread qui souhaite exploiter quasiment toutes les connexions présentes pour poster un message. Il est donc parfaitement possible que pendant ce temps, un autre thread tente d'exploiter lui aussi l'une ou l'autre de ces connexions, ce qui risque de provoquer un dysfonctionnement (en l'occurrence, la superposition chaotique de plusieurs messages).

Un tel problème de *concurrence entre threads* peut être résolu par l'utilisation d'un objet-verrou (*thread lock*). Un tel objet n'est créé qu'en un seul exemplaire, dans un espace de noms accessible à tous les threads concurrents. Il se caractérise essentiellement par le fait qu'il se trouve toujours dans l'un ou l'autre de deux états : soit *verrouillé*, soit *déverrouillé*. Son état initial est l'état déverrouillé.

Utilisation :

Lorsqu'un thread quelconque s'apprête à accéder à une ressource commune, il active d'abord la méthode **acquire()** du verrou. Si celui-ci était dans l'état déverrouillé, il se verrouille, et le thread demandeur peut alors utiliser la ressource commune, en toute tranquillité. Lorsqu'il aura fini d'utiliser la ressource, il s'empressera cependant d'activer la méthode **release()** du verrou, ce qui le fera repasser dans l'état déverrouillé.

En effet : Si un autre thread concurrent essaie d'activer lui aussi la méthode **acquire()** du verrou, alors que celui-ci est dans l'état verrouillé, la méthode « ne rend pas la main », provoquant le blocage de ce thread, lequel suspend donc son activité jusqu'à ce que le verrou repasse dans l'état déverrouillé. Ceci l'empêche donc d'accéder à la ressource commune durant tout le temps où un autre thread s'en sert. Lorsque le verrou est déverrouillé, l'un des threads en attente (il peut en effet y en avoir plusieurs) reprend alors son activité tout en refermant le verrou, et ainsi de suite.

L'objet-verrou mémorise les références des threads bloqués, de manière à n'en débloquer qu'un seul à la fois lorsque sa méthode **release()** est invoquée. Il faut donc toujours veiller à ce que chaque thread qui active la méthode **acquire()** du verrou avant d'accéder à une ressource, active également sa méthode **release()** peu après.

Pour autant que tous les threads concurrents respectent la même procédure, cette technique simple empêche donc qu'une ressource commune soit exploitée en même temps par plusieurs d'entre eux. On dira dans ce cas que les threads ont été *synchronisés*.

Programme serveur : suite et fin

Les deux classes ci-dessous complètent le script serveur. Le code implémenté dans la classe **ThreadClients()** est assez similaire à celui que nous avions développé précédemment pour le corps d'application du logiciel de *Chat*. Dans le cas présent, toutefois, nous le plaçons dans une classe dérivée de **Thread()**, parce que devons faire fonctionner ce code dans un thread indépendant de celui de l'application principale. Celui-ci est en effet déjà complètement accaparé par la boucle **mainloop()** de l'interface graphique[85].

La classe **AppServeur()** dérive de la classe **AppBombardes()** du module **canon04**. Nous lui avons ajouté un ensemble de méthodes complémentaires destinées à exécuter toutes les opérations qui résulteront du dialogue entamé avec les clients. Nous avons déjà signalé plus haut que les clients instancieront chacun une version dérivée de cette classe (afin de profiter des mêmes définitions de base pour la fenêtre, le canevas, etc.).

```
138#  class ThreadClients(threading.Thread):
139#      """objet thread gérant la connexion de nouveaux clients"""
140#      def __init__(self, boss, connex):
141#          threading.Thread.__init__(self)
142#          self.boss = boss                # réf. de la fenêtre application
143#          self.connex = connex            # réf. du socket initial
144#
145#      def run(self):
146#          "attente et prise en charge de nouvelles connexions clientes"
147#          txt ="Serveur prêt, en attente de requêtes ...\n"
148#          self.boss.afficher(txt)
149#          self.connex.listen(5)
150#          # Gestion des connexions demandées par les clients :
151#          while 1:
152#              nouv_conn, adresse = self.connex.accept()
153#              # Créer un nouvel objet thread pour gérer la connexion :
154#              th = ThreadConnexion(self.boss, nouv_conn)
155#              th.start()
156#              it = th.getName()        # identifiant unique du thread
157#              # Mémoriser la connexion dans le dictionnaire :
158#              self.boss.enregistrer_connexion(nouv_conn, it)
159#              # Afficher :
160#              txt = "Client %s connecté, adresse IP %s, port %s.\n" %\
161#                     (it, adresse[0], adresse[1])
162#              self.boss.afficher(txt)
163#              # Commencer le dialogue avec le client :
164#              nouv_conn.send("serveur OK")
165#
166#  class AppServeur(AppBombardes):
167#      """fenêtre principale de l'application (serveur ou client)"""
168#      def __init__(self, host, port, larg_c, haut_c):
169#          self.host, self.port = host, port
170#          AppBombardes.__init__(self, larg_c, haut_c)
171#          self.active =1          # témoin d'activité
172#          # veiller à quitter proprement si l'on referme la fenêtre :
173#          self.bind('<Destroy>',self.fermer_threads)
```

[85] Nous détaillerons cette question quelques pages plus loin, car elle ouvre quelques perspectives intéressantes. Voir : Optimiser les animations à l'aide des threads, page 309.

```
174#
175#     def specificites(self):
176#         "préparer les objets spécifiques de la partie serveur"
177#         self.master.title('<<< Serveur pour le jeu des bombardes >>>')
178#
179#         # widget Text, associé à une barre de défilement :
180#         st =Frame(self)
181#         self.avis =Text(st, width =65, height =5)
182#         self.avis.pack(side =LEFT)
183#         scroll =Scrollbar(st, command =self.avis.yview)
184#         self.avis.configure(yscrollcommand =scroll.set)
185#         scroll.pack(side =RIGHT, fill =Y)
186#         st.pack()
187#
188#         # partie serveur réseau :
189#         self.conn_client = {}           # dictionn. des connexions clients
190#         self.verrou =threading.Lock()   # verrou pour synchroniser threads
191#         # Initialisation du serveur - Mise en place du socket :
192#         connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
193#         try:
194#             connexion.bind((self.host, self.port))
195#         except socket.error:
196#             txt ="La liaison du socket à l'hôte %s, port %s a "\
197#                  "échoué.\n" % (self.host, self.port)
198#             self.avis.insert(END, txt)
199#             self.accueil =None
200#         else:
201#             # démarrage du thread guettant la connexion des clients :
202#             self.accueil = ThreadClients(self, connexion)
203#             self.accueil.start()
204#
205#     def depl_aleat_canon(self, id):
206#         "déplacer aléatoirement le canon <id>"
207#         x, y = AppBombardes.depl_aleat_canon(self, id)
208#         # signaler ces nouvelles coord. à tous les clients :
209#         self.verrou.acquire()
210#         for cli in self.conn_client:
211#             message = "mouvement_de,%s,%s,%s," % (id, x, y)
212#             self.conn_client[cli].send(message)
213#         self.verrou.release()
214#
215#     def goal(self, i, j):
216#         "le canon <i> signale qu'il a atteint l'adversaire <j>"
217#         AppBombardes.goal(self, i, j)
218#         # Signaler les nouveaux scores à tous les clients :
219#         self.verrou.acquire()
220#         for cli in self.conn_client:
221#             msg ='scores,'
222#             for id in self.pupi:
223#                 sc = self.pupi[id].valeur_score()
224#                 msg = msg +"%s;%s," % (id, sc)
225#             self.conn_client[cli].send(msg)
226#         time.sleep(.5)              # pour mieux séparer les messages
227#         self.verrou.release()
228#
229#     def ajouter_canon(self, id):
230#         "instancier un canon et un pupitre de nom <id> dans 2 dictionn."
231#         # on alternera ceux des 2 camps :
232#         n = len(self.guns)
233#         if n %2 ==0:
234#             sens = -1
235#         else:
236#             sens = 1
237#         x, y = self.coord_aleat(sens)
238#         coul =('dark blue', 'dark red', 'dark green', 'purple',
239#                'dark cyan', 'red', 'cyan', 'orange', 'blue', 'violet')[n]
240#         self.guns[id] = Canon(self.jeu, id, x, y, sens, coul)
241#         self.pupi[id] = Pupitre(self, self.guns[id])
242#         self.pupi[id].inactiver()
243#         return (x, y, sens, coul)
```

```
244#
245#     def enlever_canon(self, id):
246#         "retirer le canon et le pupitre dont l'identifiant est <id>"
247#         if self.active == 0:        # la fenêtre a été refermée
248#             return
249#         self.guns[id].effacer()
250#         del self.guns[id]
251#         self.pupi[id].destroy()
252#         del self.pupi[id]
253#
254#     def orienter_canon(self, id, angle):
255#         "régler la hausse du canon <id> à la valeur <angle>"
256#         self.guns[id].orienter(angle)
257#         self.pupi[id].reglage(angle)
258#
259#     def tir_canon(self, id):
260#         "déclencher le tir du canon <id>"
261#         self.guns[id].feu()
262#
263#     def enregistrer_connexion(self, conn, it):
264#         "Mémoriser la connexion dans un dictionnaire"
265#         self.conn_client[it] = conn
266#
267#     def afficher(self, txt):
268#         "afficher un message dans la zone de texte"
269#         self.avis.insert(END, txt)
270#
271#     def fermer_threads(self, evt):
272#         "couper les connexions existantes et fermer les threads"
273#         # couper les connexions établies avec tous les clients :
274#         for id in self.conn_client:
275#             self.conn_client[id].send('fin')
276#         # forcer la fermeture du thread serveur qui attend les requêtes :
277#         if self.accueil != None:
278#             self.accueil._Thread__stop()
279#         self.active =0                  # empêcher accès ultérieurs à Tk
280#
281# if __name__ =='__main__':
282#     AppServeur(host, port, largeur, hauteur).mainloop()
```

Commentaires :

- Ligne 173 : Il vous arrivera de temps à autre de vouloir « intercepter » l'ordre de fermeture de l'application que l'utilisateur déclenche en quittant votre programme, par exemple parce que vous voulez forcer la sauvegarde de données importantes dans un fichier, ou fermer aussi d'autres fenêtres, etc. Il suffit pour ce faire de détecter l'événement **`<Destroy>`**, comme nous le faisons ici pour forcer la terminaison de tous les threads actifs.
- Lignes 179 à 186 : Au passage, voici comment vous pouvez associer une barre de défilement (widget **Scrollbar**) à un widget **Text** (vous pouvez faire de même avec un widget **Canvas**), sans faire appel à la bibliothèque **PMW**[86].
- Ligne 190 : Instanciation de l'objet-verrou permettant de synchroniser les threads.
- Lignes 202, 203 : Instanciation de l'objet thread qui attendra en permanence les demandes de connexion des clients potentiels.
- Lignes 205 à 213, 215 à 227 : Ces méthodes *surchargent* les méthodes de même nom héritées de leur classe parente. Elles commencent par invoquer celles-ci pour effectuer le même travail (lignes 207, 217), puis ajoutent leur fonctionnalité propre, laquelle consiste à signaler à tout le monde ce qui vient de se passer.

[86] Voir : ***Python Mega Widgets***, page 216.

- Lignes 229 à 243 : Cette méthode instancie un nouveau poste de tir, chaque fois qu'un nouveau client se connecte. Les canons sont placés alternativement dans le camp de droite et dans celui de gauche, procédure qui pourrait bien évidemment être améliorée. La liste des couleurs prévues limite le nombre de clients à 10, ce qui devrait suffire.

Programme client

Le script correspondant au logiciel client est reproduit ci-après. Comme celui qui correspond au serveur, il est relativement court, parce qu'il utilise lui aussi l'importation de modules et l'héritage de classes. Le script serveur doit avoir été sauvegardé dans un fichier-module nommé **canon_serveur.py**. Ce fichier doit être placé dans le répertoire courant, de même que les fichiers-modules **canon03.py** et **canon04.py** qu'il utilise lui-même.

De ces modules ainsi importés, le présent script utilise les classes **Canon()** et **Pupitre()** à l'identique, ainsi qu'une forme dérivée de la classe **AppServeur()**. Dans cette dernière, de nombreuses méthodes ont été surchargées, afin d'adapter leur fonctionnalité. Considérez par exemple les méthodes **goal()** et **depl_aleat_canon()**, dont la variante surchargée ne fait plus rien du tout (instruction **pass**), parce que le calcul des scores et le repositionnement des canons après chaque tir ne peuvent être effectués que sur le serveur seulement.

C'est dans la méthode **run()** de la classe **ThreadSocket()** (lignes 86 à 126) que se trouve le code traitant les messages échangés avec le serveur. Nous y avons d'ailleurs laissé une instruction **print** (à la ligne 88) afin que les messages reçus du serveur apparaissent sur la sortie standard. Si vous réalisez vous-même une forme plus définitive de ce jeu, vous pourrez bien évidemment supprimer cette instruction.

```
  1#  ########################################################
  2#  # Jeu des bombardes - partie cliente                   #
  3#  # (C) Gérard Swinnen, Liège (Belgique) - Juillet 2004 #
  4#  # Licence : GPL                                       #
  5#  # Avant d'exécuter ce script, vérifiez que l'adresse, #
  6#  # le numéro de port et les dimensions de l'espace de  #
  7#  # jeu indiquées ci-dessous correspondent exactement   #
  8#  # à ce qui a été défini pour le serveur.              #
  9#  ########################################################
 10#
 11#  from Tkinter import *
 12#  import socket, sys, threading, time
 13#  from canon_serveur import Canon, Pupitre, AppServeur
 14#
 15#  host, port = '192.168.0.235', 35000
 16#  largeur, hauteur = 700, 400          # dimensions de l'espace de jeu
 17#
 18#  class AppClient(AppServeur):
 19#      def __init__(self, host, port, larg_c, haut_c):
 20#          AppServeur.__init__(self, host, port, larg_c, haut_c)
 21#
 22#      def specificites(self):
 23#          "préparer les objets spécifiques de la partie client"
 24#          self.master.title('<<< Jeu des bombardes >>>')
 25#          self.connex =ThreadSocket(self, self.host, self.port)
 26#          self.connex.start()
 27#          self.id =None
 28#
 29#      def ajouter_canon(self, id, x, y, sens, coul):
 30#          "instancier 1 canon et 1 pupitre de nom <id> dans 2 dictionnaires"
 31#          self.guns[id] = Canon(self.jeu, id, int(x),int(y),int(sens), coul)
 32#          self.pupi[id] = Pupitre(self, self.guns[id])
 33#          self.pupi[id].inactiver()
 34#
 35#      def activer_pupitre_personnel(self, id):
 36#          self.id =id                          # identifiant reçu du serveur
```

```
 37#          self.pupi[id].activer()
 38#
 39#      def tir_canon(self, id):
 40#          r = self.guns[id].feu()             # renvoie False si enrayé
 41#          if r and id == self.id:
 42#              self.connex.signaler_tir()
 43#
 44#      def imposer_score(self, id, sc):
 45#          self.pupi[id].valeur_score(int(sc))
 46#
 47#      def deplacer_canon(self, id, x, y):
 48#          "note: les valeurs de x et y sont reçues en tant que chaînes"
 49#          self.guns[id].deplacer(int(x), int(y))
 50#
 51#      def orienter_canon(self, id, angle):
 52#          "régler la hausse du canon <id> à la valeur <angle>"
 53#          self.guns[id].orienter(angle)
 54#          if id == self.id:
 55#              self.connex.signaler_angle(angle)
 56#          else:
 57#              self.pupi[id].reglage(angle)
 58#
 59#      def fermer_threads(self, evt):
 60#          "couper les connexions existantes et refermer les threads"
 61#          self.connex.terminer()
 62#          self.active =0                  # empêcher accès ultérieurs à Tk
 63#
 64#      def depl_aleat_canon(self, id):
 65#          pass                            # => méthode inopérante
 66#
 67#      def goal(self, a, b):
 68#          pass                            # => méthode inopérante
 69#
 70#
 71#  class ThreadSocket(threading.Thread):
 72#      """objet thread gérant l'échange de messages avec le serveur"""
 73#      def __init__(self, boss, host, port):
 74#          threading.Thread.__init__(self)
 75#          self.app = boss            # réf. de la fenêtre application
 76#          # Mise en place du socket - connexion avec le serveur :
 77#          self.connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
 78#          try:
 79#              self.connexion.connect((host, port))
 80#          except socket.error:
 81#              print "La connexion a échoué."
 82#              sys.exit()
 83#          print "Connexion établie avec le serveur."
 84#
 85#      def run(self):
 86#          while 1:
 87#              msg_recu = self.connexion.recv(1024)
 88#              print "*%s*" % msg_recu
 89#              # le message reçu est d'abord converti en une liste :
 90#              t =msg_recu.split(',')
 91#              if t[0] =="" or t[0] =="fin":
 92#                  # fermer le présent thread :
 93#                  break
 94#              elif t[0] =="serveur OK":
 95#                  self.connexion.send("client OK")
 96#              elif t[0] =="canons":
 97#                  self.connexion.send("OK")       # accusé de réception
 98#                  # éliminons le 1er et le dernier élément de la liste.
 99#                  # ceux qui restent sont eux-mêmes des listes :
100#                  lc = t[1:-1]
101#                  # chacune est la description complète d'un canon :
102#                  for g in lc:
103#                      s = g.split(';')
104#                      self.app.ajouter_canon(s[0], s[1], s[2], s[3], s[4])
105#              elif t[0] =="nouveau_canon":
106#                  self.app.ajouter_canon(t[1], t[2], t[3], t[4], t[5])
```

```
107#                 if len(t) >6:
108#                     self.app.activer_pupitre_personnel(t[1])
109#             elif t[0] =='angle':
110#                 # il se peut que l'on ait reçu plusieurs infos regroupées.
111#                 # on ne considère alors que la première :
112#                 self.app.orienter_canon(t[1], t[2])
113#             elif t[0] =="tir_de":
114#                 self.app.tir_canon(t[1])
115#             elif t[0] =="scores":
116#                 # éliminons le 1er et le dernier élément de la liste.
117#                 # ceux qui restent sont eux-mêmes des listes :
118#                 lc = t[1:-1]
119#                 # chaque élément est la description d'un score :
120#                 for g in lc:
121#                     s = g.split(';')
122#                     self.app.imposer_score(s[0], s[1])
123#             elif t[0] =="mouvement_de":
124#                 self.app.deplacer_canon(t[1],t[2],t[3])
125#             elif t[0] =="départ_de":
126#                 self.app.enlever_canon(t[1])
127#
128#         # Le thread <réception> se termine ici.
129#         print "Client arrêté. Connexion interrompue."
130#         self.connexion.close()
131#
132#     def signaler_tir(self):
133#         self.connexion.send('feu')
134#
135#     def signaler_angle(self, angle):
136#         self.connexion.send('orienter,%s,' % angle)
137#
138#     def terminer(self):
139#         self.connexion.send('fin')
140#
141# # Programme principal :
142# if __name__ =='__main__':
143#     AppClient(host, port, largeur, hauteur).mainloop()
```

Commentaires :

- Lignes 15, 16 : Vous pouvez vous-même perfectionner ce script en lui ajoutant un formulaire qui demandera ces valeurs à l'utilisateur au cours du démarrage.
- Lignes 19 à 27 : Le constructeur de la classe parente se termine en invoquant la méthode **specificites()**. On peut donc placer dans celle-ci ce qui doit être construit différemment dans le serveur et dans les clients. (Le serveur instancie notamment un widget **text** qui n'est pas repris dans les clients ; l'un et l'autre démarrent des objets threads différents pour gérer les connexions).
- Lignes 39 à 42 : Cette méthode est invoquée chaque fois que l'utilisateur enfonce le bouton de tir. Le canon ne peut cependant pas effectuer des tirs en rafale. Par conséquent, aucun nouveau tir ne peut être accepté tant que l'obus précédent n'a pas terminé sa trajectoire. C'est la valeur « vraie » ou « fausse » renvoyée par la méthode **feu()** de l'objet canon qui indique si le tir a été accepté ou non. On utilise cette valeur pour ne signaler au serveur (et donc aux autres clients) que les tirs qui ont effectivement eu lieu.
- Lignes 105 à 108 : Un nouveau canon doit être ajouté dans l'espace de jeu de chacun (c'est-à-dire dans le canevas du serveur, et dans le canevas de tous les clients connectés), chaque fois qu'un nouveau client se connecte. Le serveur envoie donc à ce moment un même message à tous les clients pour les informer de la présence de ce nouveau partenaire. Mais le message envoyé à celui-ci en particulier comporte un champ supplémentaire (lequel contient simplement la chaîne « le_vôtre »), afin que ce partenaire sache que ce message concerne son

propre canon, et qu'il puisse donc activer le pupitre correspondant, tout en mémorisant l'identifiant qui lui a été attribué par le serveur (voir également les lignes 35 à 37).

Conclusions et perspectives :

Cette application vous a été présentée dans un but didactique. Nous y avons délibérément simplifié un certain nombre de problèmes. Par exemple, si vous testez vous-même ces logiciels, vous constaterez que les messages échangés sont souvent rassemblés en « paquets », ce qui nécessiterait d'affiner les algorithmes mis en place pour les interpréter. De même, nous avons à peine esquissé le mécanisme fondamental du jeu : répartition des joueurs dans les deux camps, destruction des canons touchés, obstacles divers, etc. Il vous reste bien des pistes à explorer !

18 ***Exercices :***

18.1 Simplifiez le script correspondant au client de *chat* décrit à la page 292, en supprimant l'un des deux objets threads. Arrangez-vous par exemple pour traiter l'émission de messages au niveau du thread principal.

18.2 Modifiez le jeu des bombardes (version monoposte) du chapitre 15 (voir pages 235 et suivantes), en ne gardant qu'un seul canon et un seul pupitre de pointage. Ajoutez-y une cible mobile, dont le mouvement sera géré par un objet thread indépendant (de manière à bien séparer les portions de code qui contrôlent l'animation de la cible et celle du boulet).

Utilisation de threads pour optimiser les animations.

Le dernier exercice proposé à la fin de la section précédente nous suggère une méthodologie de développements d'applications qui peut se révéler particulièrement intéressante, dans le cas de jeux vidéo impliquant plusieurs animations simultanées.

En effet : si vous programmez les différents éléments animés d'un jeu comme des objets indépendants fonctionnant chacun sur son propre thread, alors non seulement vous vous simplifiez la tâche et vous améliorez la lisibilité de votre script, mais encore vous augmentez la vitesse d'exécution et donc la fluidité de ces animations. Pour arriver à ce résultat, vous devrez abandonner la technique de temporisation que vous avez exploitée jusqu'ici, mais celle que vous allez utiliser à sa place est finalement plus simple !

Temporisation des animations à l'aide de after()

Dans toutes les animations que nous avons décrites jusqu'à présent, le « moteur » était constitué à chaque fois par une fonction contenant la méthode **after()**, laquelle est associée d'office à tous les widgets Tkinter. Vous savez que cette méthode permet d'introduire une temporisation dans le déroulement de votre programme : un chronomètre interne est activé, de telle sorte qu'après un intervalle de temps convenu, le système invoque automatiquement une fonction quelconque. En général, c'est la fonction contenant **after()** qui est elle-même invoquée : on réalise ainsi une boucle récursive, dans laquelle il reste à programmer les déplacements des divers objets graphiques.

Vous devez bien comprendre que pendant l'écoulement de l'intervalle de temps programmé à l'aide de la méthode **after()**, votre application n'est pas du tout « figée ». Vous pouvez par exemple pendant ce temps : cliquer sur un bouton, redimensionner la fenêtre, effectuer une entrée clavier, etc. Comment cela est-il rendu possible ?

Nous avons mentionné déjà à plusieurs reprises le fait que les applications graphiques modernes comportent toujours une sorte de moteur qui « tourne » continuellement en tâche de fond : ce dispositif se met en route lorsque vous activez la méthode **mainloop()** de votre fenêtre principale. Comme son nom l'indique fort bien, cette méthode met en œuvre une boucle répétitive perpétuelle, du même type que les boucles **while** que vous connaissez bien. De nombreux mécanismes sont intégrés à ce « moteur ». L'un d'entre eux consiste à réceptionner tous les événements qui se produisent, et à les signaler ensuite à l'aide de messages appropriés aux programmes qui en font la demande (voir : Programmes pilotés par des événements, page 89), d'autres contrôlent les actions à effectuer au niveau de l'affichage, etc. Lorsque vous faites appel à la méthode **after()** d'un widget, vous utilisez en fait un mécanisme de chronométrage qui est intégré lui aussi à **mainloop()**, et c'est donc ce gestionnaire central qui déclenche l'appel de fonction que vous souhaitez, après un certain intervalle de temps.

La technique d'animation utilisant la méthode **after()** est la seule possible pour une application fonctionnant toute entière sur un seul thread, parce que c'est la boucle **mainloop()** qui dirige l'ensemble du comportement d'une telle application de manière absolue. C'est notamment elle qui se charge de redessiner tout ou partie de la fenêtre chaque fois que cela s'avère nécessaire. Pour cette raison, vous ne pouvez pas imaginer de construire un moteur d'animation qui redéfinirait les coordonnées d'un objet graphique à l'intérieur d'une simple boucle **while**, par exemple, parce que pendant tout ce temps l'exécution de **mainloop()** resterait suspendue, ce qui aurait pour conséquence que pendant tout ce temps aucun objet graphique ne serait redessiné (en particulier celui que vous souhaitez mettre en mouvement !). En fait, toute l'application apparaîtrait figée, aussi longtemps que la boucle **while** ne serait pas interrompue.

Puisqu'elle est la seule possible, c'est donc cette technique que nous avons utilisée jusqu'à présent dans tous nos exemples d'applications *mono-thread*. Elle comporte cependant un inconvénient gênant : du fait du grand nombre d'opérations prises en charge à chaque itération de la boucle **mainloop()**, la temporisation que l'on peut programmer à l'aide de **after()** ne peut pas être très courte. Par exemple, elle ne peut guère descendre en dessous de 15 ms sur un PC typique (processeur de type Pentium IV, f = 1,5 GHz). Vous devez tenir compte de cette limitation si vous souhaitez développer des animations rapides.

Un autre inconvénient lié à l'utilisation de la méthode **after()** réside dans la structure de la boucle d'animation (à savoir une fonction ou une méthode « récursive », c'est-à-dire qui s'appelle elle-même) : il n'est pas toujours simple en effet de bien maîtriser ce genre de construction logique, en particulier si l'on souhaite programmer l'animation de plusieurs objets graphiques indépendants, dont le nombre ou les mouvements doivent varier au cours du temps.

Temporisation des animations à l'aide de time.sleep()

Vous pouvez ignorer les limitations de la méthode **after()** évoquées ci-dessus, si vous en confiez l'animation de vos objets graphiques à des threads indépendants. En procédant ainsi, vous vous libérez de la tutelle de **mainloop()**, et il vous est permis alors de construire des procédures d'animation sur la base de structures de boucles plus « classiques », utilisant l'instruction **while** ou l'instruction **for** par exemple.

Au cœur de chacune de ces boucles, vous devez cependant toujours veiller à insérer une temporisation pendant laquelle vous « rendez la main » au système d'exploitation (afin qu'il puisse s'occuper des autres threads). Pour ce faire, vous ferez appel à la fonction **sleep()** du module **time**. Cette fonction permet de suspendre l'exécution du thread courant pendant un certain intervalle de temps, pendant lequel les autres threads et applications continuent à fonctionner. La tempori-

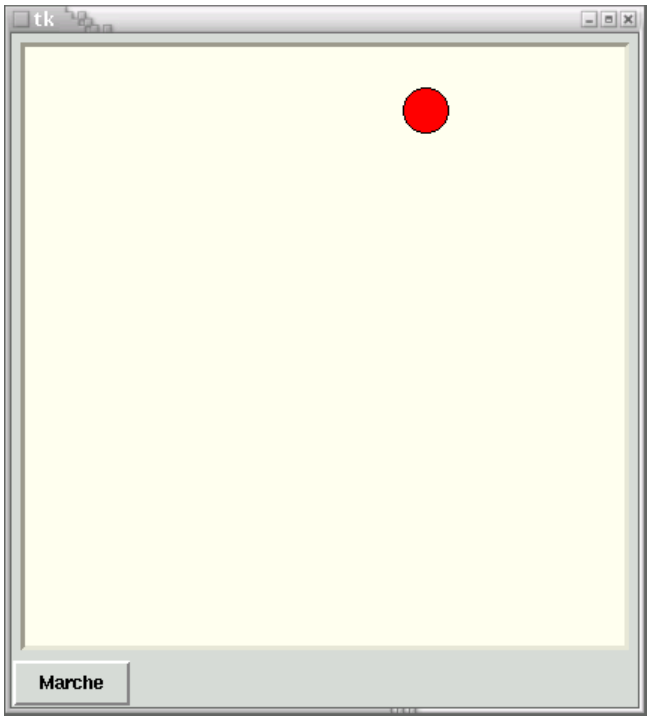
sation ainsi produite ne dépend pas de **mainloop()**, et par conséquent, elle peut être beaucoup plus courte que celle que vous autorise la méthode **after()**.

Attention : cela ne signifie pas que le rafraîchissement de l'écran sera lui-même plus rapide, car ce rafraîchissement continue à être assuré par **mainloop()**. Vous pourrez cependant accélérer fortement les différents mécanismes que vous installez vous-même dans vos procédures d'animation. Dans un logiciel de jeu, par exemple, il est fréquent d'avoir à comparer périodiquement les positions de deux mobiles (tels qu' un projectile et une cible), afin de pouvoir entreprendre une action lorsqu'ils se rejoignent (explosion, ajout de points à un score, etc.). Avec la technique d'animation décrite ici, vous pouvez effectuer beaucoup plus souvent ces comparaisons et donc espérer un résultat plus précis. De même, vous pouvez augmenter le nombre de points pris en considération pour le calcul d'une trajectoire en temps réel, et donc affiner celle-ci.

Remarque : Lorsque vous utilisez la méthode **after()**, vous devez lui indiquer la temporisation souhaitée en millisecondes, sous la forme d'un argument entier. Lorsque vous faites appel à la fonction **sleep()**, par contre, l'argument que vous transmettez doit être exprimé en secondes, sous la forme d'un réel (*float*). Vous pouvez cependant utiliser des très petites valeurs (0.0003 par ex.).

Exemple concret

Le petit script reproduit ci-dessous illustre la mise en oeuvre de cette technique, dans un exemple volontairement minimaliste. Il s'agit d'une petite application graphique dans laquelle une figure se déplace en cercle à l'intérieur d'un canevas. Son « moteur » **mainloop**() est lancé comme d'habitude sur le thread principal. Le constructeur de l'application instancie un canevas contenant le dessin d'un cercle, un bouton et un objet thread. C'est cet objet thread qui assure l'animation du dessin, mais sans faire appel à la méthode **after()** d'un widget. Il utilise plutôt une simple boucle **while** très classique, installée dans sa méthode **run()**.




```
1#  from Tkinter import *
```

```
 2#  from math import sin, cos
 3#  import time, threading
 4#
 5#  class App(Frame):
 6#      def __init__(self):
 7#          Frame.__init__(self)
 8#          self.pack()
 9#          can =Canvas(self, width =400, height =400,
10#                      bg ='ivory', bd =3, relief =SUNKEN)
11#          can.pack(padx =5, pady =5)
12#          cercle = can.create_oval(185, 355, 215, 385, fill ='red')
13#          tb = Thread_balle(can, cercle)
14#          Button(self, text ='Marche', command =tb.start).pack(side =LEFT)
15#          # Button(self, text ='Arrêt', command =tb.stop).pack(side =RIGHT)
16#          # arrêter l'autre thread si l'on ferme la fenêtre :
17#          self.bind('<Destroy>', tb.stop)
18#
19#  class Thread_balle(threading.Thread):
20#      def __init__(self, canevas, dessin):
21#          threading.Thread.__init__(self)
22#          self.can, self.dessin = canevas, dessin
23#          self.anim =1
24#
25#      def run(self):
26#          a = 0.0
27#          while self.anim == 1:
28#              a += .01
29#              x, y = 200 + 170*sin(a), 200 +170*cos(a)
30#              self.can.coords(self.dessin, x-15, y-15, x+15, y+15)
31#              time.sleep(0.010)
32#
33#      def stop(self, evt =0):
34#          self.anim =0
35#
36#  App().mainloop()
```

Commentaires :

- Lignes 13 & 14 : Afin de simplifier notre exemple au maximum, nous créons l'objet thread chargé de l'animation, directement dans le constructeur de l'application principale. Cet objet thread ne démarrera cependant que lorsque l'utilisateur aura cliqué sur le bouton « Marche », qui active sa méthode **start()** (rappelons ici que c'est cette méthode intégrée qui lancera elle-même la méthode **run()** où nous avons installé notre boucle d'animation).
- Ligne 15 : Vous ne pouvez par redémarrer un thread qui s'est terminé. De ce fait, vous ne pouvez lancer cette animation qu'une seule fois (tout au moins sous la forme présentée ici). Pour vous en convaincre, activez la ligne n° 15 en enlevant le caractère # situé au début (et qui fait que Python considère qu'il s'agit d'un simple commentaire) : lorsque l'animation est lancée, un clic de souris sur le bouton ainsi mis en place provoque la sortie de la boucle **while** des lignes 27-31, ce qui termine la méthode **run()**. L'animation s'arrête, mais le thread qui la gérait s'est terminé lui aussi. Si vous essayez de le relancer à l'aide du bouton « Marche », vous n'obtenez rien d'autre qu'un message d'erreur.
- Lignes 26 à 31 : Pour simuler un mouvement circulaire uniforme, il suffit de faire varier continuellement la valeur d'un angle **a**. Le sinus et le cosinus de cet angle permettent alors de calculer les coordonnées **x** et **y** du point de la circonférence qui correspond à cet angle[87].
  À chaque itération, l'angle ne varie que d'un centième de radian seulement (environ 0,6°), et il faudra donc 628 itérations pour que le mobile effectue un tour complet. La temporisation choisie pour ces itérations se trouve à la ligne 31 : 10 millisecondes. Vous pouvez accélérer le

[87] Vous pouvez trouver quelques explications complémentaires à ce sujet, à la page 238.

mouvement en diminuant cette valeur, mais vous ne pourrez guère descendre en dessous de 1 milliseconde (0.001 s), ce qui n'est déjà pas si mal.

> *Rappel : Vous pouvez vous procurer le code source de tous nos exemples sur le site :*
> *http://www.ulg.ac.be/cifen/inforef/swi/python.htm*
> *Vous y trouverez notamment, dans un fichier nommé canon_cibles_multi.py , un petit programme de jeu dans lequel l'utilisateur doit tirer au canon sur une série de cibles mobiles qui deviennent de plus en plus rapides et nombreuses au cours du temps. Ce jeu utilise les techniques d'animation expliquées ci-dessus.*

Annexes

Installation de Python

Si vous souhaitez essayer Python sur votre ordinateur personnel, n'hésitez pas : l'installation est très facile (et parfaitement réversible).

Sous Windows

Sur le site web officiel de Python : **http://www.python.org** , vous trouverez dans la section *Download* des logiciels d'installation automatique pour les différentes versions de Python. Vous pouvez en confiance choisir la dernière version « de production ».

Par exemple, au 11 mars 2005, il s'agissait de la version 2.4.1 - Fichier à télécharger : Python-2.4.1.exe

Copiez ce fichier dans un répertoire temporaire de votre machine, et exécutez-le. Python s'installera par défaut dans un répertoire nommé **Python**** (****** indiquant les deux premiers chiffres du n° de version), et des icônes de lancement seront mises en place automatiquement.

Lorsque l'installation est terminée, vous pouvez effacer le contenu du répertoire temporaire.

Sous Linux

Vous avez probablement installé votre système Linux à l'aide d'une distribution telle que *Ubuntu*, *SuSE*, *RedHat* ... Installez simplement les paquetages Python qui en font partie, en n'omettant pas Tkinter (parfois installé en même temps que la *Python Imaging Library*).

Sous MacOS

Vous trouverez différentes versions de Python pour MacOS 9 et Mac OS X sur le site web de Jack Jansen : *http://homepages.cwi.nl/~jack/macpython*

Installation des Python méga-widgets

Visitez le site web : *http://pmw.sourceforge.net*

et cliquez sur le lien : « Download Pmw12tar.gz » pour télécharger le fichier correspondant.

Décomprimez ce fichier archive dans un répertoire temporaire, à l'aide d'un logiciel de décompression tel que tar, *Winzip*, *Info-Zip*, *unzip* ....

Recopiez l'intégralité du sous-répertoire *Pmw* qui s'est créé automatiquement, dans le répertoire où se trouve déjà l'essentiel de votre installation de Python.

Sous *Windows*, il s'agira par exemple de *C:\Python23*

Sous *Linux*, il s'agira vraisemblablement de */usr/lib/python*

Installation de Gadfly (système de bases de données)

Depuis le site *http://sourceforge.net/projects/gadfly*, télécharger le paquetage *gadfly-1.0.0.tar.gz* Il s'agit d'un fichier archive comprimé. Copiez ce fichier dans un répertoire temporaire.

Sous Windows :

Dans un répertoire temporaire quelconque, décomprimez le fichier archive à l'aide d'un logiciel tel que *Winzip*.

Ouvrez une fenêtre DOS, et entrez dans le sous-répertoire qui s'est créé automatiquement.

Lancez la commande : **`python setup.py install`** C'est tout.

Vous pouvez éventuellement améliorer les performances, en ajoutant l'opération suivante :

Dans le sous-répertoire qui s'est créé, ouvrez le sous-répertoire *kjbuckets*, puis le sous-répertoire qui correspond à votre version de Python. Recopiez le fichier **.pyd* qui s'y trouve dans le répertoire racine de votre installation de Python.

Lorsque tout est terminé, effacez le contenu de votre répertoire temporaire.

Sous Linux :

En tant qu'administrateur (*root*), choisissez un répertoire temporaire quelconque et décomprimez-y le fichier archive à l'aide de l'utilitaire *tar*, qui fait certainement partie de votre distribution. Entrez simplement la commande : **`tar -xvzf gadfly-1.0.0.tar.gz`**

Entrez dans le sous-répertoire qui s'est créé automatiquement : **`cd gadfly-1.0.0`**

Lancez la commande : **`python setup.py install`** C'est tout.

Si votre système Linux comporte un compilateur C, vous pouvez améliorer les performances de Gadfly en recompilant la bibliothèque *kjbuckets*. Pour ce faire, entrez encore les deux commandes suivantes :

```
cd kjbuckets
python setup.py install
```

Lorsque tout est terminé, effacez tout le contenu du répertoire temporaire.

Solutions aux exercices

Pour quelques exercices, nous ne fournissons pas de solution. Efforcez-vous de les trouver sans aide, même si cela vous semble difficile. C'est en effet en vous acharnant sur de tels problèmes que vous apprendrez le mieux.

Exercice 4.2 :

```
>>> c = 0
>>> while c < 20:
...     c = c +1
...     print c, "x 7 =", c*7
```

ou encore :

```
>>> c = 1
>>> while c <= 20:
...     print c, "x 7 =", c*7
...     c = c +1
```

Exercice 4.3 :

```
>>> s = 1
>>> while s <= 16384:
...     print s, "euro(s) =", s *1.65, "dollar(s)"
...     s = s *2
```

Exercice 4.4 :

```
>>> a, c = 1, 1
>>> while c < 13:
...     print a,
...     a, c = a *3, c+1
```

Exercice 4.6 :

```
# Le nombre de secondes est fourni au départ :
# (un grand nombre s'impose !)
nsd = 12345678912
# Nombre de secondes dans une journée :
nspj = 3600 * 24
# Nombre de secondes dans un an (soit 365 jours -
# on ne tiendra pas compte des années bissextiles) :
nspa = nspj * 365
# Nombre de secondes dans un mois (en admettant
# pour chaque mois une durée identique de 30 jours) :
nspm = nspj * 30
# Nombre d'années contenues dans la durée fournie :
na = nsd / nspa          # division <entière>
nsr = nsd % nspa         # n. de sec. restantes
# Nombre de mois restants :
nmo = nsr / nspm         # division <entière>
nsr = nsr % nspm         # n. de sec. restantes
# Nombre de jours restants :
nj = nsr / nspj          # division <entière>
nsr = nsr % nspj         # n. de sec. restantes
# Nombre d'heures restantes :
nh = nsr / 3600          # division <entière>
nsr = nsr % 3600         # n. de sec. restantes
# Nombre de minutes restantes :
nmi = nsr /60            # division <entière>
nsr = nsr % 60           # n. de sec. restantes

print "Nombre de secondes à convertir :", nsd
print "Cette durée correspond à", na, "années de 365 jours, plus"
print nmo, "mois de 30 jours,",
print nj, "jours,",
```

```
print nh, "heures,",
print nmi, "minutes et",
print nsr, "secondes."
```

Exercice 4.7 :

```
# affichage des 20 premiers termes de la table par 7,
# avec signalement des multiples de 3 :

i = 1               # compteur : prendra successivement les valeurs de 1 à 20
while i < 21:
    # calcul du terme à afficher :
    t = i * 7
    # affichage sans saut à la ligne (utilisation de la virgule) :
    print t,
    # ce terme est-il un multiple de 3 ? (utilisation de l'opérateur modulo) :
    if t % 3 == 0:
        print "*",      # affichage d'une astérisque dans ce cas
    i = i + 1           # incrémentation du compteur dans tous les cas
```

Exercice 5.1 :

```
# Conversion degrés -> radians
# Rappel : un angle de 1 radian est un angle qui correspond à une portion
# de circonférence de longueur égale à celle du rayon.
# Puisque la circonférence vaut 2 pi R, un angle de 1 radian correspond
# à 360° / 2 pi , ou encore à 180° / pi

# Angle fourni au départ en degrés, minutes, secondes :
deg, min, sec  = 32, 13, 49

# Conversion des secondes en une fraction de minute :
# (le point décimal force la conversion du résultat en un nombre réel)
fm = sec/60.
# Conversion des minutes en une fraction de degré :
fd = (min + fm)/60
# Valeur de l'angle en degrés "décimalisés" :
ang = deg + fd
# Valeur de pi :
pi = 3.14159265359
# Valeur d'un radian en degrés :
rad = 180 / pi
# Conversion de l'angle en radians :
arad = ang / rad
# Affichage :
print deg, "°", min, "'", sec, '" =', arad, "radian(s)"
```

Exercice 5.3 :

```
# Conversion °Fahrenheit <-> °Celsius

# A) Température fournie en °C :
tempC = 25
# Conversion en °Fahrenheit :
tempF = tempC * 1.8 + 32
# Affichage :
print tempC, "°C =", tempF, "°F"

# B) Température fournie en °F :
tempF = 25
# Conversion en °Celsius :
tempC = (tempF - 32) / 1.8
# Affichage :
print tempF, "°F =", tempC, "°C"
```

Exercice 5.5 :

```
>>> a, b = 1, 1                     # variante :  a, b = 1., 1
>>> while b<65:
...     print b, a
```

```
...     a,b = a*2, b+1
...
```

Exercice 5.6 :

```
# Recherche d'un caractère particulier dans une chaîne

# Chaîne fournie au départ :
ch = "Monty python flying circus"
# Caractère à rechercher :
cr = "e"
# Recherche proprement dite :
lc = len(ch)    # nombre de caractères à tester
i = 0           # indice du caractère en cours d'examen
t = 0           # "drapeau" à lever si le caractère recherché est présent
while i < lc:
    if ch[i] == cr:
        t = 1
    i = i + 1
# Affichage :
print "Le caractère", cr,
if t == 1:
    print "est présent",
else:
    print "est inrouvable",
print "dans la chaîne", ch
```

Exercice 5.8 :

```
# Insertion d'un caractère d'espacement dans une chaîne

# Chaîne fournie au départ :
ch = "Gaston"
# Caractère à insérer :
cr = "*"
# Le nombre de caractères à insérer est inférieur d'une unité au
# nombre de caractères de la chaîne. On traitera donc celle-ci à
# partir de son second caractère (en omettant le premier).
lc = len(ch)    # nombre de caractères total
i = 1           # indice du premier caractère à examiner (le second, en fait)
nch = ch[0]     # nouvelle chaîne à construire (contient déjà le premier car.)
while i < lc:
    nch = nch + cr + ch[i]
    i = i + 1
# Affichage :
print nch
```

Exercice 5.9 :

```
# Inversion d'une chaîne de caractères

# Chaîne fournie au départ :
ch = "zorglub"
lc = len(ch)      # nombre de caractères total
i = lc - 1        # le traitement commencera à partir du dernier caractère
nch = ""          # nouvelle chaîne à construire (vide au départ)
while i >= 0:
    nch = nch + ch[i]
    i = i - 1
# Affichage :
print nch
```

Exercice 5.11 :

```
# Combinaison de deux listes en une seule

# Listes fournies au départ :
t1 = [31,28,31,30,31,30,31,31,30,31,30,31]
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
      'Juillet','Août','Septembre','Octobre','Novembre','Décembre']
# Nouvelle liste à construire (vide au départ) :
```

```
t3 = []
# Boucle de traitement :
i = 0
while i < len(t1):
    t3.append(t2[i])
    t3.append(t1[i])
    i = i + 1

# Affichage :
print t3
```

Exercice 5.12 :

```
# Affichage des éléments d'une liste

# Liste fournie au départ :
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
      'Juillet','Août','Septembre','Octobre','Novembre','Décembre']
# Affichage :
i = 0
while i < len(t2):
    print t2[i],
    i = i + 1
```

Exercice 5.13 :

```
# Recherche du plus grand élément d'une liste

# Liste fournie au départ :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
# Au fur et à mesure du traitement de la liste, on mémorisera dans
# la variable ci-dessous la valeur du plus grand élément déjà trouvé :
max = 0
# Examen de tous les éléments :
i = 0
while i < len(tt):
    if tt[i] > max:
        max = tt[i]         # mémorisation d'un nouveau maximum
    i = i + 1
# Affichage :
print "Le plus grand élément de cette liste a la valeur", max
```

Exercice 5.14 :

```
# Séparation des nombres pairs et impairs

# Liste fournie au départ :
tt = [32, 5, 12, 8, 3, 75, 2, 15]
pairs = []
impairs = []
# Examen de tous les éléments :
i = 0
while i < len(tt):
    if tt[i] % 2 == 0:
        pairs.append(tt[i])
    else:
        impairs.append(tt[i])
    i = i + 1
# Affichage :
print "Nombres pairs :", pairs
print "Nombres impairs :", impairs
```

Exercice 6.1 :

```
# Conversion de miles/heure en km/h et m/s

print "Veuillez entrer le nombre de miles parcourus en une heure : ",
ch = raw_input()            # en général préférable à input()
mph = float(ch)             # conversion de la chaîne entrée en nombre réel
mps = mph * 1609 / 3600     # conversion en mètres par seconde
kmph = mph * 1.609          # conversion en km/h
```

```
# affichage :
print mph, "miles/heure =", kmph, "km/h, ou encore", mps, "m/s"
```

Exercice 6.2 :

```
# Périmètre et Aire d'un triangle quelconque

from math import sqrt

print "Veuillez entrer le côté a : "
a = float(raw_input())
print "Veuillez entrer le côté b : "
b = float(raw_input())
print "Veuillez entrer le côté c : "
c = float(raw_input())
d = (a + b + c)/2                # demi-périmètre
s = sqrt(d*(d-a)*(d-b)*(d-c))    # aire (suivant formule)

print "Longueur des côtés =", a, b, c
print "Périmètre =", d*2, "Aire =", s
```

Exercice 6.4 :

```
# Entrée d'éléments dans une liste

tt = []             # Liste à compléter (vide au départ)
ch = "start"        # valeur quelconque (mais non nulle)
while ch != "":
    print "Veuillez entrer une valeur : "
    ch = raw_input()
    if ch != "":
        tt.append(float(ch))        # variante : tt.append(ch)

# affichage de la liste :
print tt
```

Exercice 6.8 :

```
# Traitement de nombres entiers compris entre deux limites

print "Veuillez entrer la limite inférieure :",
a = input()
print "Veuillez entrer la limite supérieure :",
b = input()
s = 0                    # somme recherchée (nulle au départ)
# Parcours de la série des nombres compris entre a et b :
n = a                    # nombre en cours de traitement
while n <= b:
    if n % 3 ==0 and n % 5 ==0:      # variante : 'or' au lieu de 'and'
        s = s + n
    n = n + 1

print "La somme recherchée vaut", s
```

Exercice 6.9 :

```
# Années bissextiles

print "Veuillez entrer l'année à tester :",
a = input()

if a % 4 != 0:
    # a n'est pas divisible par 4 -> année non bissextile
    bs = 0
else:
    if a % 400 ==0:
        # a divisible par 400 -> année bissextile
        bs = 1
    elif a % 100 ==0:
        # a divisible par 100 -> année non bissextile
        bs = 0
```

```
    else:
        # autres cas ou a est divisible par 4 -> année bissextile
        bs = 1
if bs ==1:
    ch = "est"
else:
    ch = "n'est pas"
print "L'année", a, ch, "bissextile."
```

Variante (proposée par Alex Misbah) :

```
a=input('entrée une année:')

if (a%4==0) and ((a%100!=0) or (a%400==0)):
    print a,"est une année bissextile"
else:
    print a,"n'est pas une année bissextile"
```

Exercice 6.11 : Calculs de triangles

```
from sys import exit      # module contenant des fonctions système

print """
Veuillez entrer les longueurs des 3 côtés
(en séparant ces valeurs à l'aide de virgules) :"""
a, b, c = input()
# Il n'est possible de construire un triangle que si chaque côté
# a une longueur inférieure à la somme des deux autres :
if a < (b+c) and b < (a+c) and c < (a+b) :
    print "Ces trois longueurs déterminent bien un triangle."
else:
    print "Il est impossible de construire un tel triangle !"
    exit()          # ainsi l'on n'ira pas plus loin.

f = 0
if a == b and b == c :
    print "Ce triangle est équilatéral."
    f = 1
elif a == b or b == c or c == a :
    print "Ce triangle est isocèle."
    f = 1
if a*a + b*b == c*c or b*b + c*c == a*a or c*c + a*a == b*b :
    print "Ce triangle est rectangle."
    f = 1
if f == 0 :
    print "Ce triangle est quelconque."
```

Exercice 6.15 :

```
# Notes de travaux scolaires

notes = []          # liste à construire
n = 2               # valeur positive quelconque pour initier la boucle
while n >= 0 :
    print "Entrez la note suivante, s.v.p. : ",
    n = float(raw_input())      # conversion de l'entrée en un nombre réel
    if n < 0 :
        print "OK. Terminé."
    else:
        notes.append(n)         # ajout d'une note à la liste
        # Calculs divers sur les notes déjà entrées :
        # valeurs minimale et maximale + total de toutes les notes.
        min = 500               # valeur supérieure à toute note
        max, tot, i = 0, 0, 0
        nn = len(notes)         # nombre de notes déjà entrées
        while i < nn:
            if notes[i] > max:
                max = notes[i]
            if notes[i] < min:
                min = notes[i]
            tot = tot + notes[i]
```

```
            moy = tot/nn
            i = i + 1
        print nn, "notes entrées. Max =", max, "Min =", min, "Moy =", moy
```

Exercice 7.3 :

```
from math import pi

def surfCercle(r):
    "Surface d'un cercle de rayon r"
    return pi * r**2

# test :
print surfCercle(2.5)
```

Exercice 7.4 :

```
def volBoite(x1, x2, x3):
    "Volume d'une boîte parallélipipédique"
    return x1 * x2 * x3

# test :
print volBoite(5.2, 7.7, 3.3)
```

Exercice 7.5 :

```
def maximum(n1, n2, n3):
    "Renvoie le plus grand de trois nombres"
    if n1 >= n2 and n1 >= n3:
        return n1
    elif n2 >= n1 and n2 >= n3:
        return n2
    else:
        return n3

# test :
print maximum(4.5, 5.7, 3.9)
```

Exercice 7.9 :

```
def compteCar(ca, ch):
    "Renvoie le nombre de caractères ca trouvés dans la chaîne ch"
    i, tot = 0, 0
    while i < len(ch):
        if ch[i] == ca:
            tot = tot + 1
        i = i + 1
    return tot

# test :
print compteCar("e","Cette chaîne est un exemple")
```

Exercice 7.10 :

```
def indexMax(tt):
    "renvoie l'indice du plus grand élément de la liste tt"
    i, max = 0, 0
    while i < len(tt):
        if tt[i] > max :
            max, imax = tt[i], i
        i = i + 1
    return imax

# test :
serie = [5, 8, 2, 1, 9, 3, 6, 4]
print indexMax(serie)
```

Exercice 7.11 :

```
def nomMois(n):
    "renvoie le nom du n-ième mois de l'année"
    mois = ['Janvier,', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet',
```

```
            'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']
    return mois[n -1]       # les indices sont numérotés à partir de zéro

# test :
print nomMois(4)
```

Exercice 7.14 :

```
def volBoite(x1 =10, x2 =10, x3 =10):
    "Volume d'une boîte parallélipipédique"
    return x1 * x2 * x3

# test :
print volBoite()
print volBoite(5.2)
print volBoite(5.2, 3)
```

Exercice 7.15 :

```
def volBoite(x1 =-1, x2 =-1, x3 =-1):
    "Volume d'une boîte parallélipipédique"
    if x1 == -1 :
        return x1           # aucun argument n'a été fourni
    elif x2 == -1 :
        return x1**3        # un seul argument -> boîte cubique
    elif x3 == -1 :
        return x1*x1*x2     # deux arguments -> boîte prismatique
    else :
        return x1*x2*x3

# test :
print volBoite()
print volBoite(5.2)
print volBoite(5.2, 3)
print volBoite(5.2, 3, 7.4)
```

Exercice 7.16 :

```
def changeCar(ch, ca1, ca2, debut =0, fin =-1):
    "Remplace tous les caractères ca1 par des ca2 dans la chaîne ch"
    if fin == -1:
        fin = len(ch)
    nch, i = "", 0            # nch : nouvelle chaîne à construire
    while i < len(ch) :
        if i >= debut and i <= fin and ch[i] == ca1:
            nch = nch + ca2
        else :
            nch = nch + ch[i]
        i = i + 1
    return nch

# test :
print changeCar("Ceci est une toute petite phrase", " ", "*")
print changeCar("Ceci est une toute petite phrase", " ", "*", 8, 12)
print changeCar("Ceci est une toute petite phrase", " ", "*", 12)
```

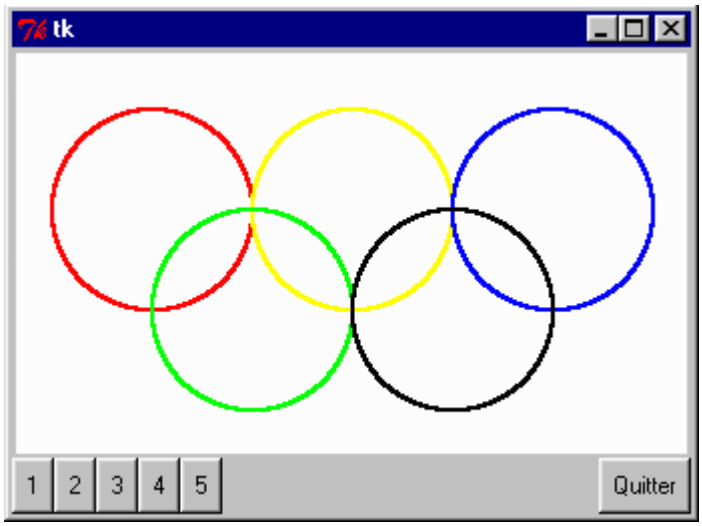
Exercice 7.17 :

```
def eleMax(lst, debut =0, fin =-1):
    "renvoie le plus grand élément de la liste lst"
    if fin == -1:
        fin = len(lst)
    max, i = 0, 0
    while i < len(lst):
        if i >= debut and i <= fin and lst[i] > max:
            max = lst[i]
        i = i + 1
    return max

# test :
serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
```

```
print eleMax(serie)
print eleMax(serie, 2)
print eleMax(serie, 2, 5)
```

Exercice 8.7 :

```
from Tkinter import *

# Coordonnées X,Y des 5 anneaux :
coord = [[20,30], [120,30], [220, 30], [70,80], [170,80]]
# Couleurs des 5 anneaux :
coul = ["red", "yellow", "blue", "green", "black"]

base = Tk()
can = Canvas(base, width =335, height =200, bg ="white")
can.pack()
bou = Button(base, text ="Quitter", command =base.quit)
bou.pack(side = RIGHT)
# Dessin des 5 anneaux :
i = 0
while i < 5:
    x1, y1 = coord[i][0], coord[i][1]
    can.create_oval(x1, y1, x1+100, y1 +100, width =2, outline =coul[i])
    i = i +1
base.mainloop()
```



Variante :

```
from Tkinter import *

# Dessin des 5 anneaux :
def dessineCercle(i):
    x1, y1 = coord[i][0], coord[i][1]
    can.create_oval(x1, y1, x1+100, y1 +100, width =2, outline =coul[i])

def a1():
    dessineCercle(0)

def a2():
    dessineCercle(1)

def a3():
    dessineCercle(2)

def a4():
    dessineCercle(3)

def a5():
    dessineCercle(4)
```

```
# Coordonnées X,Y des 5 anneaux :
coord = [[20,30], [120,30], [220, 30], [70,80], [170,80]]
# Couleurs des 5 anneaux :
coul = ["red", "yellow", "blue", "green", "black"]

base = Tk()
can = Canvas(base, width =335, height =200, bg ="white")
can.pack()
bou = Button(base, text ="Quitter", command =base.quit)
bou.pack(side = RIGHT)

# Installation des 5 boutons :
Button(base, text='1', command = a1).pack(side =LEFT)
Button(base, text='2', command = a2).pack(side =LEFT)
Button(base, text='3', command = a3).pack(side =LEFT)
Button(base, text='4', command = a4).pack(side =LEFT)
Button(base, text='5', command = a5).pack(side =LEFT)
base.mainloop()
```

Exercices 8.9 et 8.10 :

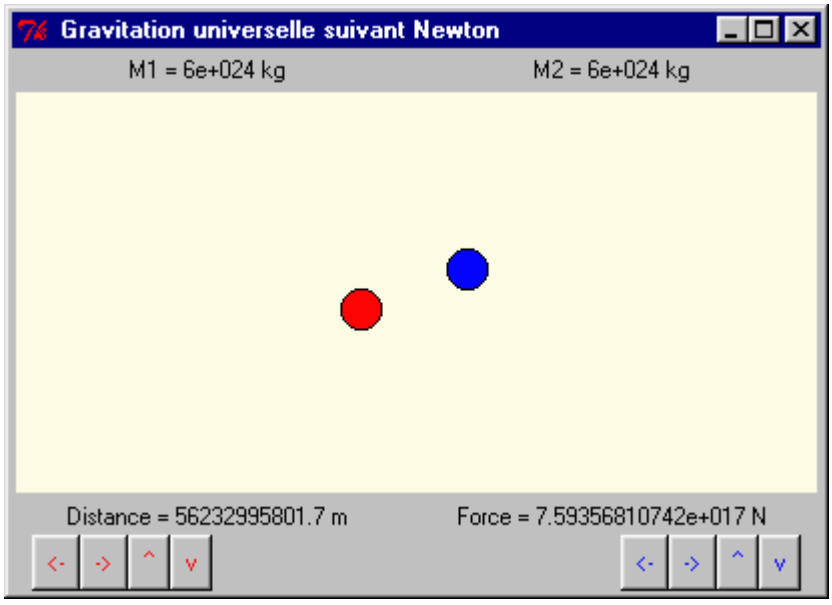
```
# Dessin d'un damier, avec placement de pions au hasard

from Tkinter import *
from random import randrange        # générateur de nombres aléatoires

def damier():
    "dessiner dix lignes de carrés avec décalage alterné"
    y = 0
    while y < 10:
        if y % 2 == 0:              # une fois sur deux, on
            x = 0                   # commencera la ligne de
        else:                       # carrés avec un décalage
            x = 1                   # de la taille d'un carré
        ligne_de_carres(x*c, y*c)
        y += 1

def ligne_de_carres(x, y):
    "dessiner une ligne de carrés, en partant de x, y"
    i = 0
    while i < 10:
        can.create_rectangle(x, y, x+c, y+c, fill='navy')
        i += 1
        x += c*2                    # espacer les carrés

def cercle(x, y, r, coul):
    "dessiner un cercle de centre x,y et de rayon r"
    can.create_oval(x-r, y-r, x+r, y+r, fill=coul)

def ajouter_pion():
    "dessiner un pion au hasard sur le damier"
    # tirer au hasard les coordonnées du pion :
    x = c/2 + randrange(10) * c
    y = c/2 + randrange(10) * c
    cercle(x, y, c/3, 'red')

##### Programme principal : #############

# Tâchez de bien "paramétrer" vos programmes, comme nous l'avons
# fait dans ce script. Celui-ci peut en effet tracer des damiers
# de n'importe quelle taille en changeant seulement la valeur
# d'une seule variable, à savoir la dimension des carrés :

c = 30                  # taille des carrés

fen = Tk()
can = Canvas(fen, width =c*10, height =c*10, bg ='ivory')
can.pack(side =TOP, padx =5, pady =5)
b1 = Button(fen, text ='damier', command =damier)
b1.pack(side =LEFT, padx =3, pady =3)
b2 = Button(fen, text ='pions', command =ajouter_pion)
```

```
b2.pack(side =RIGHT, padx =3, pady =3)
fen.mainloop()#
```

Exercice 8.12 :

```
# Simulation du phénomène de gravitation universelle

from Tkinter import *
from math import sqrt

def distance(x1, y1, x2, y2):
    "distance séparant les points x1,y1 et x2,y2"
    d = sqrt((x2-x1)**2 + (y2-y1)**2)       # théorème de Pythagore
    return  d

def forceG(m1, m2, di):
    "force de gravitation s'exerçant entre m1 et m2 pour une distance di"
    return m1*m2*6.67e-11/di**2             # loi de Newton

def avance(n, gd, hb):
    "déplacement de l'astre n, de gauche à droite ou de haut en bas"
    global x, y, step
    # nouvelles coordonnées :
    x[n], y[n] = x[n] +gd, y[n] +hb
    # déplacement du dessin dans le canevas :
    can.coords(astre[n], x[n]-10, y[n]-10, x[n]+10, y[n]+10)
    # calcul de la nouvelle interdistance :
    di = distance(x[0], y[0], x[1], y[1])
    # conversion de la distance "écran" en distance "astronomique" :
    diA = di*1e9            # (1 pixel => 1 million de km)
    # calcul de la force de gravitation correspondante :
    f = forceG(m1, m2, diA)
    # affichage des nouvelles valeurs de distance et force :
    valDis.configure(text="Distance = " +str(diA) +" m")
    valFor.configure(text="Force = " +str(f) +" N")
    # adaptation du "pas" de déplacement en fonction de la distance :
    step = di/10

def gauche1():
    avance(0, -step, 0)

def droite1():
    avance(0, step, 0)

def haut1():
    avance(0, 0, -step)

def bas1():
    avance(0, 0, step)

def gauche2():
    avance(1, -step, 0)

def droite2():
    avance (1, step, 0)

def haut2():
    avance(1, 0, -step)

def bas2():
    avance(1, 0, step)

# Masses des deux astres :
m1 = 6e24           # (valeur de la masse de la terre, en kg)
m2 = 6e24           #
astre = [0]*2       # liste servant à mémoriser les références des dessins
x =[50., 350.]      # liste des coord. X de chaque astre (à l'écran)
y =[100., 100.]     # liste des coord. Y de chaque astre
step =10            # "pas" de déplacement initial

# Construction de la fenêtre :
```

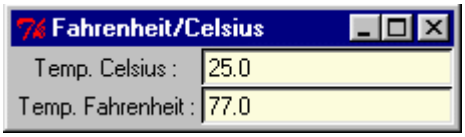
```
fen = Tk()
fen.title(' Gravitation universelle suivant Newton')
# Libellés :
valM1 = Label(fen, text="M1 = " +str(m1) +" kg")
valM1.grid(row =1, column =0)
valM2 = Label(fen, text="M2 = " +str(m2) +" kg")
valM2.grid(row =1, column =1)
valDis = Label(fen, text="Distance")
valDis.grid(row =3, column =0)
valFor = Label(fen, text="Force")
valFor.grid(row =3, column =1)
# Canevas avec le dessin des 2 astres:
can = Canvas(fen, bg ="light yellow", width =400, height =200)
can.grid(row =2, column =0, columnspan =2)
astre[0] = can.create_oval(x[0]-10, y[0]-10, x[0]+10, y[0]+10,
                           fill ="red", width =1)
astre[1] = can.create_oval(x[1]-10, y[1]-10, x[1]+10, y[1]+10,
                           fill ="blue", width =1)
# 2 groupes de 4 boutons, chacun installé dans un cadre (frame) :
fra1 = Frame(fen)
fra1.grid(row =4, column =0, sticky =W, padx =10)
Button(fra1, text="<-", fg ='red',command =gauche1).pack(side =LEFT)
Button(fra1, text="->", fg ='red', command =droite1).pack(side =LEFT)
Button(fra1, text="^", fg ='red', command =haut1).pack(side =LEFT)
Button(fra1, text="v", fg ='red', command =bas1).pack(side =LEFT)
fra2 = Frame(fen)
fra2.grid(row =4, column =1, sticky =E, padx =10)
Button(fra2, text="<-", fg ='blue', command =gauche2).pack(side =LEFT)
Button(fra2, text="->", fg ='blue', command =droite2).pack(side =LEFT)
Button(fra2, text="^", fg ='blue', command =haut2).pack(side =LEFT)
Button(fra2, text="v", fg ='blue', command =bas2).pack(side =LEFT)

fen.mainloop()
```



Exercice 8.16 :

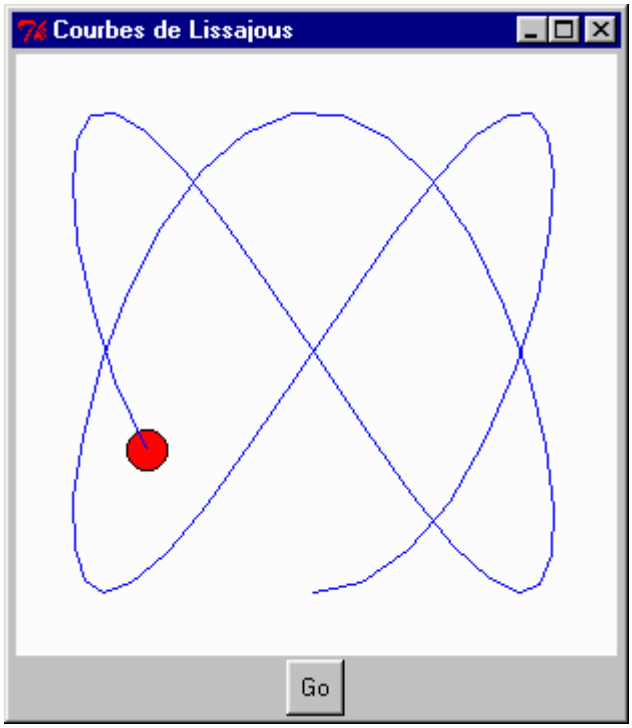
```
# Conversions de températures Fahrenheit <=> Celsius

from Tkinter import *

def convFar(event):
    "valeur de cette température, exprimée en degrés Fahrenheit"
    tF = eval(champTC.get())
    varTF.set(str(tF*1.8 +32))

def convCel(event):
    "valeur de cette température, exprimée en degrés Celsius"
```

```
    tC = eval(champTF.get())
    varTC.set(str((tC-32)/1.8))

fen = Tk()
fen.title('Fahrenheit/Celsius')

Label(fen, text='Temp. Celsius :').grid(row =0, column =0)
# "variable Tkinter" associée au champ d'entrée. Cet "objet-variable"
# assure l'interface entre TCL et Python (voir notes, page 165) :
varTC =StringVar()
champTC = Entry(fen, textvariable =varTC)
champTC.bind("<Return>", convFar)
champTC.grid(row =0, column =1)
# Initialisation du contenu de la variable Tkinter :
varTC.set("100.0")

Label(fen, text='Temp. Fahrenheit :').grid(row =1, column =0)
varTF =StringVar()
champTF = Entry(fen, textvariable =varTF)
champTF.bind("<Return>", convCel)
champTF.grid(row =1, column =1)
varTF.set("212.0")

fen.mainloop()
```



Exercice 8.18 à 8.20 :

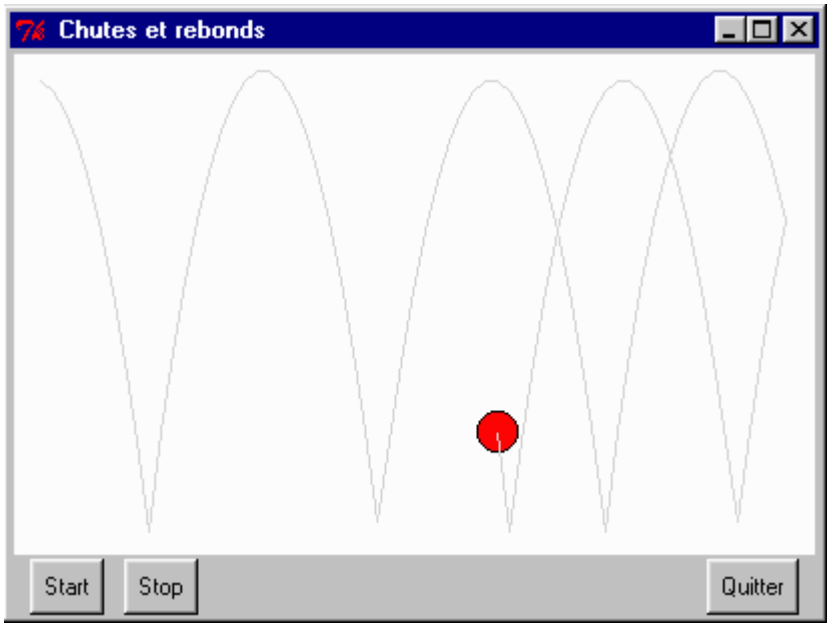
```
# Cercles et courbes de Lissajous

from Tkinter import *
from math import sin, cos

def move():
    global ang, x, y
    # on mémorise les coord. précédentes avant de calculer les nouvelles :
    xp, yp = x, y
    # rotation d'un angle de 0.1 radian :
    ang = ang +.1
    # sinus et cosinus de cet angle => coord. d'un point du cercle trigono.
    x, y = sin(ang), cos(ang)
    # Variante déterminant une courbe de Lissajous avec f1/f2 = 2/3 :
    # x, y = sin(2*ang), cos(3*ang)
    # mise à l'échelle (120 = rayon du cercle, (150,150) = centre du canevas)
    x, y = x*120 + 150, y*120 + 150
    can.coords(balle, x-10, y-10, x+10, y+10)
    can.create_line(xp, yp, x, y, fill ="blue")

ang, x, y = 0., 150., 270.
fen = Tk()
fen.title('Courbes de Lissajous')
can = Canvas(fen, width =300, height=300, bg="white")
can.pack()
balle = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(fen, text='Go', command =move).pack()

fen.mainloop()
```



Exercice 8.27 :

```
# Chutes et rebonds

from Tkinter import *

def move():
    global x, y, v, dx, dv, flag
    xp, yp = x, y            # mémorisation des coord. précédentes
    # déplacement horizontal :
    if x > 385 or x < 15 :   # rebond sur les parois latérales :
        dx = -dx             # on inverse le déplacement
    x = x + dx
    # variation de la vitesse verticale (toujours vers le bas):
    v = v + dv
    # déplacement vertical (proportionnel à la vitesse)
    y = y + v
    if y > 240:              # niveau du sol à 240 pixels :
        y = 240              #  défense d'aller + loin !
        v = -v               # rebond : la vitesse s'inverse
    # on repositionne la balle :
    can.coords(balle, x-10, y-10, x+10, y+10)
    # on trace un bout de trajectoire :
    can.create_line(xp, yp, x, y, fill ='light grey')
    # ... et on remet ça jusqu'à plus soif :
    if flag > 0:
        fen.after(50,move)

def start():
    global flag
    flag = flag +1
    if flag == 1:
        move()

def stop():
    global flag
    flag =0

# initialisation des coordonnées, des vitesses et du témoin d'animation :
x, y, v, dx, dv, flag  = 15, 15, 0, 6, 5, 0

fen = Tk()
```

```
fen.title(' Chutes et rebonds')
can = Canvas(fen, width =400, height=250, bg="white")
can.pack()
balle = can.create_oval(x-10, y-10, x+10, y+10, fill='red')
Button(fen, text='Start', command =start).pack(side =LEFT, padx =10)
Button(fen, text='Stop', command =stop).pack(side =LEFT)
Button(fen, text='Quitter', command =fen.quit).pack(side =RIGHT, padx =10)

fen.mainloop()
```



Exercice 8.33 (Jeu du serpent)

Nous ne fournissons ici qu'une première ébauche du script : le principe d'animation du « serpent ». Si le cœur vous en dit, vous pouvez continuer le développement pour en faire un véritable jeu, mais c'est du travail ! :

```
from Tkinter import *

# === Définition de quelques gestionnaires d'événements :

def start_it():
    "Démarrage de l'animation"
    global flag
    if flag ==0:
        flag =1
        move()

def stop_it():
    "Arrêt de l'animation"
    global flag
    flag =0

def go_left(event =None):
    "délacement vers la gauche"
    global dx, dy
    dx, dy = -1, 0

def go_right(event =None):
    global dx, dy
    dx, dy = 1, 0

def go_up(event =None):
    "déplacement vers le haut"
    global dx, dy
    dx, dy = 0, -1
```

```
def go_down(event =None):
    global dx, dy
    dx, dy = 0, 1

def move():
    "Animation du serpent par récursivité"
    global flag
    # Principe du mouvement opéré : on déplace le carré de queue, dont les
    # caractéristiques sont mémorisées dans le premier élément de la liste
    # <serp>, de manière à l'amener en avant du carré de tête, dont les
    # caractéristiques sont mémorisées dans le dernier élément de la liste.
    # On définit ainsi un nouveau carré de tête pour le serpent, dont on
    # mémorise les caractéristiques en les ajoutant à la liste.
    # Il ne reste plus qu'à effacer alors le premier élément de la liste,
    # et ainsi de suite ... :
    c = serp[0]             # extraction des infos concernant le carré de queue
    cq = c[0]               # réf. de ce carré (coordonnées inutiles ici)
    l =len(serp)            # longueur actuelle du serpent (= n. de carrés)
    c = serp[l-1]           # extraction des infos concernant le carré de tête
    xt, yt = c[1], c[2]     # coordonnées de ce carré
    # Préparation du déplacement proprement dit.
    # (cc est la taille du carré. dx & dy indiquent le sens du déplacement) :
    xq, yq = xt+dx*cc, yt+dy*cc             # coord. du nouveau carré de tête
    # Vérification : a-t-on atteint les limites du canevas ? :
    if xq<0 or xq>canX-cc or yq<0 or yq>canY-cc:
        flag =0             # => arrêt de l'animation
        can.create_text(canX/2, 20, anchor =CENTER, text ="Perdu !!!",
                        fill ="red", font="Arial 14 bold")
    can.coords(cq, xq, yq, xq+cc, yq+cc)    # déplacement effectif
    serp.append([cq, xq, yq])     # mémorisation du nouveau carré de tête
    del(serp[0])                  # effacement (retrait de la liste)
    # Appel récursif de la fonction par elle-même (=> boucle d'animation) :
    if flag >0:
        fen.after(50, move)

# === Programme principal : ========

# Variables globales modifiables par certaines fonctions :
flag =0                 # commutateur pour l'animation
dx, dy = 1, 0           # indicateurs pour le sens du déplacement

# Autres variables globales :
canX, canY = 500, 500   # dimensions du canevas
x, y, cc = 100, 100, 15         # coordonnées et coté du premier carré

# Création de l'espace de jeu (fenêtre, canevas, boutons ...) :
fen =Tk()
can =Canvas(fen, bg ='dark gray', height =canX, width =canY)
can.pack(padx =10, pady =10)
bou1 =Button(fen, text="Start", width =10, command =start_it)
bou1.pack(side =LEFT)
bou2 =Button(fen, text="Stop", width =10, command =stop_it)
bou2.pack(side =LEFT)

# Association de gestionnaires d'événements aux touches fléchées du clavier :
fen.bind("<Left>", go_left)         # Attention : les événements clavier
fen.bind("<Right>", go_right)       # doivent toujours être associés à la
fen.bind("<Up>", go_up)             # fenêtre principale, et non au canevas
fen.bind("<Down>", go_down)         # ou à un autre widget.

# Création du serpent initial (= ligne de 5 carrés).
# On mémorisera les infos concernant les carrés créés dans une liste de
listes :
serp =[]                            # liste vide
# Création et mémorisation des 5 carrés : le dernier (à droite) est la tête.
i =0
while i <5:
    carre =can.create_rectangle(x, y, x+cc, y+cc, fill="green")
    # Pour chaque carré, on mémorise une petite sous-liste contenant
```

```
    # 3 éléments : la référence du carré et ses coordonnées de base :
    serp.append([carre, x, y])
    x =x+cc                 # le carré suivant sera un peu plus à droite
    i =i+1

fen.mainloop()
```

Exercice 9.1 (éditeur simple, pour lire et écrire dans un fichier 'texte') :

```
def sansDC(ch):
    "cette fonction renvoie la chaîne ch amputée de son dernier caractère"
    nouv = ""
    i, j = 0, len(ch) -1
    while i < j:
        nouv = nouv + ch[i]
        i = i + 1
    return nouv

def ecrireDansFichier():
    of = open(nomF, 'a')
    while 1:
        ligne = raw_input("entrez une ligne de texte (ou <Enter>) : ")
        if ligne == '':
            break
        else:
            of.write(ligne + '\n')
    of.close()

def lireDansFichier():
    of = open(nomF, 'r')
    while 1:
        ligne = of.readline()
        if ligne == "":
            break
        # afficher en omettant le dernier caractère (= fin de ligne) :
        print sansDC(ligne)
    of.close()

nomF = raw_input('Nom du fichier à traiter : ')
choix = raw_input('Entrez "e" pour écrire, "c" pour consulter les données : ')

if choix =='e':
    ecrireDansFichier()
else:
    lireDansFichier()
```

Exercice 9.3 (génération des tables de multiplication de 2 à 30) :

```
def tableMulti(n):
    # Fonction générant la table de multiplication par n (20 termes)
    # La table sera renvoyée sous forme d'une chaîne de caractères :
    i, ch = 0, ""
    while i < 20:
        i = i + 1
        ch = ch + str(i * n) + " "
    return ch

NomF = raw_input("Nom du fichier à créer : ")
fichier = open(NomF, 'w')

# Génération des tables de 2 à 30 :
table = 2
while table < 31:
    fichier.write(tableMulti(table) + '\n')
    table = table + 1
fichier.close()
```

Exercice 9.4 :

```
# Triplement des espaces dans un fichier texte.
# Ce script montre également comment modifier le contenu d'un fichier
```

```
# en le transférant d'abord tout entier dans une liste, puis en
# réenregistrant celle-ci après modifications

def triplerEspaces(ch):
    "fonction qui triple les espaces entre mots dans la chaîne ch"
    i, nouv = 0, ""
    while i < len(ch):
        if ch[i] == " ":
            nouv = nouv + "   "
        else:
            nouv = nouv + ch[i]
        i = i +1
    return nouv

NomF = raw_input("Nom du fichier : ")
fichier = open(NomF, 'r+')              # 'r+' = mode read/write
lignes = fichier.readlines()            # lire toutes les lignes

n=0
while n < len(lignes):
    lignes[n] = triplerEspaces(lignes[n])
    n =n+1

fichier.seek(0)                         # retour au début du fichier
fichier.writelines(lignes)              # réenregistrement
fichier.close()
```

Exercice 9.5 :

```
# Mise en forme de données numériques.
# Le fichier traité est un fichier texte dont chaque ligne contient un nombre
# réel (sans exposants et encodé sous la forme d'une chaîne de caractères)

def valArrondie(ch):
    "représentation arrondie du nombre présenté dans la chaîne ch"
    f = float(ch)       # conversion de la chaîne en un nombre réel
    e = int(f + .5)     # conversion en entier (On ajoute d'abord
                        # 0.5 au réel pour l'arrondir correctement)
    return str(e)       # reconversion en chaîne de caractères

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ligne = fs.readline()       # lecture d'une ligne du fichier
    if ligne == "" or ligne == "\n":
        break
    ligne = valArrondie(ligne)
    fd.write(ligne +"\n")

fd.close()
fs.close()
```

Exercice 9.6 :

```
# Comparaison de deux fichiers, caractère par caractère :

fich1 = raw_input("Nom du premier fichier : ")
fich2 = raw_input("Nom du second fichier : ")
fi1 = open(fich1, 'r')
fi2 = open(fich2, 'r')

c, f = 0, 0                 # compteur de caractères et "drapeau"
while 1:
    c = c + 1
    car1 = fi1.read(1)      # lecture d'un caractère dans chacun
    car2 = fi2.read(1)      # des deux fichiers
    if car1 =="" or car2 ="":
        break
```

```
    if car1 != car2 :
        f = 1
        break               # différence trouvée

fi1.close()
fi2.close()

print "Ces 2 fichiers",
if f ==1:
    print "diffèrent à partir du caractère n°", c
else:
    print "sont identiques."
```

Exercice 9.7 :

```
# Combinaison de deux fichiers texte pour en faire un nouveau

fichA = raw_input("Nom du premier fichier : ")
fichB = raw_input("Nom du second fichier : ")
fichC = raw_input("Nom du fichier destinataire : ")
fiA = open(fichA, 'r')
fiB = open(fichB, 'r')
fiC = open(fichC, 'w')

while 1:
    ligneA = fiA.readline()
    ligneB = fiB.readline()
    if ligneA =="" and ligneB =="":
        break               # On est arrivé à la fin des 2 fichiers
    if ligneA != "":
        fiC.write(ligneA)
    if ligneB != "":
        fiC.write(ligneB)

fiA.close()
fiB.close()
fiC.close()
```

Exercice 9.8 :

```
# Enregistrer les coordonnées des membres d'un club

def encodage():
    "renvoie la liste des valeurs entrées, ou une liste vide"
    print "*** Veuillez entrer les données (ou <Enter> pour terminer) :"
    while 1:
        nom = raw_input("Nom : ")
        if nom == "":
            return []
        prenom = raw_input("Prénom : ")
        rueNum = raw_input("Adresse (N° et rue) : ")
        cPost = raw_input("Code postal : ")
        local = raw_input("Localité : ")
        tel = raw_input("N° de téléphone : ")
        print nom, prenom, rueNum, cPost, local, tel
        ver = raw_input("Entrez <Enter> si c'est correct, sinon <n> ")
        if ver == "":
            break
    return [nom, prenom, rueNum, cPost, local, tel]

def enregistrer(liste):
    "enregistre les données de la liste en les séparant par des <#>"
    i = 0
    while i < len(liste):
        of.write(liste[i] + "#")
        i = i + 1
    of.write("\n")              # caractère de fin de ligne

nomF = raw_input('Nom du fichier destinataire : ')
of = open(nomF, 'a')
while 1:
```

```
    tt = encodage()
    if tt == []:
        break
    enregistrer(tt)

of.close()
```

Exercice 9.9 :

```
# Ajouter des informations dans le fichier du club

def traduire(ch):
    "convertir une ligne du fichier source en liste de données"
    dn = ""                 # chaîne temporaire pour extraire les données
    tt = []                 # la liste à produire
    i = 0
    while i < len(ch):
        if ch[i] == "#":
            tt.append(dn)   # on ajoute la donnée à la liste, et
            dn =""          # on réinitialise la chaine temporaire
        else:
            dn = dn + ch[i]
        i = i + 1
    return tt

def encodage(tt):
    "renvoyer la liste tt, complétée avec la date de naissance et le sexe"
    print "*** Veuillez entrer les données (ou <Enter> pour terminer) :"
    # Affichage des données déjà présentes dans la liste :
    i = 0
    while i < len(tt):
        print tt[i],
        i = i +1
    print
    while 1:
        daNai = raw_input("Date de naissance : ")
        sexe = raw_input("Sexe (m ou f) : ")
        print daNai, sexe
        ver = raw_input("Entrez <Enter> si c'est correct, sinon <n> ")
        if ver == "":
            break
    tt.append(daNai)
    tt.append(sexe)
    return tt

def enregistrer(tt):
    "enregistrer les données de la liste tt en les séparant par des <#>"
    i = 0
    while i < len(tt):
        fd.write(tt[i] + "#")
        i = i + 1
    fd.write("\n")          # caractère de fin de ligne

fSource = raw_input('Nom du fichier source : ')
fDest = raw_input('Nom du fichier destinataire : ')
fs = open(fSource, 'r')
fd = open(fDest, 'w')
while 1:
    ligne = fs.readline()           # lire une ligne du fichier source
    if ligne =="" or ligne =="\n":
        break
    liste = traduire(ligne)         # la convertir en une liste
    liste = encodage(liste)         # y ajouter les données supplémentaires
    enregistrer(liste)              # sauvegarder dans fichier dest.

fd.close()
fs.close()
```

Exercice 9.10 :

```
# Recherche de lignes particulières dans un fichier texte :
```

```
def chercheCP(ch):
    "recherche dans ch la portion de chaîne contenant le code postal"
    i, f, ns = 0, 0, 0          # ns est un compteur de codes #
    cc = ""                     # chaîne à construire
    while i < len(ch):
        if ch[i] =="#":
            ns = ns +1
            if ns ==3:          # le CP se trouve après le 3e code #
                f = 1           # variable "drapeau" (flag)
            elif ns ==4:        # inutile de lire après le 4e code #
                break
        elif f ==1:             # le caractère lu fait partie du
            cc = cc + ch[i]     # CP recherché -> on mémorise
        i = i +1
    return cc

nomF = raw_input("Nom du fichier à traiter : ")
codeP = raw_input("Code postal à rechercher : ")
fi = open(nomF, 'r')
while 1:
    ligne = fi.readline()
    if ligne =="":
        break
    if chercheCP(ligne) == codeP:
        print ligne
fi.close()
```

> *Remarque importante : dans les scripts ci-après, nous supposons que votre poste de travail est configuré de manière à utiliser l'encodage Utf-8 par défaut (Versions récentes de Linux, par exemple). Si votre poste de travail utilise un mode d'encodage plus ancien (cas de nombreuses installations de Windows), vous devrez y remplacer chaque occurrence de 'Utf8' par 'Latin1' pour obtenir un traitement correct des caractères accentués.*

Exercice 10.2 (découpage d'une chaîne en fragments) :

```
# -*- coding: Utf8 -*-

def decoupe(ch, n):
    "découpage de la chaîne ch en une liste de fragments de n caractères"
    ch =ch.decode("Utf8")    # Conversion 'string' => 'unicode'
    d, f = 0, n              # indices de début et de fin de fragment
    tt = []                  # liste à construire
    while d < len(ch):
        if f > len(ch):      # on ne peut pas découper au-delà de la fin
            f = len(ch)
        fr = ch[d:f]         # découpage d'un fragment
        tt.append(fr)        # ajout du fragment à la liste
        d, f = f, f +n       # indices suivants
    return tt

def inverse(tt):
    "rassemble les éléments de la liste tt dans l'ordre inverse"
    ch = ""                  # chaîne à construire
    i = len(tt)              # on commence par la fin de la liste
    while i > 0 :
        i = i - 1            # le dernier élément possède l'indice n -1
        ch = ch + tt[i]
    return ch

# Test :
ch ="abcdefghijklmnopqrstuvwxyz123456789âêîôûàèìòùáéíóú"
print ch
liste = decoupe(ch, 5)
print liste
print inverse(liste)
```

Exercices 10.3 & 10.4 :

```
# -*- coding: Utf8 -*-

# Rechercher l'indice d'un caractère donné dans une chaîne

def trouve(ch, car, deb=0):
    "trouve l'indice du caractère car dans la chaîne ch"
    ch = ch.decode("Utf8")      # conversion 'string' => 'unicode'
    i = deb
    while i < len(ch):
        if ch[i] == car:
            return i        # le caractère est trouvé -> on termine
        i = i + 1
    return -1               # toute la chaîne a été scannée sans succès

# Test :
print trouve("Coucou c'est moi", "z")
print trouve("Juliette & Roméo", "&")
print trouve("César & Cléopâtre", "r", 5)
```

Exercice 10.5 :

```
# -*- coding: Utf8 -*-

# Comptage des occurrences d'un caractère donné dans une chaîne

def compteCar(ch, car):
    "trouve l'indice du caractère car dans la chaîne ch"
    ch = ch.decode("Utf8")      # conversion 'string' => 'unicode'
    car = car.decode("Utf8")

    i, nc = 0, 0                # initialisations
    while i < len(ch):
        if ch[i] == car:
            nc = nc + 1         # caractère est trouvé -> on incrémente le
compteur
        i = i + 1
    return nc

# Test :
print compteCar("ananas au jus", "a")
print compteCar("Gédéon est déjà là", "é")
print compteCar("Gédéon est déjà là", "à")
```

Exercice 10.6 :

```
# -*- coding: Utf8 -*-

# Traitement et conversion de lignes dans un petit fichier texte

def traiteLigne(ligne):
    "convertit une ligne de 'Latin1' en 'Utf8', avec insertion de -*-"
    ligne = ligne.decode("Latin1")   # conversion 'string' => 'unicode'
    newLine = u""                    # nouvelle chaîne unicode à construire
    c, m = 0, 0                      # initialisations
    while c < len(ligne):            # lire tous les caractères de la ligne
        if ligne[c] == " ":
            # Le caractère lu est un espace.
            # On ajoute une 'tranche' à la chaîne en cours de construction :
            newLine = newLine + ligne[m:c] + "-*-"
            # On mémorise dans m la position atteinte dans la ligne lue :
            m = c + 1                # ajouter 1 pour "oublier" l'espace
        c = c + 1
    # Ne pas oublier d'ajouter la 'tranche' suivant le dernier espace :
    newLine = newLine + ligne[m:]
    # Renvoyer la chaîne construite, reconcertie en 'string' Utf8 :
    return newLine.encode("Utf8")

# --- Programme principal : ---
nomFS = raw_input("Nom du fichier source (Latin-1) : ")
```

```
nomFD = raw_input("Nom du fichier destinataire (Utf-8) : ")
fs = open(nomFS, 'r')               # ouverture des 2 fichers
fd = open(nomFD, 'w')
while 1:                            # boucle de traitement
    li = fs.readline()              # lecture d'une ligne 'source'
    if li == "":
        break                       # détection de fin de fichier
    fd.write(traiteLigne(li))       # traitement + écriture
fd.close()
fs.close()
```

Exercice 10.7 :

```
prefixes, suffixe = "JKLMNOP", "ack"

for p in prefixes:
    print p + suffixe
```

Exercice 10.8 :

```
def compteMots(ch):
    "comptage du nombre de mots dans la chaîne ch"
    if len(ch) ==0:
        return 0
    nm = 1                  # la chaîne comporte au moins un mot
    for c in ch:
        if c == " ":        # il suffit de compter les espaces
            nm = nm + 1
    return nm

# Test :
print compteMots("Les petits ruisseaux font les grandes rivières")
```

Exercice 10.9 :

```
def chiffre(car):
    "renvoie <vrai> si le caractère 'car' est un chiffre"
    # Cette fonction accepte indifféremment des caractères 'string' ou
    # 'unicode', car les identifiants numériques associés aux chiffres sont
    # les mêmes dans toutes les normes d'encodage.
    if car in "0123456789":
        return 1
    else:
        return 0

# Test :
print chiffre('d'), chiffre('7'), chiffre(u'5'), chiffre(u'é')
```

Exercice 10.10 :

```
# -*- coding:Utf8 -*-

def majuscule(car):
    "renvoie <vrai> si car est une majuscule"
    car = car.decode("Utf8")          # conversion string -> unicode
    if car in u"ABCDEFGHIJKLMNOPQRSTUVWXYZÀÉÈÊËÇÎÏÂÙÔ":
        return 1
    else:
        return 0

# Test :
print majuscule('d'), majuscule('É')
```

Exercice 10.11 :

```
# -*- coding:Utf8 -*-

def chaineListe(ch):
    "convertit la chaîne ch en une liste de mots"
    ch = ch.decode("Utf8")      # conversion string => unicode
    liste, ct = [], u""         # ct est une chaîne temporaire unicode
    for c in ch:                # examiner tous les caractères de ch
```

```
        if c == " ":
            # lorsqu'on rencontre un espace, on ajoute la chaîne temporaire
            # à la liste, après l'avoir reconvertie en string :
            liste.append(ct.encode("Utf8"))
            ct = u""            # ... et on ré-initialise la chaîne temporaire
        else:
            # les autres caractères examinés sont ajoutés à la chaîne temp. :
            ct = ct + c
    # Ne pas oublier le mot restant après le dernier espace ! :
    if ct:                      # vérifier si ct n'est pas une chaîne vide
        liste.append(ct.encode("Utf8"))
    return liste                # renvoyer la liste ainsi construite

# Tests :
li = chaineListe("René est un garçon au caractère héroïque")
print li
for mot in li:
    print mot, "-",
print chaineListe("")           # test d'une chaîne vide
```

Exercice 10.12 (utilise les deux fonctions définies dans les exercices précédents) :

```
# -*- coding:Utf8 -*-

from exercice_10_10 import majuscule
from exercice_10_11 import chaineListe

txt = "Le prénom de cette Dame est Élise"
lst = chaineListe(txt)          # convertir la phrase en une liste de mots
for mot in lst:                 # analyser chacun des mots de la liste
    # Pour extraire le premier caractère du mot, il faut passer par unicode,
    # sinon les caractères accentués ne seront pas corrects :
    motU = mot.decode("Utf8")   # conversion -> unicode
    prem = motU[0]              # extraction du premier caractère
    prem = prem.encode("Utf8")  # re-conversion -> string
    if majuscule(prem):         # test de majuscule
        print mot

# Variante plus compacte, utilisant la composition :
for mot in lst:
    if majuscule(mot.decode("Utf8")[0].encode("Utf8")):
        print mot
```

Exercice 10.13 (utilise les deux fonctions définies dans les exercices précédents) :

```
# -*- coding: Utf8 -*-

from exercice_10_10 import majuscule
from exercice_10_11 import chaineListe

def compteMaj(ch):
    "comptage des mots débutant par une majuscule dans la chaîne ch"
    c = 0
    lst = chaineListe(ch)       # convertir la phrase en une liste de mots
    for mot in lst:             # analyser chacun des mots de la liste
        # Pour tester le premier caractère du mot, il faut passer par unicode,
        # sinon les lettres accentuées ne seront pas traitées correctement :
        if majuscule(mot.decode("Utf8")[0].encode("Utf8")):
            c = c +1
    return c

# Test :
phrase = "Les filles Tidgoutt se nomment Joséphine, Justine et Corinne"
print "Cette phrase contient", compteMaj(phrase), "majuscules."
```

Exercice 10.14 (table des caractères ASCII) :

```
# -*- coding: Utf-8 -*-

# Table des codes ASCII
```

```
c = 32              # premier code ASCII <imprimable>

while c < 128 :     # dernier code strictement ASCII = 127
    print "Code", c, ":", unichr(c), "  ",
    c = c + 1
```

Exercice 10.16 (échange des majuscules et des minuscules) :

```
# -*- coding: Utf8 -*-

def convMajMin(ch):
    "échange les majuscules et les minuscules dans la chaîne ch"
    ch = ch.decode("Utf8")                # conversion -> unicode
    nouvC = u""                           # chaîne à construire
    for car in ch:
        code = ord(car)
        # les codes des maj. et min. sont séparés de 32 unités :
        if code >= 65 and code <= 91:     # majuscules ordinaires
            code = code + 32
        elif code >= 97 and code <= 122:  # minuscules ordinaires
            code = code - 32
        nouvC = nouvC + unichr(code)
    # renvoi de la chaine construite, reconvertie en string :
    return nouvC.encode("Utf8")

# test :
print convMajMin("Roméo et Juliette")
```

Exercice 10.18 (tester si un caractère doonné est une voyelle) :

```
# -*- coding:Utf-8 -*-

def voyelle(cu):
    "teste si le caractère unicode <cu> est une voyelle"
    if cu in u"AEIOUYÀÉÈÊËÎÏÔÛÙaeiouyàéèêëîïôûù":
        return 1
    else:
        return 0

# Test :
print voyelle(u"g"), voyelle(u"O"), voyelle(u"à"), voyelle(u"É")
```

Exercice 10.19 (utilise la fonction définie dans le script précédent) :

```
# -*- coding:Utf-8 -*-

from exercice_10_18 import voyelle

def compteVoyelles(chu):
    "compte les voyelles présentes dans la chaîne unicode chu"
    n = 0
    for c in chu:
        if voyelle(c):
            n = n + 1
    return n

# Test :
phrase ="Maître corbeau sur un arbre perché"
nv = compteVoyelles(phrase.decode("Utf8"))
print "La phrase", phrase, "compte", nv, "voyelles."
```

Exercice 10.20 :

```
# -*- coding:Utf-8 -*-

s1 = u"àéèêëîïôöûüù"
s2 = u""
for c in s1:
    code =ord(c)
    s2 = s2 + unichr(code -32)
```

```
print s1
print s2
```

Exercice 10.21 :

```
# -*- coding:Utf-8 -*-

# Conversion en majuscule du premier caractère de chaque mot dans un texte

fiSource = raw_input("Nom du fichier à traiter (Latin-1) : ")
fiDest = raw_input("Nom du fichier destinataire (Utf-8) : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

while 1:
    ch = fs.readline().decode("Latin1")  # lecture d'une ligne
    if ch == "":
        break                            # fin du fichier
    ch = ch.title()                      # conversion des initiales en maj.
    fd.write(ch.encode("Utf8"))          # transcription

fd.close()
fs.close()
```

Exercice 10.22 :

```
# Comptage du nombre de mots dans un texte

fiSource = raw_input("Nom du fichier à traiter : ")
fs = open(fiSource, 'r')

n = 0           # variable compteur
while 1:
    ch = fs.readline()
    if ch == "":
        break
    # conversion de la chaîne lue en une liste de mots :
    li = ch.split()
    # totalisation des mots :
    n = n + len(li)
fs.close()
print "Ce fichier texte contient un total de %s mots" % (n)
```

Exercice 10.24 :

```
# -*- coding:Utf-8 -*-

# Fusion de lignes pour former des phrases

fiSource = raw_input("Nom du fichier à traiter (Latin-1) : ")
fiDest = raw_input("Nom du fichier destinataire (Utf-8) : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')

# On lit d'abord la première ligne :
ch1 = fs.readline().decode("Latin1")
# On lit ensuite les suivantes, en les fusionnant si nécessaire :
while 1:
    ch2 = fs.readline().decode("Latin1")
    if not ch2:           # Rappel : une chaîne vide est considérée
        break             # comme "fausse" dans les tests
    # Si la chaîne lue commence par une majuscule, on transcrit
    # la précédente dans le fichier destinataire, et on la
    # remplace par celle que l'on vient de lire :
    if ch2[0] in u"ABCDEFGHIJKLMNOPQRSTUVWXYZÀÂÉÈÊËÎÏÔÙÛÇ":
        fd.write(ch1.encode("Utf8"))
        ch1 = ch2
    # Sinon, on la fusionne avec la précédente, en veillant à en
    # enlever au préalable le ou les caractère(s) de fin de ligne.
    # (Pour les fichiers texte sous Windows, il faut en enlever 2) :
    else:
```

```
        ch1 = ch1[:-1] + " " + ch2

# Attention : ne pas oublier de transcrire la dernière ligne :
fd.write(ch1.encode("Utf8"))
fd.close()
fs.close()
```

Exercice 10.25 (caractéristiques de sphères) :

```
# Le fichier de départ est un fichier <texte> dont chaque ligne contient
# un nombre réel (encodé sous la forme d'une chaîne de caractères)

from math import pi

def caractSphere(d):
    "renvoie les caractéristiques d'une sphère de diamètre d"
    d = float(d)        # conversion de l'argument (=chaîne) en réel
    r = d/2             # rayon
    ss = pi*r**2        # surface de section
    se = 4*pi*r**2      # surface extérieure
    v = 4./3*pi*r**3    # volume  (! la 1e division doit être réelle !)
    # Le marqueur de conversion %8.2f utilisé ci-dessous formate le nombre
    # affiché de manière à occuper 8 caractères au total, en arrondissant
    # de manière à conserver deux chiffres après la virgule :
    ch = "Diam. %6.2f cm Section = %8.2f cm² " % (d, ss)
    ch = ch +"Surf. = %8.2f cm². Vol. = %9.2f cm³" % (se, v)
    return ch

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    diam = fs.readline()
    if diam == "" or diam == "\n":
        break
    fd.write(caractSphere(diam) + "\n")         # enregistrement
fd.close()
fs.close()
```

Exercice 10.26 :

```
# Mise en forme de données numériques
# Le fichier traité est un fichier <texte> dont chaque ligne contient un nombre
# réel (sans exposants et encodé sous la forme d'une chaîne de caractères)

def arrondir(reel):
    "représentation arrondie à .0 ou .5 d'un nombre réel"
    ent = int(reel)             # partie entière du nombre
    fra = reel - ent            # partie fractionnaire
    if fra < .25 :
        fra = 0
    elif fra < .75 :
        fra = .5
    else:
        fra = 1
    return ent + fra

fiSource = raw_input("Nom du fichier à traiter : ")
fiDest = raw_input("Nom du fichier destinataire : ")
fs = open(fiSource, 'r')
fd = open(fiDest, 'w')
while 1:
    ligne = fs.readline()
    if ligne == "" or ligne == "\n":
        break
    n = arrondir(float(ligne))      # conversion en <float>, puis arrondi
    fd.write(str(n) + "\n")         # enregistrement

fd.close()
fs.close()
```

Exercice 10.29 :

```
# Affichage de tables de multiplication

nt = [2, 3, 5, 7, 9, 11, 13, 17, 19]

def tableMulti(m, n):
     "renvoie n termes de la table de multiplication par m"
     ch =""
     for i in range(n):
          v = m * (i+1)              # calcul d'un des termes
          ch = ch + "%4d" % (v)      # formatage à 4 caractères
     return ch

for a in nt:
     print tableMulti(a, 15)         # 15 premiers termes seulement
```

Exercice 10.30 (simple parcours d'une liste) :

```
# -*- coding:Utf-8 -*-

lst = ['Jean-Michel', 'Marc', 'Vanessa', 'Anne',
       'Maximilien', 'Alexandre-Benoît', 'Louise']

for e in lst:
     # Le comptage des caractères n'est garanti correct qu'en unicode :
     print "%s : %s caractères" % (e, len(e.decode("Utf8")))
```

Exercice 10.31 :

```
# Elimination de doublons

lst = [9, 12, 40, 5, 12, 3, 27, 5, 9, 3, 8, 22, 40, 3, 2, 4, 6, 25]
lst2 = []

for el in lst:
     if el not in lst2:
          lst2.append(el)

lst2.sort()
print lst2
```

Exercice 10.33 (afficher tous les jours d'une année) :

```
## Cette variante utilise une liste de listes ##
## (que l'on pourrait aisément remplacer par deux listes distinctes)

# La liste ci-dessous contient deux éléments qui sont eux-mêmes des listes.
# l'élément 0 contient les nombres de jours de chaque mois, tandis que
# l'élément 1 contient les noms des douze mois :
mois = [[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
        ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet',
         'Août', 'Septembre', 'Octobre', 'Novembre', 'Décembre']]

jour = ['Dimanche','Lundi','Mardi','Mercredi','Jeudi','Vendredi','Samedi']

ja, jm, js, m = 0, 0, 0, 0

while ja <365:
    ja, jm = ja +1, jm +1    # ja = jour dans l'année, jm = jour dans le mois
    js = (ja +3) % 7         # js = jour de la semaine. Le décalage ajouté
                             #      permet de choisir le jour de départ

    if jm > mois[0][m]:              # élément m de l'élément 0 de la liste
        jm, m = 1, m+1

    print jour[js], jm, mois[1][m]    # élément m de l'élément 1 de la liste
```

Exercice 10.36 :

```
# Insertion de nouveaux éléments dans une liste existante
```

```
t1 = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
t2 = ['Janvier','Février','Mars','Avril','Mai','Juin',
      'Juillet','Août','Septembre','Octobre','Novembre','Décembre']

c, d = 1, 0
while d < 12 :
     t2[c:c] = [t1[d]]       # ! l'élément inséré doit être une liste
     c, d = c+2, d+1
```

Exercice 10.40 :

```
# Crible d'Eratosthène pour rechercher les nombres premiers de 1 à 999

# Créer une liste de 1000 éléments 1 (leurs indices vont de 0 à 999) :
lst = [1]*1000
# Parcourir la liste à partir de l'élément d'indice 2:
for i in range(2,1000):
    # Mettre à zéro les éléments suivants dans la liste,
    # dont les indices sont des multiples de i :
    for j in range(i*2, 1000, i):
        lst[j] = 0

# Afficher les indices des éléments restés à 1 (on ignore l'élément 0) :
for i in range(1,1000):
    if lst[i]:
        print i,
```

Exercice 10.43 (Test du générateur de nombres aléatoires, page 147) :

```
from random import random           # tire au hasard un réel entre 0 et 1

n = raw_input("Nombre de valeurs à tirer au hasard (défaut = 1000) : ")
if n == "":
    nVal =1000
else:
    nVal = int(n)

n = raw_input("Nombre de fractions dans l'intervalle 0-1 (entre 2 et "
              + str(nVal/10) + ", défaut =5) : ")
if n == "":
    nFra =5
else:
    nFra = int(n)

if nFra < 2:
    nFra =2
elif nFra > nVal/10:
    nFra = nVal/10

print "Tirage au sort des", nVal, "valeurs ..."
listVal = [0]*nVal                      # créer une liste de zéros
for i in range(nVal):                   # puis modifier chaque élément
    listVal[i] = random()

print "Comptage des valeurs dans chacune des", nFra, "fractions ..."
listCompt = [0]*nFra                    # créer une liste de compteurs

# parcourir la liste des valeurs :
for valeur in listVal:
    # trouver l'index de la fraction qui contient la valeur :
    index = int(valeur*nFra)
    # incrémenter le compteur correspondant :
    listCompt[index] = listCompt[index] +1

# afficher l'état des compteurs :
for compt in listCompt:
    print compt,
```

Exercice 10.44 : tirage de cartes

```
from random import randrange
```

```
couleurs = ['Pique', 'Trèfle', 'Carreau', 'Coeur']
valeurs = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'valet', 'dame', 'roi', 'as']

# Construction de la liste des 52 cartes :
carte =[]
for coul in couleurs:
     for val in valeurs:
          carte.append("%s de %s" % (str(val), coul))

# Tirage au hasard :
while 1:
     k = raw_input("Frappez <c> pour tirer une carte, <Enter> pour terminer ")
     if k =="":
          break
     r = randrange(52)
     print carte[r]
```

Exercice 10.45 : Création et consultation d'un dictionnaire

```
def consultation():
    while 1:
        nom = raw_input("Entrez le nom (ou <enter> pour terminer) : ")
        if nom == "":
            break
        if dico.has_key(nom):           # le nom est-il répertorié ?
            item = dico[nom]            # consultaion proprement dite
            age, taille = item[0], item[1]
            print "Nom : %s - âge : %s ans - taille : %s m."\
                  % (nom, age, taille)
        else:
            print "*** nom inconnu ! ***"

def remplissage():
    while 1:
        nom = raw_input("Entrez le nom (ou <enter> pour terminer) : ")
        if nom == "":
            break
        age = int(raw_input("Entrez l'âge (nombre entier !) : "))
        taille = float(raw_input("Entrez la taille (en mètres) : "))
        dico[nom] = (age, taille)

dico ={}
while 1:
    choix = raw_input("Choisissez : (R)emplir - (C)onsulter - (T)erminer : ")
    if choix.upper() == 'T':
        break
    elif choix.upper() == 'R':
        remplissage()
    elif choix.upper() == 'C':
        consultation()
```

Exercice 10.46 : échange des clés et des valeurs dans un dictionnaire

```
def inverse(dico):
    "Construction d'un nouveau dico, pas à pas"
    dic_inv ={}
    for cle in dico:
        item = dico[cle]
        dic_inv[item] = cle

    return dic_inv

# programme test :

dico = {'Computer':'Ordinateur',
        'Mouse':'Souris',
        'Keyboard':'Clavier',
        'Hard disk':'Disque dur',
        'Screen':'Écran'}
```

```
print dico
print inverse(dico)
```

Exercice 10.47 : histogramme

```
# -*- coding:Utf-8 -*-

# Histogramme des fréquences de chaque lettre dans un texte

nFich = raw_input('Nom du fichier : ')
fi = open(nFich, 'r')
# Conversion du fichier en une chaîne de caractères unicode.
# Suivant l'encodage du fichier source, activer l'une ou l'autre ligne :

#texte = fi.read().decode("Utf8")
texte = fi.read().decode("Latin1")

fi.close()

print texte
dico ={}
for c in texte:                    # afin de les regrouper, on convertit
    c = c.upper()       # toutes les lettres en majuscules
    dico[c] = dico.get(c, 0) +1

liste = dico.items()
liste.sort()
for car, freq in liste:
    print u"Caractère %s : %s occurrence(s)." % (car, freq)
```

Exercice 10.48 :

```
# -*- coding:Utf-8 -*-

# Histogramme des fréquences de chaque mot dans un texte

nFich = raw_input('Nom du fichier à traiter : ')
fi = open(nFich, 'r')
# Conversion du fichier en une chaîne de caractères unicode.
# Suivant l'encodage du fichier source, activer l'une ou l'autre ligne :

texte = fi.read().decode("Utf8")
#texte = fi.read().decode("Latin1")

fi.close()

# afin de pouvoir aisément séparer les mots du texte, on commence
# par convertir tous les caractères non-alphabétiques en espaces  :

alpha = u"abcdefghijklmnopqrstuvwxyzéèàùçâêîôûäëïöü"

lettres = u""           # nouvelle chaîne à construire (unicode)
for c in texte:
    c = c.lower()       # conversion de chaque caractère en minuscule
    if c in alpha:
        lettres = lettres + c
    else:
        lettres = lettres + ' '

# conversion de la chaîne résultante en une liste de mots :
mots = lettres.split()

# construction de l'histogramme :
dico ={}
for m in mots:
    dico[m] = dico.get(m, 0) +1

liste = dico.items()

# tri de la liste résultante :
liste.sort()
```

```
# affichage en clair :
for item in liste:
    print item[0], ":", item[1]
```

Exercice 10.49 :

```
# -*- coding: Utf-8 -*-
# Encodage d'un texte dans un dictionnaire

nFich = raw_input('Nom du fichier à traiter : ')
fi = open(nFich, 'r')
# Conversion du fichier en une chaîne de caractères unicode.
# Suivant l'encodage du fichier source, activer l'une ou l'autre ligne :

texte = fi.read().decode("Utf8")
#texte = fi.read().decode("Latin1")

fi.close()

# On considère que les mots sont des suites de caractères faisant partie
# de la chaîne ci-dessous. Tous les autres sont des séparateurs :

alpha = u"abcdefghijklmnopqrstuvwxyzéèàùçâêîôûäëïöü"

# Construction du dictionnaire :
dico ={}
# Parcours de tous les caractères du texte :
i =0                    # indice du caractère en cours de lecture
im =-1                  # indice du premier caractère du mot
mot = u""               # variable de travail : mot en cours de lecture
for c in texte:
    c = c.lower()       # conversion de chaque caractère en minuscule

    if c in alpha:      # car. alphabétique => on est à l'intérieur d'un mot
        mot = mot + c
        if im < 0:      # mémoriser l'indice du premier caractère du mot
            im =i
    else:               # car. non-alphabétique => fin de mot
        if mot != u"":  # afin d'ignorer les car. non-alphab. successifs
            # pour chaque mot, on construit une liste d'indices :
            if dico.has_key(mot):       # mot déjà répertorié :
                dico[mot].append(im)    # ajout d'un indice à la liste
            else:                       # mot rencontré pour la 1e fois :
                dico[mot] =[im]         # création de la liste d'indices
            mot =u""    # préparer la lecture du mot suivant
            im =-1
    i += 1              # indice du caractère suivant

# Affichage du dictionnaire, en clair :
listeMots =dico.items()          # Conversion du dico en une liste de tuples
listeMots.sort()                 # tri alphabétique de la liste
for clef, valeur in listeMots:
    print clef, ":", valeur
```

Exercice 10.50 : Sauvegarde d'un dictionnaire (complément de l'ex. 10.45).

```
def enregistrement():
    fich = raw_input("Entrez le nom du fichier de sauvegarde : ")
    ofi = open(fich, "w")
    # parcours du dictionnaire entier, converti au préalable en une liste :
    for cle, valeur in dico.items():
        # utilisation du formatage des chaînes pour créer l'enregistrement :
        ofi.write("%s@%s#%s\n" % (cle, valeur[0], valeur[1]))
    ofi.close()

def lectureFichier():
    fich = raw_input("Entrez le nom du fichier de sauvegarde : ")
    try:
        ofi = open(fich, "r")
    except:
```

```
        print "*** fichier inexistant ***"
        return

    while 1:
        ligne = ofi.readline()
        if ligne =='':              # détection de la fin de fichier
            break
        enreg = ligne.split("@")    # restitution d'une liste [clé,valeur]
        cle = enreg[0]
        valeur = enreg[1][:-1]      # élimination du caractère de fin de ligne
        data = valeur.split("#")    # restitution d'une liste [âge, taille]
        age, taille = int(data[0]), float(data[1])
        dico[cle] = (age, taille)   # reconstitution du dictionnaire
    ofi.close()
```

Ces deux fonctions peuvent être appelées respectivement à la fin et au début du programme principal, comme dans l'exemple ci-dessous :

```
dico ={}
lectureFichier()
while 1:
    choix = raw_input("Choisissez : (R)emplir - (C)onsulter - (T)erminer : ")
    if choix.upper() == 'T':
        break
    elif choix.upper() == 'R':
        remplissage()
    elif choix.upper() == 'C':
        consultation()
enregistrement()
```

Exercice 10.51 : Contrôle du flux d'exécution à l'aide d'un dictionnaire

Cet exercice complète le précédent. On ajoute encore deux petites fonctions, et on réécrit le corps principal du programme pour diriger le flux d'exécution en se servant d'un dictionnaire :

```
def sortie():
    print "*** Job terminé ***"
    return 1                        # afin de provoquer la sortie de la boucle

def autre():
    print "Veuillez frapper R, A, C, S ou T, svp."

dico ={}
fonc ={"R":lectureFichier, "A":remplissage, "C":consultation,
       "S":enregistrement, "T":sortie}
while 1:
    choix = raw_input("Choisissez :\n" +\
    "(R)écupérer un dictionnaire préexistant sauvegardé dans un fichier\n" +\
    "(A)jouter des données au dictionnaire courant\n" +\
    "(C)onsulter le dictionnaire courant\n" +\
    "(S)auvegarder le dictionnaire courant dans un fichier\n" +\
    "(T)erminer : ")
    # l'instruction ci-dessous appelle une fonction différente pour
    # chaque choix, par l'intermédiaire du dictionnaire <fonc> :
    if fonc.get(choix, autre)():
        break
    # Rem : toutes les fonctions appelées ici renvoient <None> par défaut,
    #       sauf la fonction sortie() qui renvoie 1 => sortie de la boucle
```

Exercice 12.1 :

```
class Domino(object):
    def __init__(self, pa, pb):
        self.pa, self.pb = pa, pb

    def affiche_points(self):
        print "face A :", self.pa,
        print "face B :", self.pb
```

```
    def valeur(self):
        return self.pa + self.pb

# Programme de test :

d1 = Domino(2,6)
d2 = Domino(4,3)

d1.affiche_points()
d2.affiche_points()

print "total des points :", d1.valeur() + d2.valeur()

liste_dominos = []
for i in range(7):
    liste_dominos.append(Domino(6, i))

vt =0
for i in range(7):
    liste_dominos[i].affiche_points()
    vt = vt + liste_dominos[i].valeur()

print "valeur totale des points", vt
```

Exercice 12.2 :

```
class CompteBancaire(object):
    def __init__(self, nom ='Dupont', solde =1000):
        self.nom, self.solde = nom, solde

    def depot(self, somme):
        self.solde = self.solde + somme

    def retrait(self, somme):
        self.solde = self.solde - somme

    def affiche(self):
        print "Le solde du compte bancaire de %s est de %s euros." %\
              (self.nom, self.solde)
```

Exercice 12.3 :

```
class Voiture(object):
    def __init__(self, marque = 'Ford', couleur = 'rouge'):
        self.couleur = couleur
        self.marque = marque
        self.pilote = 'personne'
        self.vitesse = 0

    def accelerer(self, taux, duree):
        if self.pilote =='personne':
            print "Cette voiture n'a pas de conducteur !"
        else:
            self.vitesse = self.vitesse + taux * duree

    def choix_conducteur(self, nom):
        self.pilote = nom

    def affiche_tout(self):
            print "%s %s pilotée par %s, vitesse = %s m/s" % \
            (self.marque, self.couleur, self.pilote, self.vitesse)

a1 = Voiture('Peugeot', 'bleue')
a2 = Voiture(couleur = 'verte')
a3 = Voiture('Mercedes')
a1.choix_conducteur('Roméo')
a2.choix_conducteur('Juliette')
a2.accelerer(1.8, 12)
a3.accelerer(1.9, 11)
a2.affiche_tout()
a3.affiche_tout()
```

Exercice 12.4 :

```
class Satellite(object):
    def __init__(self, nom, masse =100, vitesse =0):
        self.nom, self.masse, self.vitesse = nom, masse, vitesse

    def impulsion(self, force, duree):
        self.vitesse = self.vitesse + force * duree / self.masse

    def energie(self):
        return self.masse * self.vitesse**2 / 2

    def affiche_vitesse(self):
        print "Vitesse du satellite %s = %s m/s" \
                            % (self.nom, self.vitesse)

# Programme de test :

s1 = Satellite('Zoé', masse =250, vitesse =10)

s1.impulsion(500, 15)
s1.affiche_vitesse()
print s1.energie()
s1.impulsion(500, 15)
s1.affiche_vitesse()
print s1.energie()
```

Exercices 12.5-12.6 (classes de cylindres et de cônes) :

```
# Classes dérivées - polymorphisme

class Cercle(object):
    def __init__(self, rayon):
        self.rayon = rayon

    def surface(self):
        return 3.1416 * self.rayon**2

class Cylindre(Cercle):
    def __init__(self, rayon, hauteur):
        Cercle.__init__(self, rayon)
        self.hauteur = hauteur

    def volume(self):
        return self.surface()*self.hauteur

        # la méthode surface() est héritée de la classe parente

class Cone(Cylindre):
    def __init__(self, rayon, hauteur):
        Cylindre.__init__(self, rayon, hauteur)

    def volume(self):
        return Cylindre.volume(self)/3

        # cette nouvelle méthode volume() remplace celle que
        # l'on a héritée de la classe parente (exemple de polymorphisme)

cyl = Cylindre(5, 7)
print cyl.surface()
print cyl.volume()

co = Cone(5,7)
print co.surface()
print co.volume()
```

Exercice 12.7 :

```
# Tirage de cartes

from random import randrange
```

```
class JeuDeCartes(object):
    """Jeu de cartes"""
    # attributs de classe (communs à toutes les instances) :
    couleur = ('Pique', 'Trèfle', 'Carreau', 'Coeur')
    valeur = (0, 0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'valet', 'dame', 'roi', 'as')

    def __init__(self):
        "Construction de la liste des 52 cartes"
        self.carte =[]
        for coul in range(4):
            for val in range(13):
                self.carte.append((val +2, coul))   # la valeur commence à 2

    def nom_carte(self, c):
        "Renvoi du nom de la carte c, en clair"
        return "%s de %s" % (self.valeur[c[0]], self.couleur[c[1]])

    def battre(self):
        "Mélange des cartes"
        t = len(self.carte)             # nombre de cartes restantes
        # pour mélanger, on procède à un nombre d'échanges équivalent :
        for i in range(t):
            # tirage au hasard de 2 emplacements dans la liste :
            h1, h2 = randrange(t), randrange(t)
            # échange des cartes situées à ces emplacements :
            self.carte[h1], self.carte[h2] = self.carte[h2], self.carte[h1]

    def tirer(self):
        "Tirage de la première carte de la pile"
        t = len(self.carte)             # vérifier qu'il reste des cartes
        if t >0:
            carte = self.carte[0]       # choisir la première carte du jeu
            del(self.carte[0])          # la retirer du jeu
            return carte                # en renvoyer copie au prog. appelant
        else:
            return None                 # facultatif

### Programme test :

if __name__ == '__main__':
    jeu = JeuDeCartes()                 # instanciation d'un objet
    jeu.battre()                        # mélange des cartes
    for n in range(53):                 # tirage des 52 cartes :
        c = jeu.tirer()
        if c == None:                   # il ne reste aucune carte
            print 'Terminé !'           # dans la liste
        else:
            print jeu.nom_carte(c)      # valeur et couleur de la carte
```

Exercice 12.8 :

```
(On supposera que l'exercice précédent a été sauvegardé sous le nom cartes.py)
# Bataille de de cartes

from cartes import JeuDeCartes

jeuA = JeuDeCartes()        # instanciation du premier jeu
jeuB = JeuDeCartes()        # instanciation du second jeu
jeuA.battre()               # mélange de chacun
jeuB.battre()
pA, pB = 0, 0               # compteurs de points des joueurs A et B

# tirer 52 fois une carte de chaque jeu :
for n in range(52):
    cA, cB = jeuA.tirer(), jeuB.tirer()
    vA, vB = cA[0], cB[0]   # valeurs de ces cartes
    if vA > vB:
        pA += 1
    elif vB > vA:
        pB += 1             # (rien ne se passe si vA = vB)
```

```
    # affichage des points successifs et des cartes tirées :
    print "%s * %s ==> %s * %s" % (jeuA.nom_carte(cA), jeuB.nom_carte(cB), pA,
pB)

print "le joueur A obtient %s points, le joueur B en obtient %s." % (pA, pB)
```

Exercice 12.9 :

```
# -*- coding:Utf-8 -*-

from exercice_12_02 import CompteBancaire

class CompteEpargne(CompteBancaire):
    def __init__(self, nom ='Durand', solde =500):
        CompteBancaire.__init__(self, nom, solde)
        self.taux =.3          # taux d'intérêt mensuel par défaut

    def changeTaux(self, taux):
        self.taux =taux

    def capitalisation(self, nombreMois =6):
        print "Capitalisation sur %s mois au taux mensuel de %s %%." %\
              (nombreMois, self.taux)
        for m in range(nombreMois):
            self.solde = self.solde * (100 +self.taux)/100
```

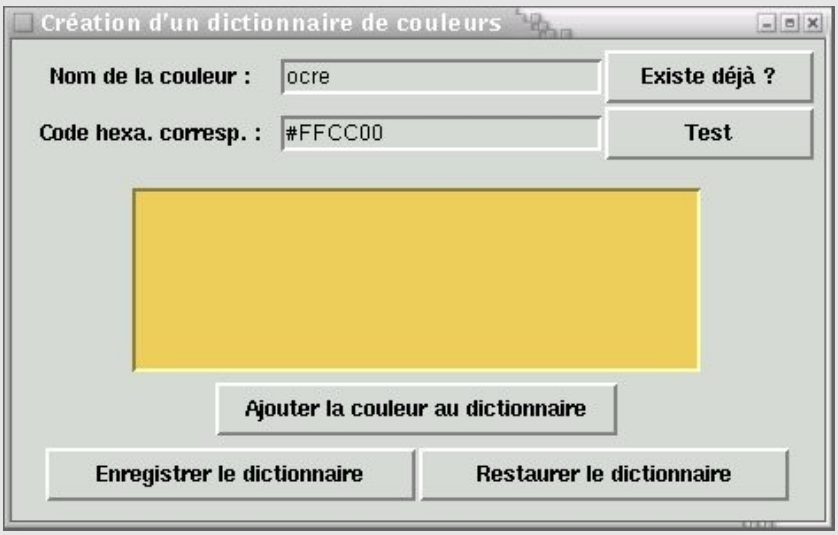
Exercice 13.6 :

```
from Tkinter import *

def cercle(can, x, y, r, coul ='white'):
    "dessin d'un cercle de rayon <r> en <x,y> dans le canevas <can>"
    can.create_oval(x-r, y-r, x+r, y+r, fill =coul)

class Application(Tk):
    def __init__(self):
        Tk.__init__(self)        # constructeur de la classe parente
        self.can =Canvas(self, width =475, height =130, bg ="white")
        self.can.pack(side =TOP, padx =5, pady =5)
        Button(self, text ="Train", command =self.dessine).pack(side =LEFT)
        Button(self, text ="Hello", command =self.coucou).pack(side =LEFT)
        Button(self, text ="Ecl34", command =self.eclai34).pack(side =LEFT)

    def dessine(self):
        "instanciation de 4 wagons dans le canevas"
        self.w1 = Wagon(self.can, 10, 30)
        self.w2 = Wagon(self.can, 130, 30, 'dark green')
        self.w3 = Wagon(self.can, 250, 30, 'maroon')
        self.w4 = Wagon(self.can, 370, 30, 'purple')

    def coucou(self):
        "apparition de personnages dans certaines fenêtres"
        self.w1.perso(3)        # 1er wagon, 3e fenêtre
        self.w3.perso(1)        # 3e wagon, 1e fenêtre
        self.w3.perso(2)        # 3e wagon, 2e fenêtre
        self.w4.perso(1)        # 4e wagon, 1e fenêtre

    def eclai34(self):
        "allumage de l'éclairage dans les wagons 3 & 4"
        self.w3.allumer()
        self.w4.allumer()

class Wagon(object):
    def __init__(self, canev, x, y, coul ='navy'):
        "dessin d'un petit wagon en <x,y> dans le canevas <canev>"
        # mémorisation des paramètres dans des variables d'instance :
        self.canev, self.x, self.y = canev, x, y
        # rectangle de base : 95x60 pixels :
        canev.create_rectangle(x, y, x+95, y+60, fill =coul)
        # 3 fenêtres de 25x40 pixels, écartées de 5 pixels :
        self.fen =[]    # pour mémoriser les réf. des fenêtres
```

```
        for xf in range(x +5, x +90, 30):
            self.fen.append(canev.create_rectangle(xf, y+5,
                                xf+25, y+40, fill ='black'))
        # 2 roues de rayon égal à 12 pixels  :
        cercle(canev, x+18, y+73, 12, 'gray')
        cercle(canev, x+77, y+73, 12, 'gray')

    def perso(self, fen):
        "apparition d'un petit personnage à la fenêtre <fen>"
        # calcul des coordonnées du centre de chaque fenêtre :
        xf = self.x + fen*30 -12
        yf = self.y + 25
        cercle(self.canev, xf, yf, 10, "pink")      # visage
        cercle(self.canev, xf-5, yf-3, 2)   # oeil gauche
        cercle(self.canev, xf+5, yf-3, 2)   # oeil droit
        cercle(self.canev, xf, yf+5, 3)     # bouche

    def allumer(self):
        "déclencher l'éclairage interne du wagon"
        for f in self.fen:
            self.canev.itemconfigure(f, fill ='yellow')

Application().app.mainloop()
```

Exercice 13.22 :

```
# Dictionnaire de couleurs
```




```
from Tkinter import *
# Module donnant accès aux boîtes de dialogue standard pour
# la recherche de fichiers sur disque :
from tkFileDialog import asksaveasfile, askopenfile

class Application(Frame):
    '''Fenêtre d'application'''
    def __init__(self):
        Frame.__init__(self)
        self.master.title("Création d'un dictionnaire de couleurs")

        self.dico ={}       # création du dictionnaire

        # Les widgets sont regroupés dans deux cadres (Frames) :
        frSup =Frame(self)      # cadre supérieur contenant 6 widgets
        Label(frSup, text ="Nom de la couleur :",
              width =20).grid(row =1, column =1)
        self.enNom =Entry(frSup, width =25)        # champ d'entrée pour
        self.enNom.grid(row =1, column =2)         # le nom de la couleur
        Button(frSup, text ="Existe déjà ?", width =12,
               command =self.chercheCoul).grid(row =1, column =3)
```

```
        Label(frSup, text ="Code hexa. corresp. :",
              width =20).grid(row =2, column =1)
        self.enCode =Entry(frSup, width =25)        # champ d'entrée pour
        self.enCode.grid(row =2, column =2)         # le code hexa.
        Button(frSup, text ="Test", width =12,
               command =self.testeCoul).grid(row =2, column =3)
        frSup.pack(padx =5, pady =5)

        frInf =Frame(self)      # cadre inférieur contenant le reste
        self.test = Label(frInf, bg ="white", width =45,    # zone de test
                          height =7, relief = SUNKEN)
        self.test.pack(pady =5)
        Button(frInf, text ="Ajouter la couleur au dictionnaire",
               command =self.ajouteCoul).pack()
        Button(frInf, text ="Enregistrer le dictionnaire", width =25,
               command =self.enregistre).pack(side = LEFT, pady =5)
        Button(frInf, text ="Restaurer le dictionnaire", width =25,
               command =self.restaure).pack(side =RIGHT, pady =5)
        frInf.pack(padx =5, pady =5)
        self.pack()

    def ajouteCoul(self):
        "ajouter la couleur présente au dictionnaire"
        if self.testeCoul() ==0:        # une couleur a-t-elle été définie ?
            return
        nom = self.enNom.get()
        if len(nom) >1:                 # refuser les noms trop petits
            self.dico[nom] =self.cHexa
        else:
            self.test.config(text ="%s : nom incorrect" % nom, bg ='white')

    def chercheCoul(self):
        "rechercher une couleur déjà inscrite au dictionnaire"
        nom = self.enNom.get()
        if self.dico.has_key(nom):
            self.test.config(bg =self.dico[nom], text ="")
        else:
            self.test.config(text ="%s : couleur inconnue" % nom, bg ='white')

    def testeCoul(self):
        "vérifier la validité d'un code hexa. - afficher la couleur corresp."
        try:
            self.cHexa =self.enCode.get()
            self.test.config(bg =self.cHexa, text ="")
            return 1
        except:
            self.test.config(text ="Codage de couleur incorrect", bg ='white')
            return 0

    def enregistre(self):
        "enregistrer le dictionnaire dans un fichier texte"
        # Cette méthode utilise une boîte de dialogue standard pour la
        # sélection d'un fichier sur disque. Tkinter fournit toute une série
        # de fonctions associées à ces boîtes, dans le module tkFileDialog.
        # La fonction ci-dessous renvoie un objet-fichier ouvert en écriture :
        ofi =asksaveasfile(filetypes=[("Texte",".txt"),("Tous","*")])
        for clef, valeur in self.dico.items():
            ofi.write("%s %s\n" % (clef, valeur))
        ofi.close()

    def restaure(self):
        "restaurer le dictionnaire à partir d'un fichier de mémorisation"
        # La fonction ci-dessous renvoie un objet-fichier ouvert en lecture :
        ofi =askopenfile(filetypes=[("Texte",".txt"),("Tous","*")])
        lignes = ofi.readlines()
        for li in lignes:
            cv = li.split()     # extraction de la clé et la valeur corresp.
            self.dico[cv[0]] = cv[1]
        ofi.close()
```

```
if __name__ == '__main__':
    Application().mainloop()
```

Exercice 13.23 (variante 3) :

```
from Tkinter import *
from random import randrange
from math import sin, cos, pi

class FaceDom(object):
    def __init__(self, can, val, pos, taille =70):
        self.can =can
        x, y, c = pos[0], pos[1], taille/2
        self. carre = can.create_rectangle(x -c, y-c, x+c, y+c,
                                           fill ='ivory', width =2)
        d = taille/3
        # disposition des points sur la face, pour chacun des 6 cas :
        self.pDispo = [((0,0),),
                       ((-d,d),(d,-d)),
                       ((-d,-d), (0,0), (d,d)),
                       ((-d,-d),(-d,d),(d,-d),(d,d)),
                       ((-d,-d),(-d,d),(d,-d),(d,d),(0,0)),
                       ((-d,-d),(-d,d),(d,-d),(d,d),(d,0),(-d,0))]

        self.x, self.y, self.dim = x, y, taille/15
        self.pList =[]      # liste contenant les points de cette face
        self.tracer_points(val)

    def tracer_points(self, val):
        # créer les dessins de points correspondant à la valeur val :
        disp = self.pDispo[val -1]
        for p in disp:
            self.cercle(self.x +p[0], self.y +p[1], self.dim, 'red')
        self.val = val

    def cercle(self, x, y, r, coul):
        self.pList.append(self.can.create_oval(x-r, y-r, x+r, y+r, fill=coul))

    def effacer(self, flag =0):
        for p in self.pList:
            self.can.delete(p)
        if flag:
            self.can.delete(self.carre)

class Projet(Frame):
    def __init__(self, larg, haut):
        Frame.__init__(self)
        self.larg, self.haut = larg, haut
        self.can = Canvas(self, bg='dark green', width =larg, height =haut)
        self.can.pack(padx =5, pady =5)
        # liste des boutons à installer, avec leur gestionnaire :
        bList = [("A", self.boutA), ("B", self.boutB),
                 ("C", self.boutC), ("Quitter", self.boutQuit)]
        bList.reverse()         # inverser l'ordre de la liste
        for b in bList:
            Button(self, text =b[0], command =b[1]).pack(side =RIGHT, padx=3)
        self.pack()
        self.des =[]            # liste qui contiendra les faces de dés
        self.actu =0            # réf. du dé actuellement sélectionné

    def boutA(self):
        if len(self.des):
            return              # car les dessins existent déjà !
        a, da = 0, 2*pi/13
        for i in range(13):
            cx, cy = self.larg/2, self.haut/2
            x = cx + cx*0.75*sin(a)             # pour disposer en cercle,
            y = cy + cy*0.75*cos(a)             # on utilise la trigono !
            self.des.append(FaceDom(self.can, randrange(1,7) , (x,y), 65))
            a += da
```

```
    def boutB(self):
        # incrémenter la valeur du dé sélectionné. Passer au suivant :
        v = self.des[self.actu].val
        v = v % 6
        v += 1
        self.des[self.actu].effacer()
        self.des[self.actu].tracer_points(v)
        self.actu += 1
        self.actu = self.actu % 13

    def boutC(self):
        for i in range(len(self.des)):
            self.des[i].effacer(1)
        self.des =[]
        self.actu =0

    def boutQuit(self):
        self.master.destroy()

Projet(600, 600).mainloop()
```

Exercice 16.1 (Création de la base de données "musique") :

```
import gadfly

connex = gadfly.gadfly()
connex.startup("musique","E:/Python/essais/gadfly")
cur = connex.cursor()
requete = "create table compositeurs (comp varchar, a_naiss integer,\
          a_mort integer)"
cur.execute(requete)
requete = "create table oeuvres (comp varchar, titre varchar,\
          duree integer, interpr varchar)"
cur.execute(requete)

print "Entrée des enregistrements, table des compositeurs :"
while 1:
    nm = raw_input("Nom du compositeur (<Enter> pour terminer) : ")
    if nm =='':
        break
    an = raw_input("Année de naissance : ")
    am = raw_input("Année de mort : ")
    requete ="insert into compositeurs(comp, a_naiss, a_mort) values \
                ('%s', %s, %s)" % (nm, an, am)
    cur.execute(requete)
# Affichage des données entrées, pour vérification :
cur.execute("select * from compositeurs")
print cur.pp()

print "Entrée des enregistrements, table des oeuvres musicales :"
while 1:
    nom = raw_input("Nom du compositeur (<Enter> pour terminer) : ")
    if nom =='':
        break
    tit = raw_input("Titre de l'oeuvre : ")
    dur = raw_input("durée (minutes) : ")
    int = raw_input("interprète principal : ")
    requete ="insert into oeuvres(comp, titre, duree, interpr) values \
                ('%s', '%s', %s, '%s')" % (nom, tit, dur, int)
    cur.execute(requete)
# Affichage des données entrées, pour vérification :
cur.execute("select * from oeuvres")
print cur.pp()

connex.commit()
```

Exercice 18.2 :

```
#####################################
# Bombardement d'une cible mobile   #
# (C) G. Swinnen - Avril 2004 - GPL #
```

```
####################################

from Tkinter import *
from math import sin, cos, pi
from random import randrange
from threading import Thread

class Canon:
    """Petit canon graphique"""
    def __init__(self, boss, num, x, y, sens):
        self.boss = boss            # référence du canevas
        self.num = num              # n° du canon dans la liste
        self.x1, self.y1 = x, y     # axe de rotation du canon
        self.sens = sens            # sens de tir (-1:gauche, +1:droite)
        self.lbu = 30               # longueur de la buse
        # dessiner la buse du canon (horizontale) :
        self.x2, self.y2 = x + self.lbu * sens, y
        self.buse = boss.create_line(self.x1, self.y1,
                                     self.x2, self.y2, width =10)
        # dessiner le corps du canon (cercle de couleur) :
        self.rc = 15                # rayon du cercle
        self.corps = boss.create_oval(x -self.rc, y -self.rc, x +self.rc,
                                      y +self.rc, fill ='black')
        # pré-dessiner un obus (au départ c'est un simple point) :
        self.obus = boss.create_oval(x, y, x, y, fill='red')
        self.anim = 0
        # retrouver la largeur et la hauteur du canevas :
        self.xMax = int(boss.cget('width'))
        self.yMax = int(boss.cget('height'))

    def orienter(self, angle):
        "régler la hausse du canon"
        # rem : le paramètre <angle> est reçu en tant que chaîne.
        # il faut donc le traduire en réel, puis le convertir en radians :
        self.angle = float(angle)*2*pi/360
        self.x2 = self.x1 + self.lbu * cos(self.angle) * self.sens
        self.y2 = self.y1 - self.lbu * sin(self.angle)
        self.boss.coords(self.buse, self.x1, self.y1, self.x2, self.y2)

    def feu(self):
        "déclencher le tir d'un obus"
        # référence de l'objet cible :
        self.cible = self.boss.master.cible
        if self.anim ==0:
            self.anim =1
            # position de départ de l'obus (c'est la bouche du canon) :
            self.xo, self.yo = self.x2, self.y2
            v = 20              # vitesse initiale
            # composantes verticale et horizontale de cette vitesse :
            self.vy = -v *sin(self.angle)
            self.vx = v *cos(self.angle) *self.sens
            self.animer_obus()

    def animer_obus(self):
        "animer l'obus (trajectoire balistique)"
        # positionner l'obus, en re-définissant ses coordonnées :
        self.boss.coords(self.obus, self.xo -3, self.yo -3,
                                    self.xo +3, self.yo +3)
        if self.anim >0:
            # calculer la position suivante :
            self.xo += self.vx
            self.yo += self.vy
            self.vy += .5
            self.test_obstacle()        # a-t-on atteint un obstacle ?
            self.boss.after(1, self.animer_obus)
        else:
            # fin de l'animation :
            self.boss.coords(self.obus, self.x1, self.y1, self.x1, self.y1)

    def test_obstacle(self):
```

```
        "évaluer si l'obus a atteint une cible ou les limites du jeu"
        if self.yo >self.yMax or self.xo <0 or self.xo >self.xMax:
            self.anim =0
            return
        if self.yo > self.cible.y -3 and self.yo < self.cible.y +18 \
        and self.xo > self.cible.x -3 and self.xo < self.cible.x +43:
            # dessiner l'explosion de l'obus (cercle orange) :
            self.explo = self.boss.create_oval(self.xo -10,
                         self.yo -10, self.xo +10, self.yo +10,
                         fill ='orange', width =0)
            self.boss.after(150, self.fin_explosion)
            self.anim =0

    def fin_explosion(self):
        "effacer le cercle d'explosion - gérer le score"
        self.boss.delete(self.explo)
        # signaler le succès à la fenêtre maîtresse :
        self.boss.master.goal()

class Pupitre(Frame):
    """Pupitre de pointage associé à un canon"""
    def __init__(self, boss, canon):
        Frame.__init__(self, bd =3, relief =GROOVE)
        self.score =0
        s =Scale(self, from_ =88, to =65,
                 troughcolor ='dark grey',
                 command =canon.orienter)
        s.set(45)                       # angle initial de tir
        s.pack(side =LEFT)
        Label(self, text ='Hausse').pack(side =TOP, anchor =W, pady =5)
        Button(self, text ='Feu !', command =canon.feu).\
                                    pack(side =BOTTOM, padx =5, pady =5)
        Label(self, text ="points").pack()
        self.points =Label(self, text=' 0 ', bg ='white')
        self.points.pack()
        # positionner à gauche ou à droite suivant le sens du canon :
        gd =(LEFT, RIGHT)[canon.sens == -1]
        self.pack(padx =3, pady =5, side =gd)

    def attribuerPoint(self, p):
        "incrémenter ou décrémenter le score"
        self.score += p
        self.points.config(text = ' %s ' % self.score)

class Cible:
    """objet graphique servant de cible"""
    def __init__(self, can, x, y):
        self.can = can              # référence du canevas
        self.x, self.y = x, y
        self.cible = can.create_oval(x, y, x+40, y+15, fill ='purple')

    def deplacer(self, dx, dy):
        "effectuer avec la cible un déplacement dx,dy"
        self.can.move(self.cible, dx, dy)
        self.x += dx
        self.y += dy
        return self.x, self.y

class Thread_cible(Thread):
    """objet thread gérant l'animation de la cible"""
    def __init__(self, app, cible):
        Thread.__init__(self)
        self.cible = cible          # objet à déplacer
        self.app = app              # réf. de la fenêtre d'application
        self.sx, self.sy = 6, 3     # incréments d'espace et de
        self.dt =300                # temps pour l'animation (ms)

    def run(self):
        "animation, tant que la fenêtre d'application existe"
        x, y = self.cible.deplacer(self.sx, self.sy)
```

```
        if x > self.app.xm -50 or x < self.app.xm /5:
                self.sx = -self.sx
        if y < self.app.ym /2 or y > self.app.ym -20:
                self.sy = -self.sy
        if self.app != None:
            self.app.after(int(self.dt), self.run)

    def stop(self):
        "fermer le thread si la fenêtre d'application est refermée"
        self.app =None

    def accelere(self):
        "accélérer le mouvement"
        self.dt /= 1.5

class Application(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.master.title('<<< Tir sur cible mobile >>>')
        self.pack()
        self.xm, self.ym = 600, 500
        self.jeu = Canvas(self, width =self.xm, height =self.ym,
                          bg ='ivory', bd =3, relief =SUNKEN)
        self.jeu.pack(padx =4, pady =4, side =TOP)

        # Instanciation d'un canon et d'un pupitre de pointage :
        x, y = 30, self.ym -20
        self.gun =Canon(self.jeu, 1, x, y, 1)
        self.pup =Pupitre(self, self.gun)

        # instanciation de la cible mobile :
        self.cible = Cible(self.jeu, self.xm/2, self.ym -25)
        # animation de la cible mobile, sur son propre thread :
        self.tc = Thread_cible(self, self.cible)
        self.tc.start()
        # arrêter tous les threads lorsque l'on ferme la fenêtre :
        self.bind('<Destroy>',self.fermer_threads)

    def goal(self):
        "la cible a été touchée"
        self.pup.attribuerPoint(1)
        self.tc.accelere()

    def fermer_threads(self, evt):
        "arrêter le thread d'animation de la cible"
        self.tc.stop()

if __name__ =='__main__':
    Application().mainloop()
```

Annexes extraites de « How to think like a computer scientist »

Suivant les termes de la GNU Free Documentation licence (voir p. 365), les annexes qui suivent doivent obligatoirement accompagner telles quelles toute distribution du texte original, que celui-ci ait été modifié (traduit, par exemple) ou non.

Contributor list

by Jeffrey Elkner

Perhaps the most exciting thing about a free content textbook is the possibility it creates for those using the book to collaborate in its development. I have been delighted by the many responses, suggestions, corrections, and words of encouragement I have received from people who have found this book to be useful, and who have taken the time to let me know about it.

Unfortunately, as a busy high school teacher who is working on this project in my spare time (what little there is of it ;-), I have been neglectful in giving credit to those who have helped with the book. I always planned to add an "Acknowlegdements" sections upon completion of the first stable version of the book, but as time went on it became increasingly difficult to even track those who had contributed.

Upon seeing the most recent version of Tony Kuphaldt's wonderful free text, "Lessons in Electric Circuits", I got the idea from him to create an ongoing "Contributor List" page which could be easily modified to include contributors as they come in.

My only regret is that many earlier contributors might be left out. I will begin as soon as possible to go back through old emails to search out the many wonderful folks who have helped me in this endeavour. In the mean time, if you find yourself missing from this list, please accept my humble apologies and drop me an email at jeff@elkner.net to let me know about my oversight.

And so, without further delay, here is a listing of the contributors:

Lloyd Hugh Allen

Lloyd sent in a correction to section 8.4. He can be reached at: lha2@columbia.edu

Yvon Boulianne

Yvon sent in a correction of a logical error in Chapter 5. She can be reached at: mystic@monuniverse.net

Fred Bremmer

Fred submitted a correction in section 2.1. He can be reached at: Fred.Bremmer@ubc.cu

Jonah Cohen

Jonah wrote the Perl scripts to convert the LaTeX source for this book into beautiful HTML. His Web page is jonah.ticalc.org and his email is JonahCohen@aol.com

Michael Conlon

Michael sent in a grammer correction in Chapter 2 and an improvement in style in Chapter 1, and he initiated discussion on the technical aspects of interpreters. Michael can be reached at: michael.conlon@sru.edu

Courtney Gleason

Courtney and Katherine Smith created the first version of horsebet.py, which is used as the case study for the last chapters of the book. Courtney can be reached at: orion1558@aol.com

Lee Harr

Lee submitted corrections for sections 10.1 and 11.5. He can be reached at: missive@linuxfreemail.com

James Kaylin

James is a student using the text. He has submitted numerous corrections. James can be reached by email at: Jamarf@aol.com

David Kershaw

David fixed the broken catTwice function in section 3.10. He can be reached at: david_kershaw@merck.com

Eddie Lam

Eddie has sent in numerous corrections to Chapters 1, 2, and 3. He also fixed the Makefile so that it creates an index the first time it is run and helped us set up a versioning scheme. Eddie can be reached at: nautilus@yoyo.cc.monash.edu.au

Man-Yong Lee

Man-Yong sent in a correction to the example code in section 2.4. He can be reaced at: yong@linuxkorea.co.kr

David Mayo

While he didn't mean to hit us over the head with it, David Mayo pointed out that the word "unconsciously" in chapter 1 needed to be changed to "subconsciously". David can be reached at: bdbear44@netscape.net

Chris McAloon

Chris sent in several corrections to sections 3.9 and 3.10. He can be reached at: cmcaloon@ou.edu

Matthew J. Moelter

Matthew has been a long-time contributor who sent in numerous corrections and suggestions to the book. He can be reached at: mmoelter@calpoly.edu

Simon Dicon Montford

Simon reported a missing function definition and several typos in Chapter 3. He also found errors in the increment function in Chapter 13. He can be reached at: dicon@bigfoot.com

John Ouzts

John sent in a correction to the "return value" definition in Chapter 3. He can be reached at: jouzts@bigfoot.com

Kevin Parks

Kevin sent in valuable comments and suggestions as to how to improve the distribution of the book. He can be reached at: cpsoct@lycos.com

David Pool

David sent in a typo in the glossary of chapter 1, as well as kind words of encouragement. He can be reached at: pooldavid@hotmail.com

Michael Schmitt

Michael sent in a correction to the chapter on files and exceptions. He can be reached at: ipv6_128@yahoo.com

Paul Sleigh

Paul found an error in Chapter 7 and a bug in Jonah Cohen's Perl script that generates HTML from LaTeX. He can be reached at: bat@atdot.dotat.org

Christopher Smith

Chris is a computer science teacher at the Blake School in Minnesota who teaches Python to his beginning students. He can be reached at: csmith@blakeschool.org or smiles@saysomething.com

Katherine Smith

Katherine and Courtney Gleason created the first version of horsebet.py, which is used as the case study for the last chapters of the book. Katherine can be reached at: kss_0326@yahoo.com

Craig T. Snydal

Craig is testing the text in a course at Drew University. He has contributed several valuable suggestions and corrections, and can be reached at: csnydal@drew.edu

Ian Thomas

Ian and his students are using the text in a programming course. They are the first ones to test the chapters in the latter half of the book, and they have make numerous corrections and suggestions. Ian can be reached at: ithomas@sd70.bc.ca

Keith Verheyden

Keith made correction in Section 3.11 and can be reached at: kverheyd@glam.ac.uk

Chris Wrobel

Chris made corrections to the code in the chapter on file I/O and exceptions. He can be reached at: ferz980@yahoo.com

Moshe Zadka

Moshe has made invaluable contributions to this project. In addition to writing the first draft of the chapter on Dictionaries, he provided continual guidance in the early stages of the book. He can be reached at: moshez@math.huji.ac.il

Preface

by J. Elkner

This book owes its existance to the collaboration made possible by the internet and the free software movement. Its three authors, a college professor, a high school teacher, and a professional programmer, have yet to meet face to face, but we have been able to work closely together, and have been aided by many wonderful folks who have donated their time and energy to helping make it better.

What excites me most about it is that it is a testament to both the benefits and future possibilities of this kind of collaboration, the framework for which has been put in place by Richard Stallman and the Free Software Foundation.

How and why I came to use Python

In 1999, the College Board's Advanced Placement Computer Science exam was given in C++ for the first time. As in many high schools throughout the country, the decision to change languages had a direct impact on the computer science curriculum where I teach at Yorktown High School, in Arlington, Virginia. Up to this point, Pascal was the language of instruction in both our first year and AP courses. In keeping with past practice of giving students two years of exposure to the same language, we made the decision to switch to C++ in the first year course for the 1997-98 school year, so that we would be in step with the College Board's change for the AP course the following year.

Two years later, I was convinced that C++ was a poor choice to use for introducing students to computer science. While it is certainly a very powerful programming language, it is also an extremely difficult language to learn and teach. I found myself constantly fighting with C++'s difficult syntax and multiple ways of doing things, and I was losing many students unnecessarily as a result. Convinced there had to be a better language choice for our first year class, I went looking for an alternative to C++.

A discussion on the High School Linux Users' Group mailing list provided a solution. A thread emerged during the latter part of January, 1999 concerning the best programming language for use with first time high school computer science students. In a posting on January 30th, Brendon Ranking wrote:

> I believe that Python is the best choice for any entry-level programming class. It teaches proper programming principles while being incredibly easy to learn. It is also designed to be object oriented from its inception so it doesn't have the add-on pain that both Perl and C++ suffer from..... It is also *very* widely supported and very much web-centric, as well.

I had first heard of Python a few years earlier at a Linux Install Fest, when an enthusiastic Michael McLay told me about Python's many merits. He and Brendon had now convinced me that I needed to look into Python.

Matt Ahrens, one of Yorktown's gifted students, jumped at the chance to try out Python, and in the final two months of the 1998-99 school year he not only learned the language but wrote an application called pyTicket which enabled our staff to report technology problems via the web. I knew that Matt could not have finished an application of that scale in so short a time in C++, and this accomplishment combined with Matt's positive assessment of Python suggested Python was the solution I was looking for.

Finding a text book

Having decided to use Python in both my introductory computer science classes the following year, the most pressing problem was the lack of an available text book.

Free content came to the rescue. Earlier in the year Richard Stallman had introduced me to Allen Downey. Both of us had written to Richard expressing an interest in developing free educational content. Allen had already written a first year computer science text book titled, How to think like a computer scientist. When I read this book I knew immediately that I wanted to use it in my class. It was the clearest and most helpful computer science text I had seen. It emphasized the processes of thought involved in programming, rather than the features of a particular language. Reading it immediately made me a better teacher.

Not only was How to think like a computer scientist an excellent book, but it was also released under a GNU public license, which meant it could be used freely and modified to meet the needs of its user. Once I decided to use Python, it occurred to me that I could translate Allen's original Java version into the new language. While I would not have been able to write a text book on my own, having Allen's book to work from made it possible for me to do so, at the same time demonstrating that the cooperative development model used so well in software could also work for educational content.

Working on this book for the last two years has been rewarding for both me and my students, and the students played a big part in the process. Since I could make instant changes whenever someone found a spelling error or difficult passage, I encouraged them to look for errors in the book by giving them a bonus point every time they found or suggested something that resulted in a change in the text. This had the double benefit of encouraging them to read the text more carefully, and of getting the text thoroughly reviewed by its most important critics, students using it to learn computer science.

For the second half of the book on object oriented programming, I knew that someone with more real programming experience than I had would be needed to do it right. The book actually sat in an unfinished state for the better part of a year until two things happened that led to its completion.

I received an email from Chris Meyers expressing interest in the book. Chris is a professional programmer who started teaching a programming course last year using Python at Lane Community College in Eugene Oregon. The prospect of teaching the course had led Chris to the book, and he started helping out with it immediately. By the end of the school year he had created a companion project on our web site at http://www.ibiblio.org/obp called Python for Fun and was working with some of my most advanced students as a master teacher, guiding them beyond the places I could take them.

Introducing programming with Python

The process of translating and using How to think like a computer scientist for the past two years has confirmed Python's suitability to teaching beginning students. Python greatly simplifies programming examples and makes important programming ideas easier to teach.

The first example from the text dramatically illustrates this point. It is the traditional "hello, world" program, which in the C++ version of the book looks like this:

```
#include <iostream.h>
void main()
{
      cout << "Hello, world." << endl;
}
```

in the Python version it becomes:

```
print "Hello, World!"
```

Even though this is a trivial example, the advantages to Python stand out. There are no prerequisites to Yorktown's Computer Science I course, so many of the students seeing this example are looking at their first program. Some of them are undoubtedly a little nervous, having heard that computer programming is difficult to learn. The C++ version has always forced me to choose between two unsatisfying options: either to explain the #include, void main(), {, and } statements, and risk confusing or intimidating some of the students right at the start, or to tell them "just don't worry about all of that stuff now, we will talk about it later" and risk the same thing. The educational objectives at this point in the course are to introduce students to the idea of a programming statement and to get them to make their first program, thereby introducing them to the programming environment. The Python program has exactly what is needed to do these things, and nothing more.

Comparing Section 1.5 of each version of the book, where this first program is located, further illustrates what this means to the beginning student. There are thirteen paragraphs of explanation of "Hello, world" in the C++ version, in the Python version there are only two. More importantly, the missing eleven paragraphs do not deal with the "big ideas" in computer programming, but with the minutia of C++ syntax. I found this same thing happening throughout the book. Whole paragraphs simply disappear from the Python version of the text because Python's much clearer syntax renders them unnecessary.

Using a very high level language like Python allows a teacher to postpone talking about low level details of the machine until students have the background that they need to better make sense of the details. It thus creates the ability to put "first things first" pedagogically.

One of the best examples of this is the way in which Python handles variables. In C++ a variable is a name for a place which holds a thing. Variables have to be declared with types at least in part because the size of the place to which they refer needs to be predetermined. Thus the idea of a variable is bound up with the hardware of the machine. The powerful and fundamental concept of a variable is already difficult enough for beginning students (in both Computer Science and Algebra). Bytes and addresses do not help the matter. In Python a variable is a name which refers to a thing. This is a far more intuitive concept for beginning students, and one which is much closer to the meaning of variable that they learned in their math class. I had much less difficulty teaching variables this year than I did in the past, and I spent less time helping students with problems using them.

Another example of how Python aides in the teaching and learning of programming is in its syntax for functions. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the key word def, so I simply tell my students, "when you define a function, begin with def, followed by the name of the function that you are defining, when you call a function, simply call (type) out its name." Parameters go with definitions, arguments go with calls. There are no return types or parameter types or reference and value parameters to get in the way, so I am now able to teach functions in less then half the time that it previously took me, with better comprehension.

Using Python has improved the effectiveness of our computer science program for all students. I see a higher general level of success and a lower level of frustration than I experienced during the two years using C++. I move faster with better results. More students leave the course with the ability to create meaningful programs, and with the positive attitude toward the experience of programming that this engenders.

Building a community

I have received email every continent on the globe and from as far away as Korea from people using this book to learn or to teach programming. A user community has begun to emerge and increasing numbers of people have been contributing to the project by sending in materials for the companion web site at http://www.ibiblio.org/obp.

With the publication of the book in print form, I expect the growth in the user community to continue and accelerate. It is the emergence of this user community and the possibility it suggests for similar collaboration among educators that has been the most exciting thing for me about working on the project. By working together we can both increase the quality of materials available for our use and save valuable time. I invite you to join our community and look forward to hearing from you.

Jeffrey Elkner
Yorktown High School
Arlington, Virginia

GNU Free Documentation License

Version 1.1, March 2000
Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document," below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical, or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, \LaTeX~input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History," and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications," preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements," provided it contains nothing but endorsements of your Modified Version by various parties---for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements," and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate," and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of Section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See

http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Index

A

after (méthode).....109, 309
alias.....145
animation.....106, 108, 309
annexes.....315
Apache.....279
appartenance à une séquence.....132
applications web.....273
arguments.....69, 82

B

barres d'outils avec bulles d'aide (exemple).....221
bases de données.....257
  client pour MySQL.....264
  dictionnaire d'application.....265
  la requête select.....263
  le modèle client/serveur.....258
  mise en oeuvre avec Gadfly.....259
  recherches dans une base de données.....261
blocs d'instructions.....33
boucle.....38, 116, 131
boutons radio (widget).....207
break (instruction).....116
bytecode.....15

C

cahier des charges.....183, 253
calculatrice.....22, 96
canevas.....91
caractères accentués.....43, 52, 215
CGI.....274
chaînes de caractères.....123
  classement des caractères.....134
  concaténation, répétition.....130
  conversion de type.....137
  encodage Utf-8.....125
  encodage/décodage.....126
  extraction de fragments.....129
  fonctions intégrées.....137
  formatage.....138
  Indiçage.....52, 123
  méthodes spécifiques.....135
  opérations élémentaires.....54
  première approche.....50
  séquences non modifiables.....133
  types string et unicode.....124
classe Canon (exemple).....238
classes.....159
  attributs d'instance.....162
  bibliothèques de classes.....180
  classes et interfaces graphiques.....183
  définition d'une classe.....160
  échange d'informations.....187
  espaces de noms.....173
  héritage.....167
  méthode constructeur.....169
  méthodes.....167
  objets comme valeurs de retour.....165
  objets composés d'objets.....164
  polymorphisme.....175
  similitude et unicité.....163
  structure d'une classe (résumé).....178
  utilité des classes.....159
client réseau multi-tâches.....292
client réseau rudimentaire.....290
code des couleurs (exemple).....183
Combo Box (widget).....214
commentaires.....42, 253
communications à travers un réseau.....287
compilation et interprétation.....15
composition d'instructions Tkinter.....104
composition des instructions.....29
conditionnelle (exécution).....31
contrôle du flux d'exécution.....31, 156, 233
curseurs (exemple).....195

D

débogage.....16
debug.....16
définir une fonction.....67
définir une méthode.....167
déplacer des dessins (exemple).....211
dérivation (des classes).....159
dessins alternés (exemple).....94
détection d'un clic.....98
développement incrémental.....225, 235
dictionnaires.....150
  accès aléatoire.....154
  clés particulières.....153
  contrôle du flux d'exécution via un dictionnaire. .156
  création.....151
  histogrammes.....155
  méthodes spécifiques.....152
  opérations sur les dictionnaires.....151
  parcours d'un dictionnaire.....153
division entière.....22

E

encapsulation.....159, 184
encodage par défaut.....44

entier (type de donnée)....47
erreurs à l'exécution....17
erreurs de syntaxe....16
erreurs sémantiques....17
étiquettes (pour arguments)....82
événement....89, 97, 199
exceptions....119
exécution conditionnelle....31
expérimentation....17, 192
expressions....27

F

fausseté....63
fenêtres....85
fenêtres avec menus (exemple)....225
Fibonacci (suite de)....40
fichiers....111
  écriture et lecture séquentielles....114
  enregistrement de variables....118
  fichiers texte....117
  noms de fichiers....112
  répertoire courant....112
  utilité des fichiers....111
float (type de donnée)....49
flux d'exécution (contrôle du)....31
fonctions originales....**67**
fonctions prédéfinies....**59**
for (instruction)....131, 144, 210
formes d'importation....113
formulaire HTML....276
Frame (widget)....209

G

Gadfly....258
graphique (interface)....85
grid (méthode)....101

H

héritage....174, 187

I

if (instruction)....32
importer....60, 77, 113
in (instruction)....131, 132
input (fonction)....59
instanciation....86, 159
instructions composées....33
instructions imbriquées....34
instructions répétitives....37
integer (type de donnée)....47
interface graphique....85, 183
interprétation (et compilation)....15

J

jeu de Ping....252
jeu des bombardes....235
jeu des bombardes, version réseau....296

K

Karrigell....279

L

lambda (expression)....221
langages formels....18
Latin-1....43, 128, 215
lignes colorées (exemple)....91
listes....140
  concaténation, multiplication....144
  contrôle du flux d'exécution via une liste....233
  copie d'une liste....145
  création, avec range()....143
  définition....140
  insertion d'éléments....142
  méthodes spécifiques....141
  modification....140
  modification (slicing)....142
  parcours, à l'aide de for....144
  première approche....55
  suppression, remplacement d'éléments....143
  test d'appartenance....145
long (type de donnée)....47

M

mainloop....88
métaprogrammation....223
mode interactif....21
module de fonctions....60
module turtle....62, 76
modules....180
modules de fonctions....76
multi-tâches....291

N

nombres aléatoires....146, 148

O

opérateurs....22, 27
opérateurs de comparaison....33
oscilloGraphe (exemple)....190

P

pack (méthode)....88, 95, 101
paramètres....68, 69, 70, 81
parcours d'une séquence....131
petit train (exemple)....187
pickle (module)....119

polymorphisme....160
positionnement d'un clic....98
PostgreSQL....258
priorité des opérations....27
procédures....73
programmes concrets....235
protocole de communication....297
prototypage....238
Python Mega Widgets....213

R

range (fonction)....144
raw_input (fonction)....59
récursivité....108
réel (type de donnée)....49
réseau....287
réservés (mots)....23

S

Scale (widget)....195
script....41, 75
script CGI....276
Scrolled Canvas (widget)....219
Scrolled Text (widget)....216
séquence d'instructions....31
serveur réseau élémentaire....288
serveur réseau multi-tâches....294
serveur web....278
sessions (serveur web)....283
sockets....287
solutions aux exercices....317
SQL....258
string (type)....50, 124, 134
structures de données....123

T

threads....291
threads concurrents....302
time.sleep (pour animations)....310
Tkinter....85, 100
try (instruction)....119
tuples....**149**
typage des paramètres....81
typage dynamique....144
types de données....**47**

U

Unicode (norme)....43, 124
unicode (type)....124, 128, 134, 215
Utf-8....44, 125, 128, 215

V

valeur par défaut (de paramètres)....81
variables....
  affectation....24
  affectations multiples....26
  affectations parallèles....26
  afficher la valeur....25
  assignation....24
  mots réservés....23
  noms de variables....23
  séparateur décimal....26
  typage des variables....25
  variables d'instance....162
  variables globales....71, 159, 184
  variables locales....71
virgule flottante (type)....49
vrai ou faux....63

W

web....273
while (instruction)....38, 64
widget....86, 100, 190, 199
widget composite....196

Table des matières

PRÉFACE 5

Choix d'un premier langage de programmation 6
Présentation du langage Python 7
Pour le professeur qui souhaite utiliser cet ouvrage comme support de cours 8
Versions du langage 9
Distribution de Python - Bibliographie 10
Exemples du livre 10
Remerciements 11

PENSER COMME UN PROGRAMMEUR 13

La démarche du programmeur 13
Langage machine, langage de programmation 14
Compilation et interprétation 15
Mise au point d'un programme & recherche des erreurs (« debug ») 16
Erreurs de syntaxe 16
Erreurs sémantiques 17
Erreurs à l'exécution 17
Recherche des erreurs et expérimentation 17
Langages naturels et langages formels 18

PREMIERS PAS 21

Calculer avec Python 21
Données et variables 23
Noms de variables et mots réservés 23
Affectation (ou assignation) 24
Afficher la valeur d'une variable 25
Typage des variables 25
Affectations multiples 26
Opérateurs et expressions 27
Priorité des opérations 27
Composition 29

CONTRÔLE DU FLUX D'EXÉCUTION 31

Séquence d'instructions 31
Sélection ou exécution conditionnelle 31
Opérateurs de comparaison 33
Instructions composées – Blocs d'instructions 33
Instructions imbriquées 34
Quelques règles de syntaxe Python 34
Les limites des instructions et des blocs sont définies par la mise en page 34
Instruction composée = En-tête , double point , bloc d'instructions indenté 35
Les espaces et les commentaires sont normalement ignorés 35

INSTRUCTIONS RÉPÉTITIVES. 37

Ré-affectation 37
Répétitions en boucle - l'instruction while 38
Commentaires 38
Remarques 39
Élaboration de tables 39
Construction d'une suite mathématique 39

Premiers scripts, ou comment conserver nos programmes **41**
Remarque concernant les caractères accentués **43**

PRINCIPAUX TYPES DE DONNÉES ***47***
Les données numériques **47**
Les types « integer » et « long » 47
Le type « float » **49**
Les données alphanumériques **50**
Le type « string » 50
Accès aux caractères individuels d'une chaîne 52
Limitations du type « string » 52
Opérations élémentaires sur les chaînes 54
Les listes (première approche) **55**

FONCTIONS PRÉDÉFINIES ***59***
Interaction avec l'utilisateur : la fonction input() **59**
Remarques importantes : 59
Importer un module de fonctions **60**
Un peu de détente avec le module turtle **62**
Véracité/fausseté d'une expression **63**
Révision **63**
Contrôle du flux - Utilisation d'une liste simple 63
Boucle while - Instructions imbriquées 64

FONCTIONS ORIGINALES ***67***
Définir une fonction **67**
Fonction simple sans paramètres 68
Fonction avec paramètre 69
Utilisation d'une variable comme argument 70
Fonction avec plusieurs paramètres 70
Variables locales, variables globales **71**
« Vraies » fonctions et procédures **73**
Utilisation des fonctions dans un script **75**
Modules de fonctions **76**
Typage des paramètres **81**
Valeurs par défaut pour les paramètres **81**
Arguments avec étiquettes **82**

UTILISATION DE FENÊTRES ET DE GRAPHISMES ***85***
Interfaces graphiques (GUI) **85**
Premiers pas avec Tkinter **85**
Programmes pilotés par des événements **89**
Exemple graphique : tracé de lignes dans un canevas **91**
Exemple graphique : deux dessins alternés 94
Exemple graphique : calculatrice minimaliste 96
Exemple graphique : détection et positionnement d'un clic de souris 98
Les classes de widgets Tkinter **100**
Utilisation de la méthode grid() pour contrôler la disposition des widgets **101**
Composition d'instructions pour écrire un code plus compact **104**
Modification des propriétés d'un objet - Animation **106**
Animation automatique - Récursivité **108**

LES FICHIERS *111*
Utilité des fichiers **111**
Travailler avec des fichiers **112**
Noms de fichiers - Répertoire courant **112**
Les deux formes d'importation **113**
Écriture séquentielle dans un fichier **114**
Lecture séquentielle d'un fichier **115**
L'instruction break pour sortir d'une boucle **116**
Fichiers texte **117**
Enregistrement et restitution de variables diverses **118**
Gestion des exceptions.
Les instructions try – except - else **119**

APPROFONDIR LES STRUCTURES DE DONNÉES *123*
Le point sur les chaînes de caractères **123**
Indiçage, extraction, longueur 123
Les types « string » et « unicode » 124
L'encodage Utf-8 125
Conversion (encodage/décodage) des chaînes 126
Conversion d'une chaîne string en chaîne unicode 126
Conversion d'une chaîne unicode en chaîne string 127
Conversion d'une chaîne Utf-8 en Latin-1, ou vice-versa 128
Quand faut-il convertir ? 128
Extraction de fragments de chaînes 129
Concaténation, Répétition 130
Parcours d'une séquence. L'instruction for ... in 131
Appartenance d'un élément à une séquence. L'instruction in utilisée seule 132
Les chaînes sont des séquences non modifiables 133
Les chaînes sont comparables 133
Classement des caractères 134
Les chaînes sont des objets 135
Fonctions intégrées 137
Formatage des chaînes de caractères 138
Le point sur les listes **140**
Définition d'une liste – Accès à ses éléments 140
Les listes sont modifiables 140
Les listes sont des objets 141
Techniques de « slicing » avancé pour modifier une liste 142
Insertion d'un ou plusieurs éléments n'importe où dans une liste 142
Suppression / remplacement d'éléments 143
Création d'une liste de nombres à l'aide de la fonction range() 143
Parcours d'une liste à l'aide de for, range() et len() 144
Une conséquence du typage dynamique 144
Opérations sur les listes 144
Test d'appartenance 145
Copie d'une liste 145
Petite remarque concernant la syntaxe : 146
Nombres aléatoires - Histogrammes 146
Tirage au hasard de nombres entiers 148
Les tuples **149**
Les dictionnaires **150**
Création d'un dictionnaire 151

Opérations sur les dictionnaires ... 151
Les dictionnaires sont des objets ... 152
Parcours d'un dictionnaire ... 153
Les clés ne sont pas nécessairement des chaînes de caractères ... 153
Les dictionnaires ne sont pas des séquences ... 154
Construction d'un histogramme à l'aide d'un dictionnaire ... 155
Contrôle du flux d'exécution à l'aide d'un dictionnaire ... 156

CLASSES, OBJETS, ATTRIBUTS ... 159
Utilité des classes ... 159
Définition d'une classe élémentaire ... 160
Attributs (ou variables) d'instance ... 162
Passage d'objets comme arguments lors de l'appel d'une fonction ... 162
Similitude et unicité ... 163
Objets composés d'objets ... 164
Objets comme valeurs de retour d'une fonction ... 165
Modification des objets ... 165

CLASSES, MÉTHODES, HÉRITAGE ... 167
Définition d'une méthode ... 167
Définition concrète d'une méthode dans un script ... 168
Essai de la méthode, dans une instance quelconque ... 169
La méthode « constructeur » ... 169
Espaces de noms des classes et instances ... 173
Héritage ... 174
Héritage et polymorphisme ... 175
Modules contenant des bibliothèques de classes ... 180

CLASSES & INTERFACES GRAPHIQUES ... 183
« Code des couleurs » : un petit projet bien encapsulé ... 183
Cahier des charges de notre programme : ... 183
Mise en œuvre concrète ... 184
Commentaires : ... 185
« Petit train » : héritage, échange d'informations entre classes ... 187
Cahier des charges : ... 187
Implémentation : ... 188
Commentaires : ... 188
« OscilloGraphe » : un widget personnalisé ... 190
Expérimentation : ... 192
Cahier des charges : ... 193
Implémentation : ... 193
« Curseurs » : un widget composite ... 195
Présentation du widget « Scale » ... 195
Construction d'un panneau de contrôle à trois curseurs ... 196
Commentaires : ... 197
Propagation des événements ... 199
Intégration de widgets composites dans une application synthèse ... 199
Commentaires : ... 201

ET POUR QUELQUES WIDGETS DE PLUS ... 207
Les « boutons radio » ... 207
Commentaires : ... 208
Utilisation de cadres pour la composition d'une fenêtre ... 209

Commentaires :.....210
Comment déplacer des dessins à l'aide de la souris 211
Commentaires :.....212
Python Mega Widgets 213
« Combo Box » 214
Commentaires : 215
Remarque concernant l'entrée de caractères accentués.....215
« Scrolled Text » 216
Commentaires :.....217
« Scrolled Canvas » 219
Commentaires :.....220
Barres d'outils avec bulles d'aide - expressions lambda 221
Métaprogrammation. Expressions lambda : 223
Fenêtres avec menus 225
Cahier des charges de l'exercice :.....225
Première ébauche du programme :.....226
Analyse du script :.....227
Ajout de la rubrique « Musiciens » 228
Analyse du script.....229
Ajout de la rubrique « Peintres » : 229
Analyse du script :.....230
Ajout de la rubrique « Options » : 230
Menu avec « cases à cocher ».....231
Menu avec choix exclusifs 232
Contrôle du flux d'exécution à l'aide d'une liste.....233
Pré-sélection d'une rubrique 234

ANALYSE DE PROGRAMMES CONCRETS 235
Jeu des bombardes 235
Prototypage d'une classe « Canon » 238
Commentaires :.....239
Ajout de méthodes au prototype 240
Commentaires :.....241
Développement de l'application 242
Commentaires :.....246
Développements complémentaires 247
Commentaires :.....251
Jeu de Ping 252
Principe 252
Programmation 253
Cahier des charges du logiciel à développer.....253

GESTION D'UNE BASE DE DONNÉES 257
Les bases de données 257
SGBDR - Le modèle client/serveur 258
Le langage SQL - Gadfly 258
Mise en œuvre d'une base de données simple avec Gadfly 259
Création de la base de données 259
Connexion à une base de données existante 260
Recherches dans une base de données
..... 261
La requête select 263
Ébauche d'un logiciel client pour MySQL 264
Décrire la base de données dans un dictionnaire d'application 265
Définir une classe d'objets-interfaces 267
Commentaires :.....268

Construire un générateur de formulaires ... 269
Commentaires : ... 270
Le corps de l'application ... 270
Commentaires : ... 271

APPLICATIONS WEB ... 273
Pages web interactives ... 273
L'interface CGI ... 274
Une interaction CGI rudimentaire ... 274
Un formulaire HTML pour l'acquisition des données ... 276
Un script CGI pour le traitement des données ... 276
Un serveur web en pur Python ! ... 278
Installation de Karrigell ... 279
Démarrage du serveur : ... 279
Ébauche de site web ... 280
Prise en charge des sessions ... 283
Exemple de mise en œuvre : ... 284
Autres développements ... 286

COMMUNICATIONS À TRAVERS UN RÉSEAU ... 287
Les sockets ... 287
Construction d'un serveur élémentaire ... 288
Commentaires : ... 289
Construction d'un client rudimentaire ... 290
Commentaires : ... 290
Gestion de plusieurs tâches en parallèle à l'aide des threads ... 291
Client gérant l'émission et la réception simultanées ... 292
Commentaires : ... 293
Serveur gérant les connexions de plusieurs clients en parallèle ... 294
Commentaires : ... 295
Jeu des bombardes, version réseau ... 296
Programme serveur : vue d'ensemble ... 297
Protocole de communication ... 297
Remarques complémentaires : ... 299
Programme serveur : première partie ... 299
Synchronisation de threads concurrents à l'aide de « verrous » (thread locks) ... 302
Utilisation : ... 302
Programme serveur : suite et fin ... 303
Commentaires : ... 305
Programme client ... 306
Commentaires : ... 308
Conclusions et perspectives : ... 309
Utilisation de threads pour optimiser les animations. ... 309
Temporisation des animations à l'aide de after() ... 309
Temporisation des animations à l'aide de time.sleep() ... 310
Exemple concret ... 311
Commentaires : ... 312

ANNEXES ... 315
Installation de Python ... 315
Sous Windows ... 315
Sous Linux ... 315
Sous MacOS ... 315
Installation des Python méga-widgets ... 315

Installation de Gadfly (système de bases de données) 315
Sous Windows : 316
Sous Linux : 316
Solutions aux exercices 317

ANNEXES EXTRAITES DE
« HOW TO THINK LIKE A COMPUTER SCIENTIST » 361
Contributor list..... 361
Preface..... 363
GNU Free Documentation License..... 365