Notes de cours pour l'apprentissage de la programmation avec Python

Gérard Swinnen
Animateur pédagogique
Institut S[t] Jean Berchmans - S[te] Marie
59, rue des Wallons - B4000 Liège

Allen B. Downey
Wellesley college
Wellesley, MA 02481 (USA)

Jeffrey Elkner
Yorktown High School
Arlington, VA 22207 (USA)

Pour une bonne part des notes qui suivent, nous nous sommes largement inspirés du tutoriel de Python écrit par Guido van Rossum lui-même (l'auteur principal de Python), ainsi que de l'excellent cours de programmation de Allen B. Downey et Jeffrey Elkner : "*How to think like a computer scientist*". (Ce cours est distribué sous licence GPL, et est disponible sur l'internet à l'adresse : http://www.ibiblio.org/obp/thinkCSpy). *Many thanks to you, Allen & Jeff !*

Introduction

Les présentes notes ont été rédigées 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 s'agit d'un texte encore provisoire, qui s'inspire très largement de plusieurs documents disponibles sur l'Internet.

Nous proposons dans ces notes une démarche d'apprentissage certainement critiquable : il n'existe malheureusement pas encore une méthode idéale pour apprendre à programmer un ordinateur, et nous avons donc été amenés à effectuer un certain nombre de choix délicats, pour lesquels nous nous sommes efforcés de respecter les principes directeurs suivants :

- L'apprentissage que nous visons doit être adapté au niveau de compréhension et aux connaissances générales d'un élève moyen. Nous nous refusons d'élaborer un cours qui soit réservé à une "élite" de petits génies.
- Dans cette option d'études et à ce niveau, l'apprentissage doit rester généraliste : il doit mettre en évidence les invariants de la programmation et de l'informatique, sans se laisser entraîner vers une spécialisation quelconque. En particulier, il doit s'efforcer de rester aussi indépendant que possible des systèmes d'exploitation et langages propriétaires.
- Les outils utilisés au cours de l'apprentissage doivent être modernes et performants, mais il faut aussi que l'élève puisse se les procurer en toute légalité à très bas prix pour son usage personnel. Toute notre démarche d'apprentissage repose en effet sur l'idée que l'élève devra très tôt mettre en chantier des réalisations personnelles qu'il pourra développer à sa guise.
- L'élève qui apprend doit pouvoir rapidement réaliser de petites applications graphiques.
  Les étudiants auxquels on s'adresse sont en effet fort jeunes (en théorie, ils sont à peine arrivés à l'âge ou l'on commence à pouvoir faire des abstractions). Dans ce cours, 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 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.

  Cette approche un peu inhabituelle leur permettra en outre de se lancer très tôt dans de petits projets personnels, par lesquels ils pourront se sentir valorisés. Nous ne voulons pas masquer la complexité des notions à acquérir, mais nous cherchons avant tout à motiver et à encourager.
  Par cette démarche, nous souhaitons également familiariser les étudiants le plus tôt possible avec le concept informatique d'objet, et ceci d'une manière très progressive. Nous voulons qu'ils puissent d'abord utiliser en abondance divers types d'objets préexistants (et notamment des objets graphiques), afin d'apprendre à exploiter efficacement les méthodes et attributs de ces objets. La construction d'objets personnalisés ne sera envisagée qu'un peu plus tard, lorsqu'une certaine familiarité avec les concepts de base aura été bien établie.

Choix d'un premier langage de programmation

Il existe un très grand nombre de langages de programmation, chacun avec ses avantages et ses inconvénients. Nous en utiliserons plusieurs, parce que nous estimons que l'enseignement secondaire au niveau duquel nous travaillons doit rester généraliste. Dans la mesure de nos moyens, nous nous efforcerons donc de présenter à l'élève un certain nombre de langages suffisamment exemplatifs de la variété existante. De toute façon, nous estimons qu'un bon programmeur doit être capable de choisir son outil de travail en fonction de la tâche à réaliser, ce qui est évidemment impossible s'il n'en connaît qu'un seul.

Mais quel langage allons-nous choisir pour commencer ?

Nous avons personnellement une assez longue expérience de la programmation sous *Visual Basic* (*Micro$oft*) et sous *Clarion* (*Top$peed*). Nous avons aussi expérimenté quelque peu sous *Delphi* (*Borl@nd*). Il était donc naturel que nous pensions d'abord à l'un ou l'autre de ces langages (avec une nette préférence pour *Clarion*).

Si l'on souhaite les utiliser comme outil de base pour un apprentissage général de la programmation, ces langages présentent toutefois deux gros inconvénients :

- Ils sont liés à des logiciels (environnements de programmation) propriétaires. Cela signifie donc, non seulement que l'institution scolaire désireuse de les utiliser devrait acheter une licence d'utilisation de ces logiciels pour chaque poste de travail (ce qui peut encore se concevoir), mais surtout que les élèves voulant utiliser leur nouvelle compétence ailleurs qu'à l'école seraient implicitement forcés d'en acquérir eux aussi des licences, ce que nous ne pouvons pas accepter.
- Ce sont des langages spécifiques du système d'exploitation *Window$*. Ils ne sont pas "portables" sur d'autres systèmes (*Un!x*, *M@c*, etc.). Cela ne cadre 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.à.d. 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 des interpréteurs et des compilateurs gratuits pour toute une série de langages dans le monde de l'*Open Source*, mais en plus ces langages sont modernes, performants, utilisables sur différents systèmes d'exploitation (*Window$*, *Linux*, *M@c* ...), et fort bien documentés.

Le langage dominant y est sans conteste le *C/C++*. Ce langage s'impose comme une référence absolue et il faudra donc que nous apprenions à l'utiliser. Il est malheureusement très rébarbatif et compliqué, trop proche de la machine. Sa syntaxe est peu lisible et fort contraignante. La mise au 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'application et autres outils d'assistance (*C++Builder*, *Kdevelop*, etc...). Si nous les utilisons trop vite, nous risquons de masquer les mécanismes de base du langage aux yeux des débutants. Nous réserverons donc l'usage du *C/C++* à l'écriture de petits morceaux de programmes ou de procédures dont la vitesse d'exécution soit primordiale. 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.

1 Un logiciel libre (*Open Source Software*) est avant tout un logiciel dont le code source est accessible à tous. 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 *LINUX*, dont la popularité ne cesse de s'accroître de jour en jour.

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, par Stéfane Fermigier.

(Le texte qui suit est extrait d'un article paru dans le magazine **Programmez!** en décembre 1998, également disponible sur http://www.linux-center.org/articles/9812/python.html)

Python est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est developpé 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*, *M$-DOS* et les différentes variantes de *Window$*. 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 (abondament 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), **reflectif** (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 librairies C existantes. On peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- La **librairie standard** de Python, et les paquetages contribués, donnent accès à une grande variété de services : chaînes de caractères et expressions régulières, services UNIX standard (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.

--- /---

Les différentes versions de Python (pour Window$, Un!x, etc.), son ***tutoriel*** original, son ***manuel de référence***, la ***documentation*** des librairies 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. Si la plupart d'entre eux n'existent encore qu'en version anglaise, on peut cependant déjà se procurer en traduction française les excellents manuels ci-après :

- ***Introduction à Python***, par Mark Lutz & David Ascher, traduction de Sébastien Tanguy, Olivier Berger & Jérôme Kalifa, Editions O'Reilly, Paris, 2000, 385 p., ISBN 2-84177-089-3
- ***L'intro Python***, par Ivan Van Laningham, traduction de Denis Frère, Karine Cottereaux et Noël Renard, Editions CampusPress, Paris, 2000, 484 p., ISBN 2-7440-0946-6
- ***Python précis & concis*** (il s'agit d'un petit aide-mémoire bien pratique), par Mark Lutz, traduction de James Guérin, Editions O'Reilly, Paris, 2000, 80 p., ISBN 2-84177-111-3

Pour aller plus loin, notamment dans l'utilisation de la librairie graphique Tkinter, ou l'exploitation maximale des ressources liées au système d'exploitation Window$, 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 ***Python Programming on Win32***, par Mark Hammond & Andy Robinson, Editions O'Reilly, Paris, 2000, 654 p., ISBN 1-56592-621-8

Il existe encore d'autres ouvrages, mais nous pensons avoir mentionné les principaux.

Remerciements

Merci à tous ceux qui oeuvrent 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 Sabine Gillman, professeur à l'Institut St. Jean-Berchmans de Liège, qui a accepté de se lancer dans l'aventure de ce nouveau cours avec ses élèves, et a également suggéré de nombreuses améliorations.

Chapitre 1 : Penser comme un programmeur

1.1 L'activité de programmation

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, assemble des composants pour réaliser des mécanismes et 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 est **la résolution de 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 grandes satisfactions.

1.2 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 programmes, c.à.d. 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, Java*...). 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.

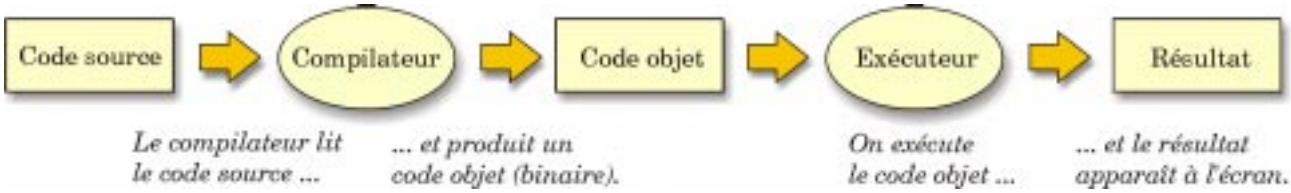
Le langage que vous allez apprendre en premier est **Python**. Il s'agit d'un langage de haut niveau, dont la traduction en codes binaires est complexe et prend donc toujours un certain temps. Cela pourrait paraître un inconvénient. En fait, les avantages que présentent les langages de haut niveau sont 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.à.d. 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.à.d. 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.

1.3 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.

- 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").



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 pseudo-code (ou *bytecode)*, lequel sera ensuite transmis à un interpréteur pour l'exécution finale. Du point de vue de l'ordinateur, le pseudo-code 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 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 pseudo-code 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 pseudo-code 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.

1.4 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 assez obscures, les erreurs de programmation s'appellent des "*bugs*" (ou "bogues", en France), et l'ensemble des techniques que l'on met en oeuvre pour les détecter et les corriger s'appelle "*debug*" (ou "déboguage").

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

1.4.1 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.

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 relancer la compilation de l'entièreté du programme après chaque modification, aussi minime soit-elle.

1.4.2 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.

1.5 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 ébauche. 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. Veuillez consulter votre professeur pour de plus amples informations et références.

1.6 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.***

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.à.d. 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.

Chapitre 2 : Premières instructions

2.1 Calculer avec Python

Python[2] présente la particularité de pouvoir être utilisé de plusieurs manières.
Nous allons d'abord l'utiliser en *mode interactif*, c.à.d. d'une manière telle que nous pourrons dialoguer avec lui directement depuis le clavier. Cela nous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, nous verrons comment créer nos premiers programmes (scripts) et les sauvegarder.

L'interpréteur peut être lancé directement depuis la ligne de commande (sous *Linux*, *M$DOS*) : il suffit d'y taper la commande **"python"** (en supposant que le logiciel ait été correctement installé). Si vous utilisez une interface graphique telle que *Window$*, *Gnome* ou *KDE*, vous préférerez vraisemblablement travailler dans une "fenêtre de terminal", ou encore dans un environnement de travail spécialisé tel que **IDLE**. Dans ce cas vous devriez disposer d'une icône de lancement pour Python sur votre bureau. Si ce n'est pas le cas, demandez à votre professeur comment procéder pour en installer une[3].

Au démarrage de l'interpréteur, vous obtenez un message d 'introduction tel que celui-ci :

```
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

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.

---

2 Pour ce guide de travail, nous nous sommes inspirés de divers textes, dont le tutoriel original de Python, écrit par Guido Van Rossum lui-même (l'auteur de Python), et dont une traduction française existe. Nous avons également puisé dans *An introduction to Tkinter* par Fredrik Lundh. Ces documents et beaucoup d'autres sont librement téléchargeables au départ du site web officiel de Python : http://www.python.org.

3 Sous *Window$*, 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. Pour tout renseignement complémentaire, veuillez consulter le site de Python.

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 :

```
>>> 20.0 / 3          # (comparez le résultat avec celui obtenu à l'exercice précédent)
>>> 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
```

2.2 Utilisation de variables

Nous reviendrons plus loin sur les différents types de données numériques. Mais avant cela, nous pouvons dès à présent aborder un concept essentiel.

La programmation d'un ordinateur - à l'aide d'un langage quelconque - 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 simple étiquette reliée à une **adresse mémoire**, c.à.d. un emplacement précis dans la mémoire vive.

A cet emplacement est stocké une **valeur** bien déterminée. Cette valeur n'est pas nécessairement une valeur numérique. Cela peut être en fait à peu près n'importe quel "objet" susceptible d'être placé dans la mémoire d'un ordinateur, comme par exemple : un nombre entier, un nombre réel, un vecteur, un nombre complexe, une chaîne de caractères typographiques, un tableau, une fonction, etc.
Toutes ces valeurs doivent évidemment être d'abord encodées sous la forme d'un certain nombre d'octets (Voir cours d'informatique générale). Étant donnée la grande variété des contenus possibles, il a fallu définir conventionnellement différents **types de variables**. Nous y reviendrons plus loin.

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***".

2.3 Noms de variables et mots réservés

Les noms de variables que vous utilisez sous Python doivent 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 (c.à.d. que 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[5]). N'utilisez 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 28 "mots réservés" ci-dessous (ils sont utilisés par le langage lui-même) :

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

2.4 Affectation (ou assignation)

Maintenant que nous savons comment définir un nom de variable, voyons comment y **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 son contenu. L'instruction d'affectation la plus commune sous Python utilise le signe "égale" :

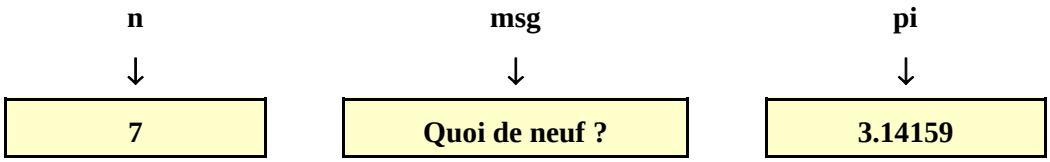
```
>>> n = 7                          # donner à n la valeur 7
>>> msg = "Quoi de neuf ?"         # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159                   # assigner sa valeur à la variable pi
```

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**.

5 Les noms commençant par une majuscule ne sont pas interdits, mais il est d'usage de les réserver aux variables qui désignent des **classes** (le concept de classe sera étudié plus loin).

Les trois instructions d'affectation ci-dessus ont eu pour effet, à la fois de créer les noms des 3 variables, de leur attribuer un type, et d'établir un lien symbolique (par un système interne de pointeurs) entre le nom de chaque variable et la valeur correspondante. On peut mieux se représenter tout cela par un petit diagramme (diagramme d'état) :



Les trois noms de variables sont mémorisés dans une zone particulière de la mémoire que l'on appelle "espace de noms". Nous aurons plus loin l'occasion de revenir sur ce concept.

2.5 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, il existe deux possibilités. La première consiste simplement à entrer au clavier le nom de la variable (suivi de <Enter>). Python répond en affichant la valeur :

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

Il s'agit là seulement d'une fonctionnalité secondaire de l'interpréteur, 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 (pour vous rappeler le type de la variable : nous y reviendrons).

2.6 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 distingue de nombreux autres langages. On dira à ce sujet que ***le typage 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 aurait fallu - par des instructions distinctes - d'abord définir le nom et le type des variables, et ensuite leur assigner un contenu compatible.

Exercice :

e 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
```

2.7 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 de **multiples affectations 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.

Attention *: les francophones que nous sommes avons l'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.à.d. le point décimal. La virgule, quant à elle, est utilisée dans Python 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.*

Exercice :

e 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 ?

2.8 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 des symboles spéciaux qui sont 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. A cette valeur est attribué automatiquement un type, lequel dépend de ce qu'il y avait 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 y 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. Exemples :

```
>>> 10 % 3
1
>>> 10 % 5
0
```

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.

2.9 Ordre 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, ces 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**ultiplication et **D**ivision ont la même priorité. Elles sont évaluées avant l'**A**ddition et la **S**oustraction, lesquelles sont donc effectuées en dernier lieu.
  Ainsi **`2*3-1 = 5`** (plutôt que 4), et **`2/3-1 = -1`** (Rappelez-vous 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).

2.10 Composition

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

Or l'une des grandes forces des langages de programmation est qu'ils permettent de construire des instructions complexes par assemblage de fragments divers. Ainsi nous savons à présent comment additionner deux nombres et comment afficher une valeur. Nous pouvons faire les deux en une seule instruction :

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

Cela n'a l'air de rien, mais cela va nous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

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

Attention cependant : il y a une limite à cette fonctionnalité. 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 : 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 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 avant, il nous faut aborder un autre concept de grande importance.

Chapitre 3 : Instructions de contrôle du flux

3.1 Suite (ou séquence[6]) d'instructions

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

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

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). Une telle instruction permet au programme de suivre différents chemins suivant les circonstances.

3.2 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 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**. Veuillez donc entrer dans votre éditeur Python les deux lignes suivantes :

```
>>> a = 4
>>> if (a > 0):
...
```

La première commande affecte la valeur 4 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[7].

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.à.d. 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

6 Une suite d'instructions est souvent désignée par le terme de ***séquence*** dans les ouvrages qui traitent de la programmation d'une manière générale. Nous préférons réserver ce terme à un concept Python précis, lequel englobe les chaînes de caractères, les tuples et les listes (voir plus loin).

7 Dans certaines versions de l'éditeur Python pour *Window$*, le prompt secondaire n'apparaît pas.

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, (*C, Java*, ...) elles seraient 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") :

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

3.3 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[8]. (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*).

8 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.

3.4 Instructions composées – Blocs d'instructions

La construction que nous avons utilisée avec l'instruction if est notre premier exemple d'instruction composée. Vous en rencontrerez bientôt d'autres. Toutes les instructions composées en Python ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie par une ou 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). Elles constituent ce que nous appellerons désormais un **bloc d'instructions** (ou une suite d'instructions). 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 par **if** est vraie.

3.5 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
```

Ce fragment de programme n'imprime la phrase "c'est peut-être un chat" que si 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.

3.6 Quelques règles de syntaxe Python

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

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

Dans d'autres 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 saut à la ligne qui joue ce rôle. (Nous verrons plus loin comment 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.

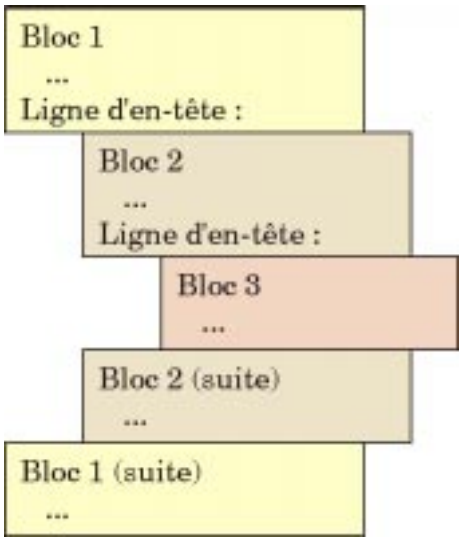
Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que BEGIN et END). En *C++* et en *Java*, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions 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 utilisant ces langages de se servir **aussi** les sauts à la ligne et de l'indentation pour bien délimiter visuellement les blocs.

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 à écrire du code lisible et à prendre de bonnes habitudes.

3.6.2 Instruction composée = En-tête , double point , bloc 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, ...) et **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.à.d. décalées vers la droite d'un même nombre d'espaces[9]). 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).

3.6.3 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.

9 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".

Chapitre 4 : Instructions répétitives.

4.1 Ré-affectation

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

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 pas 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. Sous Python, une instruction 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** maintenant, alors **a** continue à être égal à **b** pour toute la suite du raisonnement. Sous Python, une instruction d'affectation peut rendre égales les valeurs de deux variables, mais une instruction ultérieure peut changer l'une ou l'autre.

```
>>> 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 ?

Exercice :

e 3. Ecrivez 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 votre professeur vous demandera probablement 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 spécialisée, telle l'instruction SWAP du langage *Basic*). Sous Python, *l'affectation multiple* permet de programmer l'échange d'une manière élégante :

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

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

4.2 Premières itérations – 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 simples : la boucle construite à partir de l'instruction **while**.

Entrez 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>. Le programme s'exécute, et vous obtenez :

```
1
2
3
4
5
6
7
```

Que s'est-il passé ?

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 suivent, **tant que** le contenu de la variable **a** est (ou 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.à.d. décalées à droite d'un même nombre d'espaces).

Ainsi nous avons 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[10].
- Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant **le corps de la boucle**, c.à.d. :
  - l'instruction **a = a + 1** qui *incrémente* d'une unité le contenu de la variable **a**
    (c.à.d. que l'on affecte à la 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.à.d. que l'exécution reprend à la ligne contenant l'instruction **while**. La condition est à nouveau évaluée, et ainsi de suite. Tant que la *condition* **a < 7** reste vraie, le corps de la boucle est à nouveau exécuté et le bouclage se poursuit.

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

Notes :

- 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 !"
```

4.3 Elaboration de tables

Recommencez à présent le premier exercice, mais avec une petite modification :

```
>>> 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.

4.4 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 deux termes qui le précèdent. Analysez ce programme (qui utilise judicieusement l'*affectation multiple*). 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.

Exercice :

e 4. Essayez à présent d'écrire vous-même un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.

Chapitre 5 : Principaux types de données

5.1 Les données numériques

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

5.1.1 Le type "integer"

Modifions légèrement notre exercice sur la suite de *Fibonacci*, en demandant à Python d'en afficher un plus grand nombre de termes. Il suffit pour ce faire de modifier la condition dans la deuxième ligne. Avec "**`while c<51:`**" , nous devrions obtenir cinquante termes. Essayons :

```
>>> a,b,c = 1,1,1
>>> while c <51:
...     a,b,c = b,a+b,c+1
...     print b,

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657
46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887
9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296
433494437 701408733 1134903170 1836311903
Traceback (innermost last):
  File "<stdin>", line 2, in ?
OverflowError: integer addition
```

Nous avons obtenu en fait seulement 44 termes, le dernier des nombres affichés approchant les deux milliards. La suite s'interrompt alors sur un message d'erreur : "*OverflowError: integer addition*".

Ce message est affiché parce que lorsque le programme entame sa 45^{e} boucle, Python devrait assigner à la variable **b** la valeur 1134903170 + 1836311903, soit 2971215073. Or, lorsque nous avons créé la variable **b** (dans la première commande), nous lui avons assigné la valeur 1, laquelle est interprétée par Python comme une donnée de type "entier simple" ou ***integer***.
Au début du programme, la variable **b** a donc été définie implicitement comme étant elle aussi du type ***integer***. Dans la mémoire de l'ordinateur, ce type de donnée est 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).
En essayant d'outrepasser ces limites, nous provoquons une erreur de "dépassement de capacité" ou (*overflow error)* et l'exécution du programme s'interrompt.

5.1.2 Le type "float"

Que faire, dès lors, si nous voulons manipuler des nombres très grands ou très petits ?

La réponse est qu'il nous faut signaler à l'interpréteur Python que nous souhaitons utiliser un autre type de données. Nous en avons déjà rencontré précédemment un autre : le type "réel" (ou "virgule flottante"), appelé *floating point number* ou même simplement ***float*** en anglais).

Pour qu'une donnée numérique soit automatiquement considérée 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 une série dont les termes augmentent extrêmement vite, chacun d'eux étant égal au produit des deux précédents. Au huitième terme, nous dépassons déjà la capacité d'un *integer*. Au-delà, 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, mais sans message d'erreur cette fois : les nombres vraiment trop grands sont 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 sur 8 octets (64 bits) dans la mémoire de la machine.

5.1.3 Le type "long"

Python peut encore travailler avec un troisième type de donnée numérique : le type "entier long" (lequel est désigné par ***long*** dans la terminologie originale Python). Ce type permet de représenter des 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 !

On définit une donnée de ce type en ajoutant la lettre L à la fin de la valeur numérique. Exemple :

```
>>> a,b,c = 3L, 2L, 1   # => a & b seront du type 'long'
>>> while c<12:
...     a,b,c = b, a*b, c+1
...     print b

6L
12L
72L
864L
62208L
53747712L
3343537668096L
179707499645975396352L
60085879430566732227015542518579 2L
10797883156496691381438492294473845785924307043903078 4L
64880030544660752790736837369104977695001034284228042891827649456186234582611607
420928L
```

Dans cet exemple, nous commençons par assigner aux variables a, b et c les nouvelles valeurs numériques 3, 2 et 1. Ayant accolé un caractère L aux deux premières valeurs, nous indiquons à Python que les deux premières variables doivent être comprises comme étant du type "entier long". La variable c peut rester un entier simple, puisque nous l'utilisons comme compteur pour compter seulement jusqu'à douze.

Exercice :

e 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 64e case.
Écrivez un programme Python qui affiche le nombre exact de grains à déposer sur chacune des 64 cases du jeu.

5.2 Les données alphanumériques

5.2.1 Définition d'une chaîne de caractères ("string")

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. Il existe par exemple dans la plupart des langages de programmation une structure de données que l'on appelle ***chaîne de caractères*** (ou ***string*** en anglais).

Sous Python, une donnée de type ***string*** est une suite quelconque de caractères délimitée par des "simples quotes" (apostrophes) ou "doubles quotes" (guillemets).

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 d'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).

Dans une chaîne de caractère, l'antislash autorise l'insertion d'un certain nombre de codes sépiaux (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
- Rappelons encore ici que la casse est significative dans les noms de variables (Il faut respecter scrupuleusement le choix initial de majuscules ou minuscules).

"Triples 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[11] :

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

>>> print a1

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

5.2.2 Accès aux caractères d'une chaîne

Les chaînes sont un cas particulier d'un type de données 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, ces entités plus simples sont évidemment

11 L'utilisation de triples guillemets ou de triples apostrophes pour délimiter une chaîne de caractères n'est pas encore implémentée dans *PythonWin* (Interface graphique Python pour *Window$*)

les caractères.
Suivant les circonstances, nous souhaiterons traiter la chaîne comme un seul objet, ou au contraire la considérer comme une suite de caractères distincts.

Un langage de programmation tel que Python doit donc permettre d'accéder séparément à chacun des caractères d'une chaîne. Cela n'est pas très compliqué. En termes techniques, une chaîne est une collection ordonnée d'objets : les caractères de la chaîne sont disposés dans un certain ordre. Chaque caractère d'une chaîne peut donc être désigné par sa place dans la séquence, à l'aide d'un index. Sous Python, il suffit d'accoler un tel index au nom de la variable qui contient la chaîne, en l'écrivant entre crochets. D'une manière analogue, vous pouvez désigner tout un fragment (une "tranche") dans la chaîne, en utilisant deux index séparés par un double point.

Attention, cependant : comme vous aurez l'occasion de le vérifier par ailleurs, les données informatiques sont très souvent numérotées à partir de zéro (et non à partir de un). C'est le cas pour les caractères d'une chaîne. Analysez donc soigneusement les exemples ci-dessous :

```
>>> ch = "Stéphanie"
>>> print ch[0], ch[3]
S p
>>> print ch[0:4]
Stép
>>> print ch[4:8]
hani
```

La meilleure façon de se rappeler comment marche le "découpage en tranches" des exemples ci-dessus consiste à imaginer que les index pointent entre les caractères, le bord gauche du premier caractère étant noté zéro :

```
+---+---+---+---+---+---+---+---+---+
| S | t | é | p | h | a | n | i | e |
+---+---+---+---+---+---+---+---+---+
0   1   2   3   4   5   6   7   8   9
```

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
Sté
>>> print ch[3:]       # tout sauf les 3 premiers caractères
phanie
```

Nous verrons plus loin que Python dispose de nombreuses ***fonctions intégrées***. Par exemple, la fonction **len()** permet de connaître la longueur (le nombre de caractères) d'une chaîne :

```
>>> print len(ch)
9
```

Exercices :

e 6. Déterminez vous-même ce qui se passe lorsque l'un ou l'autre des indices de découpage est erroné, et décrivez cela le mieux possible. (Si le second indice plus petit que le premier, par exemple, ou bien si le second indice est plus grand que la taille de la chaîne).

e 7. Découpez une grande chaîne en fragments de 5 caractères chacun. Réassemblez ces morceaux dans l'ordre inverse.

5.3 Les listes (première approche)

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de ***données composites***, lesquelles 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**. Nous n'allons aborder ici que le premier de ces trois types, et 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 définit une liste comme ***une suite 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 ci-dessus, les éléments qui constituent une liste peuvent être de types variés. Dans notre exemple, les trois premiers éléments sont des chaînes de caractères, le quatrième un entier, le cinquième un réel, etc. (Nous verrons 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" ou de "variable indicée" que l'on rencontre dans d'autres langages de programmation.

Les éléments d'une liste sont bien entendu référencés chacun par un index. Et comme c'était déjà le cas pour les caractères dans une chaîne, il faut se rappeler ici également que la numérotation de ces index commence à partir de zéro et non de un.

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

A la différence de ce qui se passe pour les chaînes, qui constituent un type de données non-modifiables (nous aurons l'occasion 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 remplacer certains éléments :

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

On peut en enlever :

```
>>> jour[3:5] = []                    # [] est une liste vide
>>> print jour
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
```

On peut en insérer :

```
>>> jour[2:2] = [7000,8000,9000]
>>> print jour
['lundi', 'mardi', 7000, 8000, 9000, 'mercredi', 'jeudi', 'vendredi']
```

Note : si l'on n'insère qu'un seul élément, il faut s'assurer que cet élément soit du type 'liste'. Ainsi l'élément jour[4] n'est pas une liste (c'est l'entier 9000), alors que l'élément jour[5:7] en est une. Pour insérer un élément qui n'est pas déjà une liste, il faut le présenter entre crochets :

```
>>> jour[1:1] = [jour[4]]
>>> print jour
['lundi', 9000,'mardi', 7000, 8000, 9000, 'mercredi', 'jeudi', 'vendredi']
```

La fonction intégrée **len()** s'applique aussi aux listes (elle indique le nombre d'éléments présents dans la liste) :

```
>>> len(jour)
9
```

Nous aurons encore beaucoup de choses à apprendre à propos des listes. Signalons simplement ici que sous Python, les listes sont une catégorie particulière d'***objets***, auxquels on pourra appliquer différentes ***méthodes*** (ces termes seront expliqués plus loin).

Avant cela, reprenons quelques exercices de programmation. Analysez le petit programme ci-dessous et commentez 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*", fréquemment utilisé en programmation, et que l'on représente par % dans de nombreux langages (dont Python).

Quelle est l'opération effectuée par cet opérateur ?

Exercice :

e 8. 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...]`**.

Chapitre 6 : Premiers scripts

6.1 Comment conserver nos programmes ?

Jusqu'à présent, nous avons utilisé Python en mode interactif. Ainsi nous avons pu 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 nous avons écrites disparaissent irrémédiablement dès que nous fermons l'interpréteur. Avant d'aller plus loin dans notre étude, il est donc temps que nous apprenions sauvegarder nos programmes dans des fichiers sur disque, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, nous allons maintenant rédiger nos séquences d'instructions en utilisant un éditeur de textes quelconque (par exemple *Kedit* ou *Kwrite* sous *KDE-Linux*, *Edit* sous *MS-DOS*, *Notepad* sous *Window$*, etc.), ou encore une interface graphique de développement telle que *PythonWin* ou *IDLE*, et nous allons ainsi écrire un script que nous pourrons ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur[12].

Lorsque nous voudrons faire s'exécuter notre programme, il nous suffira de lancer l'interpréteur Python en lui fournissant le nom du fichier contenant le script. Par exemple, si nous avons placé un script dans un fichier nommé "MonScript", il suffit d'entrer la commande suivante dans une fenêtre de terminal pour que le script s'exécute :

```
Python MonScript
```

Nous pouvons faire mieux encore, en donnant au fichier un nom qui se termine par l'extension **.py**

Si nous respectons cette convention, nous pouvons faire s'exécuter le script simplement en cliquant sur son nom (à l'aide de la souris) dans dans le gestionnaire de fichiers (l'explorateur), parce que le système d'exploitation de l'ordinateur "sait" qu'il doit lancer l'interpréteur Python chaque fois que l'on essaye d'ouvrir un fichier dont le nom se termine par **.py**. (Ceci suppose bien entendu que le gestionnaire de fichiers de notre machine ait été correctement configuré).

Un script Python contiendra donc des séquences d'instructions identiques à celles que nous avons expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues peut-être beaucoup plus tard, il est très fortement conseillé de les expliciter le mieux possible en y adjoignant de nombreuses remarques. Un bon programmeur s'efforce toujours d'insérer un grand nombre de commentaires dans ses scripts, de manière à ce que d'autres que lui puissent aisément comprendre ses algorithmes. Lui-même sera d'ailleurs très heureux de retrouver ses commentaires s'il doit retravailler un des ses propres programmes après quelques années.

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

12 Il est évidemment préférable d'utiliser un éditeur "intelligent" tel que IDLE, qui vous aide à éviter les fautes de syntaxe. Avec IDLE sous Window$, 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 de le sauvegarder, puis de suivre le menu : Edit → Run script (ou de frapper CTRL-F5).

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

```
#!/usr/bin/env python

# 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
```

Comme vous l'aurez remarqué, nous avons inséré tout au début du script une ligne contenant le "faux commentaire" suivant :

```
#! /usr/bin/env python
```

Il vous est conseillé (ce n'est pas indispensable) de toujours commencer un script Python ainsi, parce que cette ligne rend le script directement exécutable depuis la ligne de commande d'un shell dans les environnements de type *Unix (Linux*, etc.). Cela signifie qu'il suffira de frapper son nom, ou de cliquer dessus, pour le lancer.

Afin de montrer tout de suite le bon exemple, nous avons ajouté ensuite deux autres lignes de pur commentaire contenant une courte description de ce que le programme est censé faire. Faites de même dans vos propres scripts.

Après avoir bien vérifié votre texte, sauvegardez-le et exécutez-le (Si vous travaillez en mode texte sous *Linux* ou sous *M$DOS*, lancez la commande "python <nom du script>". Si vous travaillez en mode graphique sous *Linux*, vous pouvez ouvrir une fenêtre de terminal, et faire la même chose. Sous *KDE* ou *Window$*, et à la condition d'avoir respecté les conventions décrites plus haut, vous pouvez lancer l'exécution de votre script en effectuant un simple clic sur l'icône correspondante. 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>. Consultez votre professeur concernant les autres possibilités de lancement sur d'autres systèmes d'exploitation).

Exercices (à chaque fois, écrire le script capable d'effectuer le travail demandé) :

e 9. Calculer le volume d'un parallélipipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.

e 10. Convertir 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 : %).

e 11. Convertir en radians un angle fourni au départ en degrés, minutes, secondes

e 12. Convertir en degrés, minutes, secondes un angle fourni au départ en radians.

e 13. Convertir en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse. La formule de conversion est : $T_F = T_C \times 100 + 32$.

6.2 Importation de modules. Utilisation de fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de **fonction**[13]. 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 réécrire à chaque fois.

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 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 **librairies**.

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 *) du module *math*, lequel contient une librairie 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* la valeur du sinus de *angle* (en radians !), etc.

Exemple :

```
#!/usr/bin/env python
# Démo : utilisation des fonctions du module <math>

from math import *

nombre = 121
angle = pi/6  # soit 30° (la librairie math inclut aussi la définition de pi)
print 'racine carrée de', nombre, '=', sqrt(nombre)
print 'sinus de', angle, 'radians', '=', sin(angle)
```

13 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.

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 est constituée d'**un nom quelconque associé à des parenthèses**
  exemple : sqrt()
- dans les parenthèses, on indique à la fonction un ou plusieurs **arguments**
  exemple : sqrt(121)
- la fonction fournit une **valeur en retour** (on dira aussi qu'elle "retourne" 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'oeil dans la documentation des librairies 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. Avant cela, analysons encore un exemple avec une fonction intégrée particulièrement importante :

6.3 Interaction avec l'utilisateur - entrée d'informations avec 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 simple en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction **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 :

```
#!/usr/bin/env python
# Utilisation de la fonction intégrée input()

from math import *
print 'Veuillez entrer un nombre positif quelconque : ',
nn = input()
print 'La racine carrée de', nn, 'vaut', sqrt(nn)
```

Notes :

- La fonction **input()** retourne une valeur dont le type correspondant 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.à.d. 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, vous préférerez peut-être utiliser la fonction similaire **raw_input()**, laquelle retourne toujours une chaîne de caractères.

Exercices :

e 14. Convertir en mètres par seconde et en km/h une vitesse fournie par l'utilisateur en miles/heure. (1 mile = 1609 mètres)

e 15. Calculer le périmètre et l'aire d'un triangle quelconque dont on vous fournit les 3 côtés. (L'aire d'un triangle quelconque se calcule à l'aide de la formule : $S = \sqrt{(d.(d-a).(d-b).(d-c))}$, dans laquelle d désigne la longueur du demi-périmètre, et a, b, c celles des trois côtés.

e 16. Calculer la période d'un pendule simple de longueur donnée. La formule qui permet de calculer la période d'un pendule simple est $T = 2\pi.\sqrt{(l/g)}$, l représentant la longueur du pendule et g la valeur de l'accélération de la pesanteur au lieu d'expérience.

6.4 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.à.d. 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 retourne 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.

6.5 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.

6.5.1 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 !) :

```
#!/usr/bin/env python
# Utilisation d'une liste et de branchements conditionnels

print "Ce script recherche le plus grand de trois nombres"
nn = input('Veuillez entrer trois nombres séparés par des virgules : ')
# Note : la fonction input() retourne une valeur du type correspondant
# à ce que l'utilisateur a entré. nn contiendra donc ici une liste.
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.

6.5.2 Boucle while - Instructions imbriquées

Continuons dans cette voie en imbriquant d'autres structures :

```
#!/usr/bin/env python                                                  #  1
# Instructions composées <while> - <if> - <elif> - <else>              #  2

print 'Choisissez un nombre de 1 à 3 (ou zéro pour terminer) ',        #  3
a = input()                                                            #  4
while a != 0:           # l'opérateur != signifie "différent de"       #  5
    if a == 1:                                                         #  6
        print "Vous avez choisi un :"                                  #  7
        print "le premier, l'unique, l'unité ..."                      #  8
    elif a == 2:                                                       #  9
        print "Vous préférez le deux :"                                # 10
        print "la paire, le couple, le duo ..."                        # 11
    elif a == 3:                                                       # 12
```

```
        print "Vous optez pour le plus grand des trois :"              # 13
        print "le trio, la trinité, le triplet ..."                    # 14
    else :                                                             # 15
        print "Un nombre entre UN et TROIS, s.v.p."                    # 16
    print 'Choisissez un nombre de 1 à 3 (ou zéro pour terminer) ',    # 17
    a = input()                                                        # 18
print "Vous avez entré zéro :"                                         # 19
print "L'exercice est donc terminé."                                   # 20
```

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 "sortir" 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 :

e 17. 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'
```

e 18. 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'
```

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

e 19. 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=33 → 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 33, le résultat devrait donc être : 0+3+5+6+9+10+12+15+18+20+21+24+25+27+30 = 225.

e 20. 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).

e 21. Demander à l'utilisateur son nom et son sexe (M ou F). En fonction de ces données, afficher "Cher Monsieur" ou "Très chère amie" suivi du nom de l'élève.

e 22. 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.

e 23. 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.

e 24. 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

e 25. 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.

e 26. Écrivez un script capable d'afficher la liste de tous les jours d'une année imaginaire, laquelle commencerait un Jeudi :
Jeudi 1 Janvier Vendredi 2 Janvier Samedi 3 Janvier Dimanche 4 Janvier
... et ainsi de suite jusqu'au Jeudi 31 Décembre.

Chapitre 7 : Les fonctions

7.1 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 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**[14] 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 NOM(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[15], 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 est conseillé de n'utiliser que 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).

14 Il existe aussi dans d'autres langages des **routines** (parfois appelés sous-programmes) et des **procédures**.
Python utilise le même terme de **fonction**, pour désigner à la fois les fonctions au sens strict (qui fournissent une valeur en retour), et les procédures (qui n'en fournissent pas, ou retournent plutôt une valeur sans objet).

15 Voir page 15.

7.1.1 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 dans 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[16], 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 :

```
>>> 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.

16 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 veut 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.

- 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.

7.1.2 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**.

7.1.3 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** . A 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 la même valeur).

7.1.4 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
...
```

7.2 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*[17]. 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):
  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
```

17 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.

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 globale et l'autre 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.à.d. 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
>>>
```

7.3 "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[18]. Une "vraie" fonction (au sens strict) doit en effet *retourner* 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() retourne 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 "retournée" 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 retourne 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. Nous en profiterons pour apprendre dans la foulée 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 forme d'une liste :

```
>>> 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)

18 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.

Notes:

- Comme nous l'avons vu dans l'exemple précédent, l'instruction **return** définit ce que doit être la valeur "retournée" par la fonction. En l'occurrence, il s'agira ici du contenu de la variable **result**, c.à.d. la liste des nombres générés par la fonction[19].
- L'instruction **result.append(b)** est notre premier exemple de l'utilisation d'un concept important sur lequel nous reviendrons 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 rien d'autre en fait 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. Concrètement, on met en œuvre la méthode **methode4()** d'un objet **objet3** à l'aide d'une instruction du type "**objet3.methode4()**" (c.à.d. le nom de l'objet, puis le nom de la méthode, reliés par un point).

  Dans notre exemple, nous appliquons donc la méthode **append()** à l'objet **result**. Les listes sont en effet considérés par Python comme des objets, et on peut effectivement leur appliquer toute une série de méthodes. En l'occurrence, la méthode **append()** est donc une fonction qui sert à ajouter un élément à la fin de la liste concernée. 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 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 recopiage préalable 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 s'il s'agit de traiter des listes volumineuses).

- Il n'est nullement indispensable que la valeur retournée 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.

19 **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 **None** (valeur nulle, chaîne ou liste vide).

7.4 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 à la suite de l'autre, dans l'ordre où elles apparaissent dans le code source. Dans le 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 fonction 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 du module **__main__**, où qu'elle puisse se trouver dans le listing (en général ce sera donc plutôt vers la fin).

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.

Dans notre exemple, La fonction principale **__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 concrètement que vous devrez souvent analyser un script en commençant par les dernières lignes !

Exercices :

e 27. Définissez une fonction **SurfCercle(R)**. Cette fonction doit retourner 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**

e 28. Définissez une fonction **VolBoite(x1,x2,x3)** qui retourne 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**

e 29. Définissez une fonction **Max(n1,n2,n3)** qui retourne le plus grand de 3 nombres n1, n2, n3 fournis en arguments. Par exemple, l'exécution de l'instruction :
`print Max(2,5,4)` doit donner le résultat : **5**

e 30. Définissez une fonction **CompteCar(Ca,Ch)** qui retourne 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**

e 31. Définissez une fonction **NomMois(n)** qui retourne le nom du n^{e} mois de l'année.
Par exemple, l'exécution de l'instruction :
`print NomMois(4)` doit donner le résultat : **Avril**

e 32. Définissez une fonction **ChangeCar(Ca1,Ca2,Ch)** qui remplace tous les caractères **Ca1** par des caractères **Ca2** dans la chaîne de caractères **Ch**. Par exemple, l'exécution des instructions :

```
Phrase = 'Ceci est une toute petite phrase.'
ChangeCar(' ', '*', Phrase)
print Phrase
```

doit afficher : `Ceci*est*une*toute*petite*phrase.`

7.5 Typage des paramètres

Vous avez appris que le typage des variables sous Python est un typage dynamique. 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 sera le même que celui de l'argument qui aura 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
```

Vous pouvez constater que la même fonction afficher3fois() accepte aussi bien un argument numérique qu'une chaîne de caractères ou même une liste. Le dernier exemple montre que l'on peut aussi utiliser une expression. Dans ce cas l'expression est d'abord évaluée, et c'est le résultat qui est transmis ensuite comme argument.

7.6 Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et 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. 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 :
« **question('Voulez-vous vraiment terminer ? ')** » ou bien :
« **question('Faut-il effacer ce fichier ? ', 3)** », ou même encore :
« **question('Avez-vous compris ? ', 2, 'Vous devez répondre par oui ou par non !')** ».

(Essayez et décortiquez ces exemples)

7.7 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 correspondants 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

>>>
```

Nous allons voir directement d'autres exemples de ce mécanisme dans le chapitre suivant.

Chapitre 8 : Utilisation de fenêtres et de graphismes

8.1 Interfaces graphiques (GUI)

Jusqu'à présent, nous avons utilisé Python exclusivement en mode «texte». Nous avons procédé ainsi afin de découvrir le plus rapidement possible les bases essentielles du langage. Nous ne sommes certainement pas près d'en avoir terminé, mais avant d'aller plus loin nous allons nous distraire un peu en allant dès à présent faire une petite incursion dans le vaste domaine des interfaces graphiques.

Si vous ne le savez pas encore, apprenez-le maintenant : le domaine des interfaces graphiques (ou *GUI : Graphical User Interface*) est extrêmement complexe. Chaque système d'exploitation peut en effet proposer plusieurs «librairies» de fonctions graphiques de base, auxquelles viendront 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 **propriétés** et les **méthodes**.

Avec Python, la librairie graphique la plus utilisée jusqu'à présent est la librairie ***Tkinter***, qui est une adaptation de la librairie ***Tk*** développée à l'origine pour le langage ***Tcl***. D'autres librairies graphiques fort intéressantes ont été proposées pour Python, telles les librairies ***GTK*** et ***QT/KDE***, mais il ne nous est évidemment pas possible de les étudier toutes. De toute façon, ces librairies ne sont pas nécessairement disponibles pour tous les systèmes d'exploitation qui nous intéressent. Nous allons donc nous limiter d'abord à quelques expériences la librairie *Tkinter*, dont il existe fort heureusement des versions similaires (et gratuites) pour les plate-formes *Linux*, *Window$* et *Mackinto$h*. Lorsque nous commencerons à élaborer des programmes plus ambitieux, nous utiliserons également la librairie ***wxPython***, qui est plus performante.

8.2 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*»[20] 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.quit)
>>> bou1.pack()
>>> fen1.mainloop()
```

20 Un «widget», c'est littéralement un «machin». Dans certains environnements de programmation, on appellera cela plutôt un «contrôle» ou un «composant». 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. Nous préférerions pour notre part une appellation telle que « objet graphique ».

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. (A ce propos, et si vous effectuez cet exercice sous *Window$*, nous vous conseillons d'utiliser de préférence une version standard de Python, laquelle permet de mieux observer ce qui se passe après l'entrée de chaque commande, plutôt que la version *PythonWin*).

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

1. Comme déjà 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.à.d. 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 **classes**. En informatique, on appelle ainsi des **modèles d'objets**, lesquels sont eux-mêmes des morceaux de programmes réutilisables.
   Plutôt que d'essayer de fournir ici une définition définitive et précise de ce que sont les objets et les classes, nous allons directement commencer à en utiliser quelques-uns. 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*** qui sera notre premier ***objet,*** à savoir la fenêtre **fen1**.

   Ce processus d'instanciation d'un objet à partir d'une classe est une opération fondamentale dans la technique moderne de programmation que l'on appelle *"programmation orientée objet" (ou OOP : Object Oriented Programming).*

   La classe est en quelque sorte un modèle général (ou un patron) à partir duquel on demande à la machine de construire un objet informatique particulier. La classe contient en effet 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 librairie *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.

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

3. A la troisième ligne : "**tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')**", nous créons un autre *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 plus précisément de fragments de texte quelconques, destinés à l'affichage de messages divers à l'intérieur d'une fenêtre.
   En nous efforçant d'apprendre au passage la manière correcte d'exprimer les choses, nous dirons donc que nous créons ici l'objet **tex1** par instanciation de la classe **Label()**.

   Remarquons ici que nous faisons appel à une classe de la même manière que nous faisons appel à une fonction, c.à.d. en fournissant un certain nombre d'arguments entre parenthèses. Nous verrons plus loin qu'une classe est en fait un 'conteneur' dans lequel sont regroupées des fonctions.

Quels arguments avons-nous donc fournis pour l'instanciation ?

- Le premier argument transmis (**fen1**), indique que ce nouveau widget que nous sommes en train de créer sera contenu dans un autre *widget* préexistant, que nous désignons donc ici comme son «parent» : l'objet **fen1** est le *widget parent* de l'objet **tex1**. (On pourra dire aussi que l'objet **tex1** est un *widget enfant* de l'objet **fen1**).
- Les deux arguments suivants servent à préciser un peu plus la forme exacte que doit prendre notre objet, en ajoutant dans la commande deux options de création, chacune fournie sous la forme d'une chaîne de caractères : d'abord le texte de l'étiquette, et ensuite sa couleur d'avant-plan (ou *foreground*, en abrégé **fg**). Ainsi le texte que nous voulons afficher est bien défini, et il devra apparaître 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 une valeur par défaut dans les définitions internes de la classe **Label().** Nous ne devons donc indiquer des options que pour les caractéristiques que nous voulons 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()**. Il convient de préciser que le terme de *méthode* employé ici est encore une fois un mot qui a un sens particulier dans les textes qui parlent de la programmation d'ordinateur avec utilisation d'objets. Nous apprendrons bientôt qu'un objet informatique est en fait un morceau de programme contenant toujours deux séries d'entités :
   - un certain nombre de données (numériques ou autres), contenues dans des variables de types divers : on les appelle les **propriétés** (ou les attributs) de l'objet.
   - un certain nombre de procédures ou de fonctions (qui sont donc des algorithmes) : on les appelle les **méthodes** de l'objet.

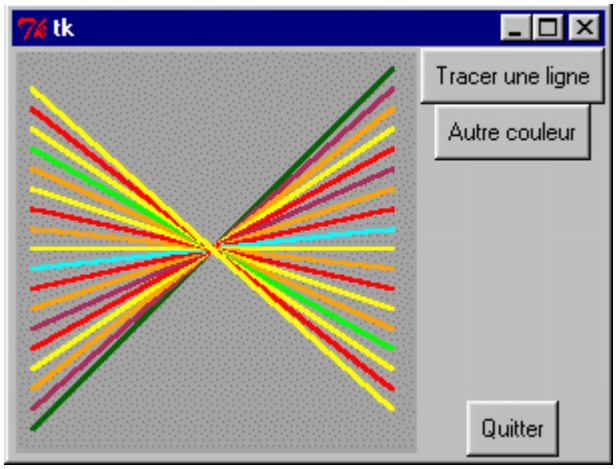
   La méthode **pack()** fait partie d'un ensemble de méthodes qui sont applicables non seulement aux *widgets* de la classe Label(), mais aussi à la plupart des autres *widgets* Tkinter, et qui agissent sur leur disposition géométrique dans la fenêtre. Comme vous pouvez le constater par vous-même si vous entrez les commandes de notre exemple une par une, la méthode **pack()** réduit automatiquement la taille de la fenêtre "parent" afin qu'elle soit juste assez grande pour contenir les *widgets* "enfants" définis au préalable.
5. A la cinquième ligne : "**bou1 = Button(fen1, text='Quitter', command = fen1.quit)**", nous créons notre second *widget* "enfant" : un bouton, cette fois.
   Comme nous l'avons fait pour le widget précédent, nous appelons la classe Button en fournissant entre parenthèses un certain nombre d'arguments. Etant 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 **quit** associée à l'objet **fenetre1**, ce qui devrait provoquer la fermeture de la fenêtre[21].
6. La sixième ligne utilise encore la méthode **pack()** pour adapter la géométrie de la fenêtre.
7. La septième ligne : "**fen1.mainloop()**" est très importante, parce que c'est elle qui provoque le démarrage du gestionnaire 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 boucle de programme qui "tourne" en permanence au niveau principal de l'exécution du programme, en testant continuellement tous les périphériques d'entrée de l'ordinateur (souris, clavier, etc.).

21 Ceci ne marche pas nécessairement avec toutes les versions de Python fonctionnant en mode interactif. Dans ce cas vous devrez fermer la fenêtre à l'aide de son icône. Par contre, cela marchera toujours dans un script.

8.2.1 Exemple de script : "Tracé de lignes"

```
#! /usr/bin/env python
# Petit exercice utilisant la librairie graphique Tkinter
from Tkinter import *
from whrandom import *

# 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

# définition de 2 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 = int(random()*8)      # => génère un nombre aléatoire de 0 à 7
    coul = pal[c]

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

# Création du widget principal ("parent") :
fen1 = Tk()
# création des widgets "enfants" :
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()

# démarrage de l'observateur d'évènements (boucle principale) :
fen1.mainloop()
```



Exercices : modifications au programme "Tracé de lignes" ci-dessus.

e 33. Comment faut-il modifier le programme pour ne plus avoir que des lignes de couleur cyan, maroon et green ?

e 34. Comment modifier le programme pour que toutes les lignes tracées soient horizontales et parallèles ?

e 35. 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.

e 36. 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.

e 37. 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 !)

e 38. 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 !)

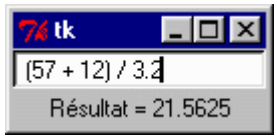
Si vous définissez "x1, y1, x2, y2 = 10, 390, 390, 10" à la place de "global x1, y1, ...", Que se passe-t-il ? Pourquoi ? Quelle conclusion pouvez-vous tirer de tout cela ?

e 39. 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

e 40. Dans votre cahier, 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). Laisser de la place pour compléter ultérieurement.

8.2.2 Exemple de script : "Calculatrice ultra-simplifiée"

```
#! /usr/bin/env python
# Exercice utilisant la librairie 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['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()
```



Dans cet exemple, 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 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 modifie la propriété "text" d'un objet appelé "chaine" (lequel est lui-même défini à l'intérieur du corps du programme principal). Ladite propriété reçoit donc ici une nouvelle valeur, définie 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 prédéfinies dans Python : **eval()** et **str()**, et d'une méthode associée à un widget Tkinter : la méthode **get()**.

eval() lance l'exécution d'une commande Python décrite dans une chaîne de caractères et fournit en retour le résultat de cette commande. Exemple :

```
chaine = "(25 + 8)/3"     # définition d'une chaîne de caractères
res = eval(chaine)        # évaluation de la commande contenue dans la chaîne
print res +5              # => le contenu de la variable res est numérique
```

str() transforme une expression numérique en chaîne de caractères. Étant donné que la fonction eval() décrite ci-dessus 'retourne' une valeur numérique, nous devons re-transformer celle-ci en chaîne de caractères si nous voulons l'incorporer au message "Résultat = ".

get() est une méthode associée à un widget 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.

Dans le corps du programme principal, nous définissons une fenêtre à partir de la classe Tk, et cette fenêtre contient deux widgets : un widget "chaine" instancié à partir de la classe Label, et un widget "entree" instancié à partir de la classe Entry. A ce dernier widget, nous associons un événement à l'aide de la méthode **bind** :

```
entree.bind("<Return>",evaluer)
```

Ce qui signifie : *pour l'objet "entree", il faut lier (bind) l'événement "pression sur la touche Return" au gestionnaire d'événement "evaluer"*. Comme nous l'avons déjà vu précédemment, un gestionnaire d'événement sera défini sous Python comme une simple fonction. L'événement lui-même est décrit dans une chaîne de caractères (on peut ainsi faire référence à un grand nombre d'événements, tels les mouvements et les clics de la souris, l'enfoncement des touches du clavier, le redimensionnement ou la mise en avant d'une fenêtre, etc.). Notons aussi au passage la syntaxe des instructions qui mettent en oeuvre une méthode associée à un objet :

objet.méthode(arguments)

On écrit d'abord le nom de l'objet sur lequel on désire intervenir, puis un point, puis le nom de la méthode à mettre en oeuvre, puis on ouvre des parenthèses entre lesquelles on fournit un certain nombre d'arguments (variable suivant l'objet et/ou la méthode utilisé(e)).

Dans la définition de la fonction "evaluer", vous pouvez remarquer que nous avons fourni un argument **event** (entre les parenthèses). Il s'agit d'un objet Python standard, auquel sont associés un certain nombre d'attributs. Exemple :

```
#! /usr/bin/env python
# Détection et positionnement d'un clic de souris dans une fenêtre :

from Tkinter import *

def pointeur(event):
  chaine['text'] = "Clic détecté en X =" + str(event.x) + ", Y =" + str(event.y)

root = Tk()
fen1 = Frame(root, width =200, height =150, bg="light yellow")
fen1.bind("<Button-1>", pointeur)
fen1.pack()
chaine = Label(root)
chaine.pack()

fen1.mainloop()
```

Ce petit script fait apparaître une fenêtre contenant une surface carrée de couleur jaune clair, dans laquelle vous êtes invité à effectuer des clics de souris. On voit dans le corps du programme qu'une instruction bind associe l'événement "clic à l'aide du premier bouton de la souris (celui de gauche)" au gestionnaire "pointeur".

Dans la définition de celui-ci, on utilise les attributs x et y associés à l'objet standard **event** pour construire la chaîne de caractères qui affichera la position de la souris au moment du clic.

Après ces quelques exemples et exercices, il est certainement nécessaire que vous preniez maintenant connaissance de la liste des widgets disponibles.

8.3 Exploration de quelques classes de widgets Tkinter

***Note** : Le petit guide qui suit ne constitue pas une référence complète, loin s'en faut ! Si vous souhaitez davantage de précisions, nous vous invitons à consulter le texte « An introduction to Tkinter », par Fredrik Lundh, document PDF disponible sur le site www.python.org.*

Il existe 15 classes de widgets :

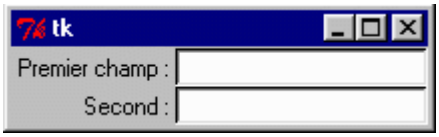
Widget	***Description***
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
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	C'est le widget "fenêtre" qui contient tous les autres. Cette fenêtre peut posséder une bordure et un fond coloré.
Label	Un texte 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".

A chacun de ces widgets sont associées un grand nombre de méthodes. Nous allons en découvrir quelques-unes. On peut aussi leur lier des événements, comme nous venons de le voir. Enfin, il vous faut encore savoir que tous ces widgets peuvent être disposés géométriquement sur les fenêtres en utilisant trois méthodes différentes : la méthode **Grid**, la méthode **Pack** et la méthode **Place**.

L'utilité de ces méthodes apparaît clairement lorsque l'on souhaite réaliser des programmes **portables** (c.à.d. susceptibles de fonctionner indifféremment sur des systèmes d'exploitation aussi différents que UN!X, M@C ou Window$), et dont les fenêtres soient redimensionnables.

8.4 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. Essayez par exemple :

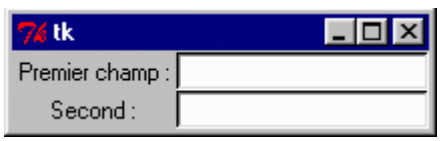
```
from Tkinter import *

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 escomptions !!! :

Pour mieux comprendre comment fonctionne la méthode pack, vous pouvez essayer maintenant 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 bien temps d'apprendre une autre méthode. Analysez donc le script ci-dessous : il contient en effet (presque) la solution de notre petit problème :

```
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. On y indique simplement dans quelle ligne (*row*, en anglais) et dans quelle colonne (*column*) on veut placer le widget. On peut numéroter les lignes et les colonnes comme on veut, en partant de zéro, ou de un, ou encore d'un nombre quelconque : 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 donc 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 faut encore améliorer légèrement notre script, en adjoignant à la méthode grid() utilisée pour ces widgets une option "**sticky**". Cette option 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 éventuellement remplacer le nom du fichier image (Martin_P.gif) par le nom d'une image de votre choix. Attention : la librairie Tkinter standard n'accepte qu'un petit nombre de formats image. Choisissez de préférence le format GIF.

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 y 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 à partir de la classe PhotoImage[22].

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,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 indiquent l'épaisseur de l'espace à réserver autour de ce widget (en largeur et en hauteur).

...Et puisque nous l'avons "sous la main", nous pouvons encore profiter de ce script pour apprendre à simplifier notre code. Python permet en effet une écriture extrêmement compacte, comme nous allons le voir.

22 Il existe d'autres classes d'images, mais pour les utiliser il faut importer dans le script d'autres modules que la seule librairie Tkinter. Vous pouvez par exemple expérimenter la librairie PMW (Python Mega Widgets).

1. La première chose que nous pouvons souvent faire (pas toujours !), c'est appliquer la méthode de mise en page des widgets (la méthode grid(), en l'occurrence) directement au moment même où nous créons ces widgets. Le code correspondant devient un peu plus fouillé, mais il reste très lisible. Nous pouvons par exemple remplacer les deux lignes :
   ```
   txt1 = Label(fen1, text = 'Premier champ :')
   txt1.grid(row =1,sticky =E)
   ```
   par une seule, qui serait :
   ```
   txt1 = Label(fen1, text = 'Premier champ :').grid(row =1,sticky =E)
   ```
2. Une deuxième simplification consistera à supprimer les variables que nous avons définies pour bien dégager les étapes de notre démarche, mais qui ne servent à rien dans le programme. En effet : jusqu'à présent, lorsque nous avons créé des objets divers par instanciation à partir d'une classe quelconque, nous les avons chaque fois confiés à des variables. Par exemple, lorsque nous avons écrit :
   `txt1 = Label(fen1, text = 'Premier champ :').grid(row =1,sticky =E)` nous avons créé une instance de la classe Label que nous avons placé dans la variable **txt1**.
   Que faisons-nous ensuite de cette variable dans le script ? Plus rien.
   Dans le script original, nous avions besoin de cette variable pour désigner à la méthode grid() le widget auquel elle devait s'appliquer. A partir du moment ou nous appliquons la méthode grid() directement au moment de la création du widget, cette variable peut devenir inutile (à la condition toutefois que nous n'ayons plus besoin par après d'aucune référence à ce widget).
   Notre ligne de script peut donc maintenant s'écrire :
   `Label(fen1, text = 'Premier champ :').grid(row =1,sticky =E)` Dans le cas des widgets de la classe Label, nous pourrons souvent omettre ainsi l'insertion dans une variable, car généralement ces widgets ne seront plus référencés après qu'on les ait créés. Pour d'autres widgets, par contre, tels ceux de la classe Entry par exemple, il faudra bien utiliser des variables, de manière à pouvoir interagir encore avec ces widgets plus loin dans le programme.
3. La troisième simplification qui nous est encore accessible, c'est que nous pouvons omettre l'indication de nombreux numéros de lignes et de colonnes. A partir du moment où vous utilisez la méthode grid() quelque part dans votre script pour positionner des widgets, Tkinter considère qu'il existe des lignes et des colonnes[23]. Si un numéro de ligne ou de colonne n'est pas indiqué, le widget correspondant sera placé dans la première case vide disponible.

Analysez à présent le script ci-dessous, dans lequel ces simplifications ont été appliquées :

```
from Tkinter import *
fen1 = Tk()
# création de widgets 'Label' et 'Entry' :
Label(fen1, text = 'Premier champ :').grid(sticky =E)
Label(fen1, text = 'Second :').grid(sticky =E)
Label(fen1, text = 'Troisième :').grid(sticky =E)
entr1 = Entry(fen1).grid(row =0, column =1)
entr2 = Entry(fen1).grid(row =1, column =1)
entr3 = Entry(fen1).grid(row =2, column =1)
chek1 = Checkbutton(fen1, text ='Case à cocher, pour voir').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)
# démarrage :
fen1.mainloop()
```

23 Surtout, n'utilisez pas plusieurs méthodes de positionnement différentes dans le même script ! Les méthodes grid(), pack() et place() sont mutuellement exclusives.

8.5 Modification des propriétés d'un objet - Animation

A ce stade de votre apprentissage, vous souhaitez certainement pouvoir faire apparaître un petit dessin quelconque dans un canevas, puis le déplacer à volonté, par exemple à l'aide de boutons. Veuillez donc 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)

# définition des 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 ("parent") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")

# création des widgets "enfants" :
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 de l'observateur d'évènements (boucle principale) :
fen1.mainloop()
```

Le corps de ce programme correspond à ce que vous connaissez déjà : On y crée une fenêtre Tk dans laquelle on installe un canevas contenant lui-même un cercle coloré, plus cinq boutons de contrôle. Remarquez tout de même au passage que nous n'avons plus placé les widgets "boutons" dans des variables (c'est inutile, puisque nous n'y faisons plus référence par après), et que nous avons aussi appliqué la méthode pack() directement au moment de la création de ces objets.

La vraie nouveauté de ce programme réside dans la fonction définie tout au début du script. Chaque fois qu'elle sera appelée, cette fonction (ou procédure) va redéfinir les coordonnées de l'objet "cercle coloré" que nous avons créé dans le canevas. Ainsi nous ne créons pas un nouvel objet, mais nous modifions les propriétés d'un objet existant.

Il s'agit là d'une manière de procéder tout à fait caractéristique de la programmation "orientée objet". On crée des objets, puis on agit sur ces objets en leur appliquant des méthodes et en modifiant leurs propriétés.

Exercices :

e 41. Ecrire un programme qui fasse apparaître une fenêtre avec un canevas. Dans ce canevas on voit 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, on utilise évidemment la loi de la gravitation universelle de Newton.

e 42. 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.

e 43. 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 !).

e 44. Même exercice avec des charges électriques (loi de Coulomb). Donner cette fois une possibilité de choisir le signe des charges.

e 45. Ecrire un petit programme qui fasse apparaître une fenêtre avec deux champs : l'un indique une température en degrés Celcius, 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, consultez votre professeur de Physique. Revoyez aussi le petit programme concernant la calculatrice simplifiée.

e 46. 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.

e 47. 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.

e 48. Modifiez le programme ci-dessus, de telle manière que la balle en se déplaçant laisse derrière elle une trace de la trajectoire décrite.

e 49. Modifiez le programme ci-dessus de manière à tracer d'autres figures. Consultez votre professeur pour des suggestions (courbes de Lissajous).

e 50. 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.à.d. 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.

8.6 Animation automatique - Récursivité

Pour conclure (provisoirement) notre exploration de 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 *

def move():
    "déplacement de la balle"
    global x1, y1, dx, dy, flag
    x1, y1 = x1 +dx, y1 + dy
    if x1 >360:
        x1, dx, dy = 360, 0, 15
    if y1 >360:
        y1, dx, dy = 360, -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():
    "arrêt de l'animation"
    global flag
    flag =0

def start_it():
    "démarrage de l'animation"
    global flag
    flag = flag +1      # préférable à flag =1 :
    if flag ==1:        # voir explications dans le texte
        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" (canevas + balle, boutons):
can1 = Canvas(fen1,bg='dark grey',height=400,width=400)
can1.pack(side=LEFT)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
bou1 = Button(fen1,text='Quitter',command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1,text='Démarrer',command=start_it)
bou2.pack()
bou3 = Button(fen1,text='Arrêter',command=stop_it)
bou3.pack()
# démarrage de l'observateur d'évènements (boucle principale) :
fen1.mainloop()
```

La seule nouveauté mise en oeuvre dans ce script se trouve tout à la fin de la définition de la fonction **move()** : vous y noterez l'application de la méthode **after**. Cette méthode peut s'appliquer à une fenêtre quelconque. Elle déclenche l'appel d'une fonction après qu'un certain laps de temps se soit écoulé. Ainsi **window.after(200,qqc)** déclenche pour la fenêtre **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 ainsi pour la première fois une technique de programmation que l'on appelle ***récursivité***. Pour faire simple, ***la récursivité est ce qui se passe lorsqu'une fonction s'appelle elle-même***. Il s'agit donc d'un bouclage, qui peut se perpétuer indéfiniment si nous ne prévoyons pas un moyen pour l'interrompre.

Voyons comment cela fonctionne dans notre exemple :
La fonction **move()** invoquée une première fois fait son travail (c.à.d. positionner la balle), puis 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... Tout au moins si nous n'avons pas pris la précaution de placer quelque part dans la boucle une instruction de sortie. C'est ce que nous avons réalisé dans notre script à l'aide d'un 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** passe à zéro, alors le bouclage ne s'effectue plus et l'animation s'arrête. **flag** étant une variable globale, nous pouvons aisément changer sa valeur à l'aide d'autres fonctions, que nous associerons aux boutons "Démarrer" et "Arrêter". Nous obtenons ainsi un mécanisme simple pour lancer ou arrêter l'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 et 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 valeur de **flag** augmente mais rien d'autre ne se passe, parce que **move()** n'est appelée que lorsque **flag** vaut 1. On évite ainsi le démarrage de plusieurs boucles concurrentes. Le bouton "Arrêter" remet **flag** à zéro et la boucle s'interrompt.

Exercices :

e 51. Dans la fonction **start_it()**, remplacez l'instruction **flag = flag +1** par **flag =1**.
Que se passe-t-il ?
(Cliquez plusieurs fois sur le bouton "Démarrer") Tâchez d'exprimer le plus clairement possible votre explication des faits observés.

e 52. Modifiez le programme de telle façon que la balle change de couleur à chaque "virage".

e 53. 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").

e 54. Modifiez le programme de manière à obtenir d'autres mouvements. Tâchez par exemple d'obtenir un mouvement circulaire. (Comme dans les exercices 7, 8, 9 page 67).

e 55. 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 !

e 56. Écrivez un programme dans lequel évoluent plusieurs balles de couleurs différentes, qui rebondissent les unes sur les autres ainsi que sur les parois.

e 57. 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...).

Chapitre 9 : Lire et écrire dans un fichier

9.1 Utilité des 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 les données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient totalement inadéquate lorsque l'on souhaite traiter une quantité d'information plus importante.

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 simples du type :

```
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 simple. Tout va se compliquer en effet lorsque nous essayerons d'imaginer la suite du programme, c.à.d. les instructions qui serviront 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 (et ce sera bien pénible à écrire :n'oubliez pas que nous voulons 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 meilleure que la première, on voit que la lisibilité n'est pas fameuse, et en outre nous sommes toujours confrontés à des 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 liste "kilométrique". 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 le programme lui-même, puisqu'il comportera de nombreuses lignes de données encombrantes.

- L'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) reste tout simplement impossible, puisque ces données font partie du programme lui-même.

Il est donc temps que nous apprenions séparer données et programmes dans des fichiers différents. Pour que cela soit possible, il faut que nos programmes disposent de divers mécanismes pour créer des fichiers, y envoyer des données et pouvoir les récupérer par après. Tous 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 vite nécessaire de structurer les relations entre ces données, et on doit alors élaborer des systèmes appelés **bases de données relationnelles** dont la gestion peut s'avérer très complexe. C'est là l'affaire de logiciels spécialisés tels que *Oracle, DB2, Adabas, PostgreSQL, MySQL,* etc. Python est parfaitement capable de dialoguer avec ces systèmes, mais nous laisserons cela pour un peu plus tard.

Nos ambitions présentes seront 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 et les y relire. Python dispose pour ce faire d'instructions puissantes et faciles à mettre en œuvre.

9.2 Ecriture séquentielle dans un fichier

Pour python, un fichier est rendu accessible par l'intermédiaire d'un **objet-fichier** que l'on crée à l'aide de la fonction interne **open()**. Après avoir appelé cette fonction, vous pouvez lire et écrire dans le fichier en appliquant des méthodes spécifiques à cet objet.

L'exemple ci-dessous vous montre comment ouvrir un fichier "en écriture", y enregistrer deux lignes de texte, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Si le nom de fichier utilisé concerne un fichier préexistant qui contient déjà des données, les lignes que nous y enregistrons viendront s'ajouter à la suite de celles qui s'y trouvent déjà. Notez aussi que nous faisons tout cet exercice directement à la ligne de commande.

```
>>> obFichier = open('Monfichier','a')
>>> obFichier.write('Bonjour, fichier texte !\n')
>>> obFichier.write("Quel beau temps, aujourd'hui !\n")
>>> obFichier.close()
>>>
```

Remarques :

- A la première ligne, on crée l'objet-fichier "obFichier", lequel fait référence à un fichier véritable (sur disque ou disquette) dont le nom sera "Monfichier". Ne confondez pas ce nom de fichier avec le nom de l'objet-fichier qui y donne accès. A la suite de cet exercice, vous pouvez aisément vérifier qu'il s'est bien créé sur votre système (dans le répertoire courant) un fichier dont le nom est "Monfichier" (et vous pouvez en visualiser le contenu à l'aide d'un éditeur quelconque). L'argument "**a**" indique qu'il faut ouvrir ce fichier en mode "ajout", ce qui signifie que les données à enregistrer seront ajoutées à la fin du fichier, à la suite de celles qui s'y trouvent déjà.
- Notez le marqueur de fin de ligne "\n" que nous avons inséré à la fin de chacune des deux lignes de texte à enregistrer. Sans ce marqueur, les deux chaînes de texte seraient enregistrées à la suite l'une de l'autre sans séparation (et l'ensemble serait donc considéré ensuite comme une seule longue chaîne).
- La méthode close() referme le fichier

9.3 Lecture séquentielle d'un fichier

Nous allons maintenant rouvrir le fichier, mais cette fois "en lecture", et y relire les informations que nous y avons enregistrées :

```
>>> ofi = open('Monfichier','r')
>>> a = ofi.readline()
>>> b = ofi.readline()
>>> print a
Bonjour, fichier texte !
>>> print b
Quel beau temps, aujourd'hui !
>>> ofi.close()
>>>
```

Remarques :

Le fichier que nous voulons lire s'appelle "Monfichier". L'instruction d'ouverture de fichier devra donc nécessairement faire référence à ce nom-là. Par contre, nous ne sommes tenus à aucune obligation concernant le nom à choisir pour l'objet-fichier. A la première ligne, nous créons donc un objet-fichier "ofi" faisant référence au fichier réel "Monfichier", lequel doit être ouvert en lecture (argument "**r**").

Les deux lignes suivantes appliquent chacune la méthode **readline()** à l'objet-fichier "ofi".
La méthode **readline()** effectue une **lecture séquentielle** du fichier : chaque fois qu'elle est appelée, elle retourne en effet le contenu de la ligne suivante du fichier (tant qu'il reste des lignes à lire, sinon elle retourne une chaîne vide). Des appels successifs à **readline()** vous fourniront donc toutes les lignes du fichier, dans l'ordre où elles ont été enregistrées. Si vous souhaitez lire les lignes dans un autre ordre, il vous faudra utiliser des méthodes plus sophistiquées, que nous ne décrirons pas ici.

9.4 Application

Veuillez à présent écrire, tester et analyser le petit script ci-dessous :

```
nomF = raw_input('Nom du fichier à traiter : ')
choix = raw_input('Entrez "e" pour écrire, "c" pour consulter les données : ')
if choix =='e':
    of = open(nomF, 'a')
    while 1:
        ligne = raw_input("entrez une phrase (<Enter> pour terminer): ")
        if ligne =='':
            break
        else:
            of.write(ligne + '\n')
else:
    of = open(nomF, 'r')
    while 1:
        ligne = of.readline()
        print ligne,
        if ligne =='':
            break
of.close()
```

Vous aurez remarqué que les deux boucles **while** utilisées dans ce script sont construites d'une manière différente de ce que vous avez rencontré précédemment. Vous savez que l'instruction **while** doit toujours être suivie d'une condition à évaluer. Le bloc d'instructions indenté qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie.

Or vous savez déjà[24] que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro. Une boucle **while** construite comme nous l'avons fait devrait donc boucler indéfiniment, puisque la condition de continuation reste toujours vraie. Pour interrompre ce bouclage, nous pouvons faire appel à l'instruction **break**, laquelle permet 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.
```

24 Voir page 39 : Véracité/fausseté d'une expression

Chapitre 10 : Fenêtres avec menus

Dans le programme que nous allons décrire ci-dessous, vous allez apprendre à construire une fenêtre d'application dotée de différents types de menus déroulants. Bien évidemment, cela vous permettra d'abord de compléter quelque peu votre documentation concernant les widgets *Tkinter* (nous sommes encore loin d'en avoir fait le tour !). Comprenez bien cependant que l'exercice est surtout destiné à vous faire réviser et approfondir un certain nombre de concepts.

Le script complet étant relativement long, nous allons réaliser ce travail par étapes, en appliquant une stratégie de programmation que l'on appelle **développement incrémental**.

Comme nous l'avons déjà expliqué précédemment[25], 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...

10.1.1 Première ébauche du programme

```
def effacer():
    can.delete(ALL)

def fileMenu():
    mbu = Menubutton(mBar, text ='Fichier')
    mbu.pack(side =LEFT)

    me1 = Menu(mbu)
    me1.add_command(label ='Effacer', underline =0, command =effacer)
    me1.add_command(label ='Terminer', underline =0, command =mBar.quit)
    mbu.configure(menu = me1)

from Tkinter import *
root = Tk()
root.title('Exemples de menus')
mBar = Frame(root, borderwidth =2)
mBar.pack(fill =X)
can = Canvas(root, bg='light grey', height=190, width=250, borderwidth =2)
can.pack()
fileMenu()

root.mainloop()
root.destroy()
```

Encodez ces lignes et lancez-en l'exécution. Vous devriez obtenir une simple fenêtre avec un

25 Voir page 10 : Recherche des erreurs et expérimentation

titre, une barre de menus contenant la seule rubrique "fichier", et un canevas gris clair.

Cliquez sur la rubrique "fichier" de la barre pour faire apparaître le menu correspondant. L'option "Effacer" n'est pas encore fonctionnelle (elle servira à effacer le contenu du canevas), mais l'option "Terminer" devrait déjà vous permettre de fermer proprement votre application. Comme tous les menus gérés par *Tkinter*, le menu que vous avez créée 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 ou bon vous semble sur le bureau.

Analyse du script

Le corps principal du programme ressemble beaucoup à ce que nous avons déjà rencontré précédemment : après importation du module Tkinter, on crée une fenêtre principale **root**, à laquelle on incorpore deux widgets "enfants" : un canevas **can** et un cadre (ou *Frame*) **mBar**, lequel servira en l'occurrence de barre de menus.

Pour définir les rubriques de ce menu, nous pourrions continuer à ajouter des instructions dans le corps principal du programme. Il est cependant beaucoup plus judicieux de regrouper toutes ces instructions dans des fonctions séparées (une fonction par rubrique). Ainsi nous donnerons à notre programme une véritable structure, et le produit fini sera à la fois plus lisible, plus fiable et plus facile à modifier éventuellement par la suite.

La fonction **fileMenu()** est donc ici un simple sous-programme servant à la création d'un widget de la classe **Menubutton**. Celui-ci est défini comme "enfant" du widget **mBar**, et on le positionne dans son cadre à l'aide de la méthode **pack()**. Comme son nom de classe l'indique, ce widget se comporte comme un bouton : une action se produira lorsque l'on cliquera dessus. Afin que cette action consiste en l'apparition d'un menu, il reste encore à définir celui-ci : ce sera encore un nouveau widget, de la classe **Menu** cette fois, défini comme "enfant" du widget Menubutton.

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 les deux *items* "Effacer" et "Terminer". Nous y intégrons l'option **underline**, qui définit un raccourci clavier : cette option indique en effet lequel des caractères de l'item devra apparaître souligné à l'écran. L'utilisateur sait ainsi 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 comme d'habitude par l'option **command**. Dans notre script, l'une des commandes appelle la fonction **effacer()**, que nous avons définie nous-même, et qui servira plus loin à vider le canevas. L'autre actionne la méthode prédéfinie **quit()**, laquelle peut être appliquée indifféremment à l'objet **mBar** ou à l'objet **root**. Cette méthode provoque la sortie de la boucle **mainloop()**, c.à.d. l'arrêt du gestionnaire d'événements associé à la fenêtre d'application.

Lorsque les items du menu ont été définis, il reste encore à reconfigurer le widget *Menubutton* de manière à ce que son option "menu" désigne effectivement le menu que nous venons de construire. En effet, nous ne pouvons pas déjà préciser cette option lors de la définition initiale du widget *Menubutton*, puisqu'à ce stade le menu n'existe pas encore. Nous ne pouvons pas non plus définir le menu en premier lieu, puisqu'il doit être défini comme "enfant" du widget *Menubutton*. On résout ce petit problème en utilisant la méthode **configure()**. Cette méthode peut être appliquée à n'importe quel widget préexistant pour en modifier l'une ou l'autre option.

Vous aurez constaté que nous avons dû définir deux variables intermédiaires **mbu** et **me1** pour référencer nos widgets. Étant donné qu'elles sont définies à l'intérieur d'une fonction, ces variables resteront strictement locales. De cette manière, nous aurons la possibilité de réutiliser les mêmes noms de variables ailleurs dans notre programme (si nous le souhaitons), sans risque d'interférence.

L'une des principales raisons d'utiliser des fonctions dans un script est qu'elles permettent de bien isoler les uns des autres les différents composants d'un programme.

Note : La méthode **destroy()** détruit le widget auquel elle est appliquée. Nous l'utilisons ici pour faire disparaître la fenêtre lorsque l'utilisateur décide de fermer l'application. Il ne faut invoquer cette méthode qu'après la sortie de **mainloop()**, de manière à quitter "proprement" l'application.

10.1.2 Ajout de la rubrique "Musiciens"

Continuons le développement de notre petit programme, en lui ajoutant les définitions de fonctions suivantes (au début du script) :

```
def showMusi17():
    can.create_text(10, 10, anchor=NW, text='H. Purcell',
                    font=('Times', 20, 'bold'), fill='yellow')

def showMusi18():
    can.create_text(245, 40, anchor =NE, text = "W. A. Mozart",
                    font =('Times', 20, 'italic'), fill ='dark green')

def musiMenu():
    mbu =Menubutton(mBar, text ='Musiciens')
    mbu.pack(side =LEFT, padx ='3')

    me1 =Menu(mbu)
    me1.add_command(label ='17e siècle', underline =1,
                    foreground ='red', background = 'yellow',
                    font =('Comic Sans MS',11),
                    command =showMusi17)
    me1.add_command(label ='18e siècle', underline =1,
                    foreground='royal blue', background ='white',
                    font =('Comic Sans MS',11,'bold'),
                    command =showMusi18)
    mbu.configure(menu = me1)
    return mbu
```

Pour que ces fonctions puissent servir à quelque chose, nous devons également ajouter une ligne dans le corps principal du programme (juste après la ligne **fileMenu()**) :

```
mBar.musi = musiMenu()
```

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 en inspirer pour vos travaux futurs. Les commandes que nous avons associées à ces items sont évidemment simplifiées pour ne pas alourdir l'exercice : elles provoquent l'affichage de petits textes sur le canevas.

Analyse du script

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, comme dans notre exemple, ou encore la définition de longues listes 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.

A la différence de la fonction **fileMenu()** que nous avions définie précédemment, la fonction

musiMenu() "retourne" une valeur : le contenu de la variable locale **mbu**. Cela va nous permettre de transmettre élégamment ce contenu ***local*** à la variable ***globale*** **mBar.musi** (sans qu'il soit nécessaire d'utiliser une instruction **global** à l'intérieur de la fonction).

Le contenu qui est transféré ainsi est la référence du widget de type Menubutton qui a été créé par la fonction **musiMenu()**. Nous devons mémoriser cette référence dans une variable globale, parce qu'ailleurs dans notre programme, il fous faudra pouvoir accéder à ce widget.

La variable globale dont nous parlons est elle-même définie d'une manière particulière. En effet : plutôt que de choisir pour cette variable un nom quelconque, nous avons utilisé le nom d'un objet préexistant auquel nous avons associé une extension (nous avons ajouté l'extension **musi** au nom du widget **mBar**, en les associant à l'aide d'un point). En procédant de cette manière, la variable que nous créons est définie comme ***une nouvelle propriété*** de l'objet **mBar**. On dira également que la variable **musi** a été ***encapsulée*** dans l'objet **mBar**.

Le terme d'objet a déjà été utilisé précédemment. Il s'agit d'un concept informatique essentiel, que nous continuerons à découvrir petit à petit. Les objets sont des entités caractérisées par un certain nombre de ***propriétés et de méthodes***. Les propriétés sont des variables, les méthodes sont des fonctions. Les unes et les autres sont ***encapsulées***, c.à.d. en quelque sorte enfermées dans un tout qui constitue l'objet proprement dit. Par exemple, vous savez déjà que les widgets *Tkinter* sont des objets. Les options que vous choisissez en les créant sont quelques-unes de leurs propriétés. Vous agissez sur les widgets en faisant appel à l'une ou l'autre de leurs méthodes.

On crée un objet par ***instanciation*** à partir d'une ***classe*** préexistante. (Par exemple, l'objet **mBar** a été créé à partir de la classe *Menubar*). On peut créer ainsi autant d'objets que l'on veut à partir de la même classe. Comme une simple variable, chacun reçoit un nom qui permet de le distinguer des autres. Grâce à l'encapsulation, chacun possède ses propriétés personnelles.

Pour accéder aux propriétés et aux méthodes d'un objet, on fait appel à un système de notation hiérarchisé qui relie les noms par des points. Ainsi, on accède à la propriété **musi** de l'objet **mBar** en utilisant le nom **mBar.musi**. Puisque cette propriété **musi** est elle-même la référence d'un objet (un objet de la classe *MenuButton* qui correspond à la rubrique "Musiciens" de la barre de menus), on pourra accéder aux propriétés et aux méthodes de cet objet en ajoutant un point supplémentaire. Par exemple, nous pourrons reconfigurer cet objet en lui appliquant la méthode **mBar.musi.configure()**, comme nous le verrons un peu plus loin.

10.1.3 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". Ajoutez donc les définitions suivantes au début du script :

```
def showRomanti():
    can.create_text(245, 70, anchor =NE, text = "E. Delacroix",
                    font =('Times', 20, 'bold italic'), fill ='blue')

def tabMonet():
    can.create_text(10, 100, anchor =NW, text = 'Nymphéas à Giverny',
                    font =('Technical', 20), fill ='red')

def tabRenoir():
    can.create_text(10, 130, anchor =NW, text = 'Le moulin de la galette',
                    font =('Dom Casual BT', 20), fill ='maroon')

def tabDegas():
    can.create_text(10, 160, anchor =NW, text = 'Danseuses au repos',
                    font =('President', 20), fill ='purple')
```

```
def peinMenu():
    mbu =Menubutton(mBar, text ='Peintres')
    mbu.pack(side =LEFT, padx='3')

    me1 =Menu(mbu)                                                  #
    me1.add_command(label ='classiques', state=DISABLED)            #
    me1.add_command(label ='romantiques', underline =0,             #
                    command = showRomanti)                          # voir
    me2 =Menu(me1)                                                  # texte
    me2.add_command(label ='Claude Monet', underline =7,            #
                    command =tabMonet)                              #
    me2.add_command(label ='Auguste Renoir', underline =8,          #
                    command =tabRenoir)                             #
    me2.add_command(label ='Edgar Degas', underline =6,             #
                    command =tabDegas)
    me1.add_cascade(label ='impressionistes ', underline =0,
                    menu = me2)
    mbu.configure(menu = me1)
    return mbu
```

... et n'oubliez pas d'ajouter également la ligne ci-dessous dans le corps principal du programme (juste après la ligne **mBar.musi = musiMenu()** :

```
mBar.pein = peinMenu()
```

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 définis comme un menu "enfant" du menu de niveau précédent (dans notre exemple, **me2** est défini comme un menu "enfant" de **me1**). L'intégration est assurée ensuite à l'aide de la méthode **add_cascade()**.

Pour clarifier le script, nous avons référencé ces menus et sous-menus dans de simples variables locales (nous n'avons pas l'intention d'y accéder ailleurs dans le programme). Comme nous allons le voir dans la rubrique suivante, nous aurions pu encapsuler ces références comme des propriétés de l'objet **mbu**, en encodant par exemple (les lignes marquées d'un #) :

```
    mbu.me1 =Menu(mbu)
    mbu.me1.add_command(label ='classiques', state=DISABLED)
    mbu.me1.add_command(label ='romantiques', underline =0,
                        command = showRomanti)
    mbu.me1.me2 =Menu(mbu.me1)
    mbu.me1.me2.add_command(label ='Claude Monet', underline =7,
                            command =tabMonet)
    mbu.me1.me2.add_command(label ='Auguste Renoir', underline =8,
                            command =tabRenoir)
    etc.
```

Ainsi l'objet **me1** serait encapsulé dans **mbu**, et l'objet **me2** encapsulé dans **me1**. Nous allons mettre cette idée en application dans la définition de la rubrique suivante.

10.1.4 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*. Veuillez donc ajouter les définitions ci-dessous au début de votre script :

```
def reliefBarre():
```

```
    choix = mBar.opt.me1.relief.get()
    mBar.config(relief =[FLAT,RAISED,SUNKEN,GROOVE,RIDGE,SOLID][choix])

def choixActifs():
    p = mBar.opt.me1.peint.get()
    m = mBar.opt.me1.music.get()
    mBar.pein.configure(state =[DISABLED, NORMAL][p])
    mBar.musi.configure(state =[DISABLED, NORMAL][m])

def optionsMenu():
    mbu = Menubutton(mBar, text ='Options')
    mbu.pack(side =LEFT, padx ='3')

    mbu.me1 = Menu(mbu)
    mbu.me1.relief =IntVar()
    mbu.me1.peint =IntVar()
    mbu.me1.music =IntVar()

    mbu.me1.add_command(label = 'Activer :', foreground ='blue')
    mbu.me1.add_checkbutton(label ='musiciens', command =choixActifs,
                            variable =mbu.me1.music)
    mbu.me1.add_checkbutton(label ='peintres', command =choixActifs,
                            variable =mbu.me1.peint)
    mbu.me1.add_separator()
    mbu.me1.add_command(label = 'Relief :', foreground ='blue')
    mbu.me1.add_radiobutton(label ='aucun', variable =mbu.me1.relief,
                            value =0, command = reliefBarre)
    mbu.me1.add_radiobutton(label ='sorti', variable =mbu.me1.relief,
                            value =1, command = reliefBarre)
    mbu.me1.add_radiobutton(label ='rentré', variable =mbu.me1.relief,
                            value =2, command = reliefBarre)
    mbu.me1.add_radiobutton(label ='rainure', variable =mbu.me1.relief,
                            value =3, command = reliefBarre)
    mbu.me1.add_radiobutton(label ='crête', variable =mbu.me1.relief,
                            value =4, command = reliefBarre)
    mbu.me1.add_radiobutton(label ='bordure', variable =mbu.me1.relief,
                            value =5, command = reliefBarre)
    mbu.configure(menu = mbu.me1)
    return mbu
```

... ainsi que les deux lignes suivantes dans le corps principal du programme (juste après la ligne **mBar.pein = peinMenu()**) :

```
mBar.opt =optionsMenu()
mBar.opt.me1.invoke(2)
```

Le programme est maintenant beaucoup plus complet. Les options ajoutées permettent d'activer ou désactiver à volonté les rubriques "Musiciens" et "Peintres". Vous pouvez également modifier à volonté l'aspect de la barre de menus elle-même.

Analyse du script

a) Qualification des noms

Vous pourrez constater au premier coup d'oeil que dans cette partie du programme, nous faisons abondamment appel à la notation hiérarchisée (noms reliés par des points). Les noms construits de cette manière sont souvent désignés dans la littérature comme étant des ***noms pleinement qualifiés***.

La première instruction de la fonction **reliefBarre()** constitue un bel exemple d'utilisation de cette logique. Cette fonction sert à redéfinir l'aspect général de toute la barre de menus, qui pourra apparaître plate, en relief, avec une bordure, etc. On y fait appel à la méthode **get()** , appliquée à l'option **relief** du widget *Menu* **me1** , lequel est lui-même intégré au widget *Menubutton* **opt** , celui-ci étant lui-même encapsulé dans le widget *Frame* **mBar**. (Nous aurions pu pousser la qualification plus loin encore, en définissant **mBar** comme une propriété de la fenêtre principale **root**, mais cela ne présenterait pas d'intérêt, puisqu'aucune partie de notre programme ne doit accéder à la fenêtre **root** depuis l'extérieur).

La méthode **get()** permet de récupérer l'état d'une **variable *Tkinter*** qui contient le numéro du choix opéré par l'utilisateur dans le sous-menu "Relief". Une telle variable est un peu particulière parce qu'elle sert d'interface entre les classes de la librairie *Tkinter* (développée à l'origine pour le language *Tcl*) et l'interpréteur Python. Nous verrons un peu plus loin ci-dessous comment définir une telle variable.

b) Contrôle du flux d'exécution à l'aide d'une liste

A la seconde ligne de la définition de la fonction **reliefBarre()**, 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 **mBar**.

La variable **choix** est donc utilisée ici comme index, pour 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:
    mBar.config(relief =FLAT)
elif choix ==1:
    mBar.config(relief =RAISED)
elif choix ==2:
    mBar.config(relief =SUNKEN)
...
etc.
```

D'un point de vue strictement fonctionnel, le résultat est exactement le même. Vous devez cependant considérer que la construction que nous avons choisie est plus efficiente. Imaginez par exemple que votre programme doive effectuer une sélection dans un grand nombre d'éléments : avec la construction ci-dessus, vous seriez peut-être amené à encoder plusieurs pages de "**elif**" !

Nous avons utilisé la même construction dans la fonction **choixActifs()**. Ainsi l'instruction :

```
mBar.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 fonction **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**, qui sont elles-mêmes le reflet des variables Tkinter que nous décrivons ci-après. (Les variables intermédiaires **m** et **p** servent seulement à clarifier le script).

c) Variables Tkinter

Dans la fonction optionsMenu(), nous définissons les rubriques du menu déroulant. Celui-ci comporte deux parties. Pour bien les mettre en évidence, nous avons inséré une ligne de séparation et deux "fausses rubriques" qui servent simplement de titres. Nous faisons apparaître ceux-ci en couleur afin que l'utilisateur ne les confonde pas avec de véritables commandes.

Les rubriques de la première partie sont dotées de "cases à cocher". Lorsque l'utilisateur effectue un clic de souris sur l'une ou l'autre de ces rubriques, les options correspondantes sont activées ou désactivées, et ces états "actif" / "inactif" sont affichés sous la forme d'une coche. Les instructions qui servent à mettre en place ce type de rubrique sont assez explicites. Elles présentent en effet ces commandes comme des widgets de type "chekbutton" :

```
mbu.me1.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. Cette variable n'est pas une variable ordinaire de Python, puisque la librairie Tkinter est en quelque sorte un logiciel indépendant avec lequel Python a établi un dialogue. Afin que ses utilisateurs puissent avoir accès à ses variables internes, Tkinter propose un mécanisme relativement simple :

- On commence par déclarer une ou plusieurs variables comme des nouvelles propriétés du widget Menu, à l'aide de la fonction prédéfinie **IntVar()** :

  ```
  mbu.me1.music =IntVar()
  ```

  La fonction IntVar() fait partie du module Tkinter que vous avez importé en totalité au début du script (Elle ne fait pas partie du module standard de Python). Elle permet de créer une variable de type entier, dans un format spécifique à Tkinter.

- Ensuite, on associe l'option "variable" de l'objet checkbutton à la variable ainsi définie :

  ```
  mbu.me1.add_checkbutton(label ='musiciens', variable =mbu.me1.music)
  ```

  Il est nécessaire de procéder ainsi en deux étapes, parce que pour le langage dans lequel a été écrit Tkinter (contrairement à ce qui se passe dans Python), une variable doit être d'abord définie à l'aide d'une instruction spécifique, avant que l'on ne puisse ensuite lui affecter une valeur.
  Vous rencontrerez cette contrainte dans de nombreux langages de programmation (*C++*, *Java*, *Clarion*, *Delphi*, etc.).

- Pour lire le contenu d'une variable Tkinter, il faudra également utiliser une fonction spécifique, la méthode get() , laquelle est associée au widget comme nous l'avons vu au point a) :

  ```
  m = mBar.opt.me1.music.get()
  ```

  Dans cette instruction, nous affectons à **m** (variable ordinaire de Python) le contenu d'une variable Tkinter 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 langages différents.

d) 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 spécialisés appelés "boutons radio". 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. L'enfoncement d'un bouton quelconque faisait en effet automatiquement ressortir tous les autres.

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 :

```
mbu.me1.add_radiobutton(label ='bordure', variable =mbu.me1.relief,
                        value =5, command = reliefBarre)
```

configure une rubrique du menu "Options" de telle manière qu'elle se comporte comme un bouton radio. Lorsque l'utilisateur sélectionnera cette rubrique, la valeur 5 sera affectée à la variable Tkinter **mbu.me1.relief**, et un appel sera lancé à la fonction **reliefBarre()**. Celle-ci récupérera la valeur mémorisée dans la variable Tkinter pour effectuer son travail.

Tout ceci serait plus simple si nous pouvions passer directement un argument à la fonction **reliefBarre()**, mais malheureusement cela n'est pas possible : l'option **command** associée aux widgets Tkinter n'autorise pas le passage d'arguments.

e) Pré-sélection d'une rubrique

Lorsque vous exécutez le script complet, 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.opt.me1.invoke(2)
```

Nous avons ajouté 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.opt.me1** 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" (Rappel : 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.

Chapitre 11 : Encore les structures de données

11.1 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 (ou *string*) 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 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.

11.1.1 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.

11.1.2 Indiçage, extraction, longueur

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.à.d. à partir de zéro. Pour extraire un caractère d'une chaîne, il suffit d'indiquer son indice **entre crochets** :

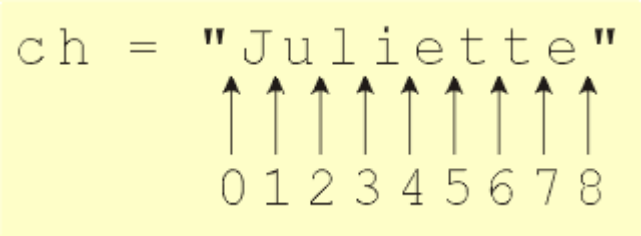
```
>>> nom = 'Cédric'
>>> print nom[1], nom[3], nom[5]
é r c
```

Il arrive aussi très 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 sauf les 3 premiers caractères
iette
```

Nous connaissons aussi déjà la fonction intégrée **len()**, qui retourne la longueur (c.à.d. le nombre de caractères) d'une chaîne :

```
>>> len(nom)
6
```

Exercices

e 58. Tâchez d'écrire une petite fonction **trouve()** qui fera exactement le contraire de ce que fait l'opérateur d'indexage (c.à.d. les crochets **[]**). Au lieu de partir d'un index donné pour retrouver le caractère correspondant, cette fonction devra trouver rechercher 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**

Veillez à ce que la fonction retourne la valeur **-1** si le caractère recherché n'existe pas dans la chaîne traitée.

e 59. 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 !)

e 60. Écrivez une fonction **comptecar()** qui compte le nombre d'occurrences d'un caractère donné dans une chaîne. Ainsi l'instruction :

print comptecar("Ananas au jus","a") devra afficher : **4**

11.1.3 Parcours d'une séquence. L'instruction for ... in ...

Il arrive très souvent que l'on doive traiter l'entièreté 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**. Sur la base des outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours sur la base de l'instruction **while** :

```
nom = 'Jacqueline'
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 bel et bien traité ainsi tous les caractères de la chaîne, puisque ceux-ci sont indicés de zéro à 9.

Le parcours d'une séquence est une opération tellement fréquente en programmation que Python vous propose une structure de boucle plus appropriée, basée sur l'instruction **for ... in ... :**

Avec cette instruction, le programme ci-dessus devient :

```
nom = 'Jacqueline'
for caract in nom:
    print caract + ' *',
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle nous évite d'avoir à définir et à incrémenter une variable spécifique pour gérer l'indice du caractère que nous voulons traiter à chaque itération. La 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 traitera 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']
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
```

L'instruction **for** 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.

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.à.d. 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 (rappel : dans le cas d'une liste, tous les éléments ne sont pas nécessairement du même type).

Exemple :

```
misc = ['cheval', 3, 17.25, [5, 'Jean']]
for e in misc:
    print e
```

L'exécution de ce script donne :

```
cheval
3
17.25
[5, 'Jean']
```

Bien que les éléments de la liste **misc** soient tous de types différents (une chaîne de caractères, 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 :

e 61. Dans un conte américain, huit petits canetons s'appellent respectivement : Jack, Kack, Lack, Mack, Nack, Oack, Pack et Qack. Écrivez un 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.

e 62. Rechercher le nombre de mots contenus dans une phrase donnée.

11.1.4 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. Par exemple, si vous essayer d'exécuter le petit script suivant :

```
salut = 'bonjour à tous'
salut[0] = 'B'
print salut
```

Au lieu d'afficher "Bonjour à tous", ce script "lèvera" 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 :

```
salut = 'bonjour à tous'
salut = 'B' + salut[1:]
print salut
```

Dans cet exemple en effet, nous (re)créons une nouvelle chaîne **salut** 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).

11.1.5 Les chaînes sont comparables

Tous les opérateurs de comparaison dont nous avons parlé à propos des instructions de contrôle de flux (c.à.d. les instructions **if ... elif ... else**) fonctionnent aussi avec les chaînes de caractères. Cela vous sera très 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 les caractères alphabétiques qui constituent une chaîne de caractères sont mémorisés dans la mémoire de l'ordinateur sous forme de nombres binaires dont la valeur est liée à la place qu'occupe le caractère dans l'alphabet. Dans le système de codage ASCII, par exemple, A=65, B=66, C=67, etc.[26]

11.1.6 Classement des caractères

Il est souvent utile de pouvoir déterminer si tel caractère extrait d'une chaîne est une lettre majuscule ou minuscule, ou même plus généralement de déterminer s'il s'agit bien d'une lettre, d'un chiffre, ou encore d'un autre caractère typographique.

Nous pouvons bien entendu écrire différentes fonctions pour assurer ces tâches. Par exemple, la fonction ci-dessous retourne "vrai" si l'argument qu'on lui passe est une minuscule :

```
def minuscule(ch):
     if 'a' <= ch <= 'z' :
          return 1
     else:
          return 0
```

Exercices :

Note : dans les exercices ci-après, omettez délibérément les caractères accentués et spéciaux.

e 63. Écrivez une fonction **majuscule()** qui retourne "vrai" si l'argument transmis est une majuscule.

e 64. Écrivez une fonction qui retourne "vrai" si l'argument transmis est un caractère alphabétique quelconque (majuscule ou minuscule). Dans cette nouvelle fonction, utilisez les fonctions **minuscule()** et **majuscule()** définies auparavant.

e 65. Écrivez une fonction qui retourne "vrai" si l'argument transmis est un chiffre.

e 66. Écrivez une fonction qui compte le nombre de caractères majuscules dans une phrase donnée.

26 En fait, il existe plusieurs systèmes de codage : les plus connus sont les codages ASCII et ANSI, assez proches l'un de l'autre sauf en ce qui concerne les caractères particuliers spécifiques des langues autres que l'anglais (caractères accentués, cédilles, etc.). Un nouveau système de codage intégrant tous les caractères spéciaux de toutes les langues mondiales est apparu depuis quelques années. Ce système appelé *unicode* devrait s'imposer petit à petit. Python l'intègre à partir de sa version 2.

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 comme argument. En retour, elle fournit le code ASCII correspondant à ce caractère. Ainsi **ord('A')** retournera la valeur **65**.

La fonction **chr(num)** fait exactement le contraire. L'argument qu'on lui transmet doit être un entier compris entre 0 et 255. En retour, on obtient le caractère ASCII correspondant :
Ainsi **chr(65)** retourne le caractère **A**.

Exercices :

Note : dans les exercices ci-après, omettez délibérément les caractères accentués et spéciaux.

e 67. 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 les relations numériques reliant chaque caractère majuscule à chaque caractère minuscule.

e 68. A partir des relations trouvées dans l'exercice précédent, écrivez une fonction qui convertit tous les caractères d'une phrase donnée en minuscules.

e 69. A partir des mêmes relations, écrivez une fonction qui convertit tous les caractères minuscules en majuscules, et vice-versa (dans une phrase fournie en argument).

e 70. Écrivez une fonction qui compte le nombre de fois qu'apparaît tel caractère (fourni en argument) dans une phrase donnée.

e 71. Rechercher le nombre de voyelles contenues dans une phrase donnée.

11.1.7 Le module string. Les deux formes d'importation.

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, comme nous allons le voir ci-dessous. Chacune présente des avantages et des inconvénients.

Voici un exemple de la première méthode :

```
>>> import string
>>> mots = string.split('Salut les copains')
>>> mots
['Salut','les','copains']
```

La première ligne de cet exemple importe ***l'entièreté*** du module string, lequel contient de nombreuses fonctions intéressantes. A la seconde ligne, nous utilisons la fonction **split()** du module **string**. (Le point séparateur exprime une relation d'appartenance. Nous retrouvons bien évidemment ici la logique de ***qualification des noms*** déjà expliquée dans les chapitres précédents). Comme vous pouvez le constater, la fonction **split()** traite une chaîne de caractères comme une phrase, dont elle extrait tous les mots pour les placer dans une liste.

Par comparaison, voici un exemple similaire utilisant la seconde méthode d'importation :

```
>>> from string import split
>>> mots = split('Salut les copines')
>>> mots
```

```
['Salut','les','copines']
```

Dans ce nouvel exemple, nous n'avons importé du module string que la fonction **split()** 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 string.

Nous pourrions de la même manière importer plusieurs fonctions du même module :

```
from string import split, find, count
```

voire même les importer toutes, 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 d'ailleurs 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. 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.à.d. 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).

Le module string propose un grand nombre de fonctions intéressantes pour le traitement des chaînes de caractères. Des fonctions de conversion (du type *chaîne* vers les types *entier* ou *réel*, ou l'inverse), des fonctions de recherche et de comptage (recherche d'une sous-chaîne dans une chaîne, comptage de caractères), des fonctions de fractionnement et de jointure (comme la fonction split() utilisée ci-dessus), ainsi que des fonctions de formatage (centrage d'une chaîne dans un champ, suppression des espaces au début ou à la fin, conversions minuscules/majuscules, etc.).

Exemples :

Recherche de la position d'une sous-chaîne dans une chaîne plus grande :

```
>>> import string
>>> ch1 = "Cette leçon vaut bien un fromage, sans doute ?"
>>> ch2 = "fromage"
>>> print string.find(ch1,ch2)
25
```

"Capitalisation", c.à.d. mise en majuscule de la première lettre de chaque mot :

```
>>> ch3 = string.capwords(ch1)
>>> print ch3
Cette Leçon Vaut Bien Un Fromage, Sans Doute ?
```

Comptage du nombre de sous-chaînes dans une chaîne :

```
>>> ch1 = "Le héron au long bec emmanché d'un long cou"
>>> ch2 = 'long'
>>> print string.count(ch1,ch2)
2
```

Veuillez donc consulter la documentation en ligne de Python (Library reference), ou encore le petit aide-mémoire : *Python précis et concis* (par Mark Lutz – Editions O'Reilly) pour une description détaillée de ces fonctions. Il ne nous paraît pas utile d'incorporer toutes ces informations dans les présentes notes.

11.2 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).

11.2.1 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]
```

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), alors qu'un élément isolé peut contenir n'importe quel type de donnée. Nous allons approfondir cette distinction tout au long des exemples suivants.

11.2.2 Les listes sont modifiables

Contrairement aux chaînes de caractères, les listes sont des séquences modifiables. On peut ourra donc construire des grandes listes de manière dynamique (à l'aide d'un algorithme quelconque).

Exemples :

```
>>> nombres[0] = 17
>>> nombres
[17, 38, 10, 25]
```

Dans l'exemple ci-dessus, on a donc remplacé le premier élément de la liste **nombres**, en utilisant l'opérateur **[]** à la gauche du signe égale (opérateur d'affectation).

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 numéro 1 d'une liste qui est elle-même l'élément numéro 3 d'une autre liste : la liste **stuff**.

11.2.3 Les listes sont des objets

Dans les chapitres précédents, nous avons déjà rencontré de nombreux objets. Vous savez donc que l'on peut agir sur un objet à l'aide de méthodes (c.à.d. des fonctions associées à l'objet).

Sous Python, les listes sont des objets à part entière, et vous pouvez par exemple leur appliquer un certain nombre de méthodes particulièrement efficaces :

```
>>> 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 sera effacé).

11.2.4 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 fonction (**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 :

a) 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 deux particularités suivantes :

1. 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.à.d. deux index réunis par le symbole :), et non un élément isolé dans cette liste.

2. L'élément que vous fournissez à la droite du signe égale doit lui-même être une liste.

Vous comprendrez mieux ces contraintes en analysant ce qui suit :

b) Suppression / remplacement d'éléments

```
>>> mots[2:5] = []                          # [] désigne une liste vide
>>> mots
['saucisson', 'ketchup', 'confiture', 'chocolat']
>>> mots[1:3] = ['salade']
>>> mots
['saucisson', 'salade', 'chocolat']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['saucisson', '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.

11.2.5 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. 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 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.

11.2.6 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
```

11.2.7 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)`**

11.2.8 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
```

11.2.9 Copie d'une liste

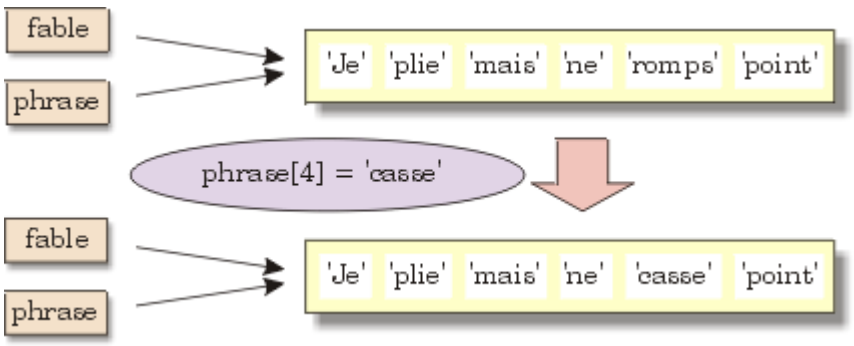
Considérons que vous souhaitiez réaliser une copie de la liste **fable** définie dans l'exemple précédent, et placer cette copie dans une nouvelle variable que vous appellerez **phrase**. La première idée qui vous viendra à l'esprit sera certainement d'écrire une simple affectation telle que :

```
>>> 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[1] ='souffre'
>>> phrase
['Je', 'souffre', 'mais', 'ne', 'romps', '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.

Exercices :

e 72. Créer une liste **A** contenant quelques éléments. Effectuer une vraie copie de cette liste dans une nouvelle variable **B**. Suggestion : créer d'abord une liste **B** de même taille que **A** mais ne contenant que des zéros. Remplacer ensuite tous ces zéros par les éléments tirés de **A**.

e 73. Même question, mais autre suggestion : créer d'abord une liste **B** vide. La remplir ensuite à l'aide des éléments de **A** ajoutés l'un après l'autre.

e 74. Même question, autre suggestion encore : pour créer la liste **B**, découper dans la liste **A** une tranche incluant tous les éléments (à l'aide de l'opérateur **[:]**).

11.2.10 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 pouvons 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.à.d. 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 :

e 75. 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).

e 76. 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 :

e 77. 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 :

a) 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).

b) Demander à l'utilisateur en combien de fractions il souhaite partager l'intervalle des valeurs possibles (c.à.d. l'intervalle de 0 à 1). Ici aussi, il faudrait proposer un nombre de par défaut (5 fractions, par exemple). Vous pouvez également limiter le choix de l'utilisateur à un nombre compris entre 2 et le 1/10^{e} du nombre de valeurs tirées au hasard.

c) Construire une liste de N compteurs (N étant le nombre de fractions souhaitées). Chacun d'eux sera évidemment initialisé à zéro.

d) Tirer au hasard toutes les valeurs demandées, à l'aide de la fonction **random() ,** et mémoriser ces valeurs dans une liste.

e) Mettre en oeuvre 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.

f) 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, et le nombre de fractions choisi par l'utilisateur) et fournirait en retour l'index du compteur à incrémenter (c.à.d. 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").

Chapitre 12 : Classes et objets

Les chapitres précédents vous ont déjà mis en contact à plusieurs reprises avec la notion d'objet. Vous savez déjà qu'un objet est toujours construit par **instanciation** à partir d'une **classe** (c.à.d. en quelque sorte une "catégorie" ou un "type" d'objet). Par exemple, on peut trouver dans la librairie *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 nous pouvons nous-mêmes définir de nouvelles classes d'objets. Il s'agit là d'un sujet relativement ardu, mais nous allons l'aborder de manière très progressive, en commençant par définir des classes d'objets très simples, que nous allons perfectionner ensuite. Attendez-vous cependant à rencontrer dans la suite de votre apprentissage des objets de plus en plus complexes. Comme les objets de la vie courante, les objets informatiques peuvent être très simples ou très compliqués. Ils peuvent être composés eux-mêmes de différentes parties, qui soient elles-mêmes des objets, ceux-ci étant faits à leur tour d'autres objets plus simples, etc.

12.1 Données composites définies par le programmeur

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction **class**. Nous allons donc apprendre à utiliser cette instruction, et pour commencer nous allons définir un type d'objet élémentaire, 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 mathématique de **point**. Dans un espace à deux dimensions, un point est caractérisé par deux nombres (ses coordonnées). En notation mathématique, on représente souvent 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 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:
      "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()**. Remarques :

- 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. (Par convention, si la première ligne suivant l'instruction **class** est une chaîne de caractères, celle-ci pourra être incorporée automatiquement dans un dispositif de documentation des classes qui fait partie intégrante de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit. Faites de même pour les fonctions).
- Rappelez-vous aussi la convention qui consiste à toujours donner aux classes des noms qui commencent par une majuscule. Dans la suite de ce texte, nous respecterons en outre une autre convention qui consiste à associer à chaque nom de classe une paire de parenthèses, comme nous le faisons déjà pour les noms de fonctions.

Nous venons de définir une classe **Point()**. Il nous reste maintenant à nous en servir pour créer des objets de type, par instanciation. Créons par exemple un nouvel objet **p9** :

```
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 être appelées avec des arguments.

12.2 Variables d'instance

Nous pouvons ajouter des composants à un objet (une instance) en utilisant le système de qualification des noms par points :

```
p9.x = 3.0
p9.y = 4.0
```

Les variables ainsi définies sont des **variables d'instance**. Elles sont incorporées, ou plutôt **encapsulées** dans l'objet. Vous rencontrerez de nombreux textes où l'on désigne aussi ces variables sous le nom d'**attributs** de l'instance. On peut les utiliser dans n'importe quelle expression, comme toutes les variables ordinaires :

```
>>> print p9.x
3.0
>>> print p9.x**2 + p9.y**2
25.0
```

Que se passe-t-il si nous essayons d'afficher l'instance elle-même ?

```
>>> print p9
<__main__.Point instance at 00B1C64C>
```

Ceci indique, comme vous l'aurez certainement bien compris tout de suite, que **p9** est une instance de la classe **Point()** et qu'elle est définie au niveau principal du programme. Elle a reçu de Python un identifiant unique, qui apparaît ici en notation hexadécimale (veuillez consulter votre cours d'informatique générale à ce sujet).

12.3 Passage d'objets comme arguments

Les fonctions peuvent utiliser des objets comme paramètres (elles peuvent également fournir un objet comme valeur de retour). Par exemple, vous pouvez définir une fonction telle que celle-ci :

```
>>> 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()**, puisque l'instruction qui suit utilise 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
```

Exercice :

e 78. Ecrivez une fonction **distance()** qui permette de calculer la distance entre deux points.
Cette fonction attendra évidemment deux objets **Point()** comme arguments.

12.4 Objets composés d'objets

Supposons maintenant que nous voulions définir une classe pour 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 méthode qui soit retenue.

Définissons donc notre nouvelle classe :

```
class Rectangle:
      "définition d'une classe de rectangles"
```

... et servons nous-en tout de suite pour créer une instance :

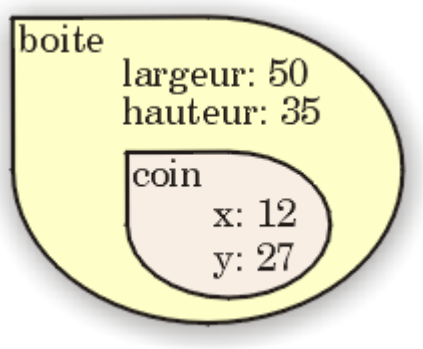
```
boite = Rectangle()
boite.largeur = 50.0
boite.hauteur = 35.0
```

Nous créons ainsi un nouvel objet **Rectangle()** et deux variables d'instance. Pour spécifier le coin supérieur gauche, nous allons utiliser une 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
```

Pour accéder à un objet qui se trouve à l'intérieur d'un autre objet, on utilise la qualification des noms hiérarchisée (à l'aide de points) que nous avons déjà rencontrée à plusieurs reprises.

Vous pourrez peut-être mieux vous représenter à l'avenir les objets composites, à l'aide de diagrammes similaires à celui que nous reproduisons ci-contre.



12.5 Méthodes

Pour aller un peu plus loin, nous allons définir encore une nouvelle classe, destinée cette fois à mémoriser des instants, des durées, etc. :

```
class Time:
     "Définition d'une classe temporelle"
```

Créons à présent un objet de ce type, et adjoignons-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 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**

Votre fonction ressemblera probablement à ceci :

```
>>> def affiche_heure(t):
        print str(t.heure) + ":" + str(t.minute) + ":" + str(t.seconde)
```

Si par la suite vous utilisez fréquemment des objets de la classe **Time()**, il y a gros à parier que cette fonction d'affichage vous sera fréquemment utile.

Il serait donc probablement fort judicieux **d'encapsuler** cette fonction dans la classe **Time()**, de manière à s'assurer qu'elle soit toujours automatiquement disponible chaque fois que l'on doit manipuler des objets de la classe **Time()**.

Une fonction qui est ainsi encapsulée dans une classe s'appelle une **méthode**. Vous avez déjà rencontré des méthodes à de nombreuses reprises (et vous savez donc déjà qu'une méthode est bien une fonction associée à une classe d'objets).

On définit une méthode comme on définit une fonction, avec cependant deux différences :

- ***La définition d'une méthode doit être 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.
- ***Le premier paramètre utilisé par une méthode est toujours le mot réservé*** " **self** ".
  Ce mot réservé désigne l'instance à laquelle la méthode sera associée, dans les instructions internes à la définition. (La définition d'une méthode comporte donc toujours au moins un paramètre, alors que la définition d'une fonction peut n'en comporter aucun).

Voyons maintenant comment cela se passe en pratique. Pour réécrire la fonction **affiche_heure()** comme une méthode, nous allons donc déplacer sa définition à l'intérieur de celle de la classe, et changer le nom de son paramètre :

```
class Time:
    "Nouvelle classe temporelle"
    def affiche_heure(self):
        print str(self.heure) + ":" + str(self.minute) \
              + ":" + str(self.seconde)
```

La définition de la méthode fait maintenant partie du bloc d'instructions indentées après l'instruction **class**. Notez bien l'utilisation du mot réservé **self** , qui se réfère donc à toute instance susceptible d'être créée à partir de cette classe. (Le code \ permet de continuer une instruction trop longue sur plusieurs lignes).

Nous pouvons à présent instancier un objet de cette classe :

```
>>> maintenant = Time()
```

Si nous essayons d'utiliser un peu trop vite notre nouvelle méthode, ça ne marche pas :

```
>>> maintenant.affiche_heure()
AttributeError: 'Time' instance has no attribute 'heure'
```

C'est normal : nous n'avons pas encore créé les variables d'instance. Il faudrait faire par exemple :

```
>>> maintenant.heure = 13
>>> maintenant.minute = 34
>>> maintenant.seconde = 21
>>> maintenant.affiche_heure()
13:34:21
```

12.6 La méthode _init_ : valeurs par défaut pour les variables d'instance

L'erreur que nous avons rencontrée au paragraphe précédent n'est pas très plaisante. Il serait sans doute préférable que la méthode **affiche_heure()** puisse toujours afficher quelque chose, même si nous n'avons encore fait aucune manipulation sur l'objet nouvellement créé. En d'autres termes, il serait judicieux que les variables d'instance soient définies elles aussi à l'intérieur de la classe, avec pour chacune d'elles une valeur "par défaut".

Pour obtenir cela, nous devons faire appel à une méthode particulière, que Python exécutera automatiquement lors de l'instanciation d'un objet à partir de sa classe. Une telle méthode est souvent appelée un ***constructeur***. On peut y placer tout ce qui peut être nécessaire pour initialiser automatiquement l'objet que l'on crée. Sous Python, une méthode est automatiquement reconnue comme un constructeur si on lui donne le nom réservé **__init__** (deux caractères "souligné", le mot **init**, puis encore deux caractères "souligné").

Exemple :

```
class Time:
     "Encore une nouvelle classe temporelle"
     def __init__(self):
          self.heure =0
          self.minute =0
          self.seconde =0

     def affiche_heure(self):
          print str(self.heure) + ":" + str(self.minute) \
                + ":" + str(self.seconde)

>>> tantot = Time()
>>> tantot.affiche_heure()
0:0:0
```

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. Ceux-ci vont jouer un rôle important, parce qu'ils vont permettre d'instancier un objet et d'initialiser certaines de ses variables d'instance en une seule opération. Dans l'exemple ci-dessus, veuillez donc modifier 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
```

Les arguments qui seront transmis à la méthode **__init__()** sont ceux que nous placerons dans les parenthèses qui accompagnent le nom de la classe, dans l'instruction d'instanciation.

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 créer de tels objets **Time()** en omettant une partie des arguments :

```
>>> rentree = Time(10, 30)
>>> rentree.affiche_heure()
10:30:0
```

12.7 Espaces de noms des classes et instances

Vous avez appris précédemment (voir page 47) 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 lecture 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 seront appelées les attributs de la 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 plus haut. A la fin de la page précédente, nous avons instancié deux objets de cette classe : **recreation** et **rentree**. Chacun a été initialisé avec des valeurs différentes, indépendantes. Nous pouvons modifier et réafficher ces valeurs à volonté dans chacun de ces deux 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:                              # 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 a, Espaces.a, essai.a                # 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.

12.8 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 possédera quelques fonctionnalités supplémentaires. 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 contiendra un ensemble de caractéristiques propres à ce type d'animal. A partir de cette classe, nous pourrons alors dériver une classe **Primate()**, une classe **Rongeur()**, une classe **Carnivore()**, etc., qui ***hériteront*** de toutes les caractéristiques de la classe **Mammifere()**, en y ajoutant leurs spécificités.

Au départ de la classe **Carnivore()**, nous pourrons ensuite dériver une classe **Belette()**, une classe **Loup()**, une classe **Chien()**, etc., qui hériteront encore une fois de toutes les caractéristiques de la classe parente avant d'y ajouter les leurs.

Exemple :

```
class Mammifere:
    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 de l'attribut défini pour cette classe, mais également des 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 ensuite on place entre parenthèses le nom de la classe parente.

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 homonymes 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.

12.9 Formatage des chaînes de caractères

Avant d'aller plus loin, il nous semble utile d'ouvrir ici une petite parenthèse pour vous présenter les fonctionnalités de Python en ce qui concerne la mise en forme d'une chaîne de caractères élaborée à partir de morceaux divers.

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, et la température dans une variable de type *float*. Vous souhaitez à présent que votre programme construise une 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 la fonction **str()** pour convertir en chaîne de caractères la valeur numérique contenue dans la variable de type *float* (faites l'exercice).

Python vous offre d'autres possibilités. Vous pouvez construire une chaîne en assemblant deux éléments à l'aide de l'opérateur % : à gauche vous fournissez une chaîne de format (un patron, en quelque sorte) qui contient des marqueurs de conversion, et à droite (entre parenthèses) un ou plusieurs 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 format 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, ...). Vous pouvez expérimenter d'autres mises en forme en utilisant d'autres marqueurs. Essayez par exemple de remplacer le deuxième `%s` par `%d` , ou par `%f` , ou encore par `%8.2g` . Le marqueur `%d` attend un nombre et le convertit en entier ; les marqueurs `%f` et `%g` attendent des réels et peuvent déterminer la largeur et la précision qui seront affichées.

La description complète de toutes les 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.

12.10 Modules contenant des librairies de classes

Vous connaissez déjà depuis longtemps l'utilité des modules Python. Vous savez qu'ils servent à regrouper des librairies de classes et de fonctions. A titre d'exercice de révision, vous allez encoder les lignes d'instruction ci-dessous dans un module (c.à.d. un fichier) que vous nommerez **formes.py** :

```
class Rectangle:
    "Classe de rectangles"
    def __init__(self, longueur =0, largeur =0):
        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):
        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 :

```
>>> 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()** hérite de toutes les caractéristiques de la classe **Rectangle()**. 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 (auquel cas les instructions qui suivent doivent être exécutées), ou au contraire utilisé comme une librairie de classes importée ailleurs. Dans ce cas cette partie du code est sans effet.

12.11 Application au cas des objets graphiques

Le domaine des objets graphiques est évidemment un domaine de choix pour l'application de tous ces concepts. Des librairies de classes comme *Tkinter* ou *wxPython* fournissent une base de widgets déjà fort étoffée, que vous pouvez adapter à vos besoins par dérivation.

Code source pour la fenêtre ci-contre :

```
from Tkinter import *

class GUI:
   def __init__(self):
      self.root = Tk()                                                    # 1
      self.root.title('Frames')                                           # 2
      self.fgen = Frame(self.root, bg = '#80c0c0')                        # 3
      self.fint = [0]*7                                                   # 4
      for n, rel, txt in [(0, RAISED, 'Relief sortant'),                  # 5
                          (1, SUNKEN, 'Relief rentrant'),
                          (2, FLAT, 'Pas de relief'),
                          (3, RIDGE, 'Bordure'),
                          (4, GROOVE, 'Rainure')]:
         self.fint[n] = Frame(self.fgen, borderwidth =2, relief =rel)     # 6
         Label(self.fint[n], text =txt, width =20).pack(side =LEFT)       # 7
         self.fint[n].pack(padx =15, pady =5)                             # 8
      self.fgen.pack()                                                    # 9

myGUI = GUI()                                                             # 10
myGUI.root.mainloop()                                                     # 11
```

Cette petite application nous montre une manière classique de dériver la classe **Tk()** pour créer une classe de fenêtres personnalisées.

Dans le constructeur de la classe **GUI()**, on instancie divers widgets Tk : une fenêtre contenant un cadre coloré (*frame*), ce cadre contenant lui-même 5 autres cadres plus petits qui contiennent chacun une étiquette (*label*).

La couleur du cadre est déterminée par l'argument bg (*background*) utilisé à la ligne 3. 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. Le noir serait obtenu avec #000000, le blanc avec #ffffff, le rouge pur avec #ff0000, un bleu sombre avec #000050, etc.

Notez bien la construction de la ligne 5 qui parcourt une liste constituée de 5 tuples. Chacun de ces tuples est constitué de 3 éléments : un entier, une constante (prédéfinie dans Tkinter), une chaîne de caractères. La boucle **for** effectue 5 itérations pour parcourir les 5 éléments de la liste. A chaque itération, le contenu d'un des tuples est affecté aux variables **n**, **rel** et **txt** , et les instructions des lignes 6, 7 & 8 sont exécutées.

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 principal bleu cyan. Un petit espace est réservé autour de chaque petit cadre grâce aux arguments **padx** et **pady**.